Exception Handling for Copyless Messaging

Exception Handling for Copyless Messaging

Svetlana Jakšić

Univerzitet u Novom Sadu, Fakultet tehničkih nauka
Trg Dositeja Obradovića 6, 21000 Novi Sad, Serbia
email: sjaksic@uns.ac.rs

Luca Padovani

Università di Torino, Dipartimento di Informatica
Corso Svizzera 185, 10149 Torino, Italy
email: padovani@di.unito.it

Abstract

Copyless messaging is a communication paradigm in which only pointers to messages are exchanged between sender and receiver processes. Because of its intrinsically low overhead, copyless messaging is suitable for the efficient implementation of communication-intensive software systems where processes have access to a shared address space. Unfortunately, the very nature of the paradigm fosters the proliferation of programming errors due to the explicit use of pointers and to the sharing of data. In this paper we study a type discipline for copyless messaging that, together with some minimal support from the runtime system, is able to guarantee the absence of communication errors, memory faults, and memory leaks in presence of exceptions. To formalize the semantics of processes we draw inspiration from software transactional memories: in our case a transaction is a process that is meant to accomplish some exchange of messages and that should either be executed completely, or should have no observable effect if aborted by an exception.

Keywords: Copyless message passing, Session types, Exception handling, Memory leak prevention

1. Introduction

Communication has become a central aspect of all modern software systems, which range from distributed processes connected by wide area networks down

to collections of threads running on different cores within the same processing unit. In all these scenarios, message passing is a flexible paradigm that allows autonomous entities to exchange information and to synchronize with each other. The term "message passing" seems to suggest a paradigm where messages *move* from one entity to another, although more often than not messages are in fact *copied* during communication. While this is inevitable in a distributed setting, the availability of a shared address space makes it possible to implement a *copyless* form of message passing, whereby only pointers to messages are exchanged.

The Singularity Operating System (Singularity OS) [1, 2] is a notable example of system that heavily relies on the copyless paradigm. In Singularity OS, processes have access to a shared region called the *exchange heap*, inter-process communication solely occurs by means of message passing over channels allocated on the exchange heap, and messages are themselves pointers to the exchange heap. As detailed by Hunt et al. [1], Hunt and Larus [2], Finley [3], it is not practical to automatically garbage collect objects on the exchange heap, which therefore must be explicitly managed by processes.

The copyless paradigm has obvious performance advantages over more conventional forms of message passing. At the same time, it fosters the proliferation of subtle programming errors arising from the explicit management of objects and the sharing of data. For this reason the designers of Singularity OS have equipped Sing$^\#$, the programming language used for the development of Singularity OS, with explicit constructs, types, and static analysis techniques to assist programmers in writing code that is free from a number of programming errors, including: *memory faults*, namely the access to unallocated/deallocated objects in the heap; *memory leaks*, that is the accumulation of unreachable allocated objects in the heap; communication errors, which could cause the abnormal termination of processes and trigger the previous kinds of errors.

Some aspects of Sing$^\#$ have already been formalized and studied by Fähndrich et al. [4], Stengel and Bultan [5], Villard et al. [6], Bono and Padovani [7]. In particular, in [7] it was shown that Sing$^\#$ *channel contracts* can be conveniently represented as a variant of session types [8, 9], and that the information given by session types along with a linear type discipline can prevent memory leaks, memory faults, and communication errors. In the present paper we focus on *exceptions* and *exception handling*. The interest in our research stems from the observation that copyless messaging and exceptions are clearly at odds with each other: on the one hand, copyless messaging requires a very disciplined and controlled access to memory; on the other hand, exceptions are in general unpredictable and disrupt the normal control flow of programs. Consequently, and perhaps not sur-

prisingly, these two aspects can be reconciled only with some native support from the runtime system. Below is a summary of the contributions of this research:

- we formalize a calculus of processes that communicate through the copyless paradigm and that can throw exceptions;
- we develop a type system for preventing the aforementioned errors even in the presence of exceptions, if suitable exception handlers are provided;
- we show how to take advantage of the invariants guaranteed by the type system in order to reduce the cost of exception handling.

Origin of the material. An early version of this paper appeared in the proceedings of the PPDP 2012 conference [10]. The present paper corresponds to a revised version of [10] where we provide detailed proofs of all the results and we generalize the process language (with multiple exceptions and exception propagation) as well as the type system (with subtyping). In particular, we show that the exception annotations within types induce an original subtyping relation that sheds light on the differences between exceptions and regular messages.

Structure of the paper. In Section 2 we illustrate the problem we are addressing and informally sketch our solution in terms of types and a revised exception handling construct. In Section 3 we formally define the syntax and the semantics of a language of processes to model Sing$^{\#}$ programs. The section ends with the definition of *well-behaved processes*, namely of those processes in which memory faults, memory leaks, and communication errors do not occur. Section 4 develops a type system for the process language presented in Section 3 and shows its soundness (well-typed processes are well behaved). Section 5 provides an overview of related work and discusses similarities and differences between our approach and similar ones. Section 6 concludes with a brief summary of the work and hints at further relaxations of the typing rules, in light of the common pattern usage of exception handling mechanisms as found in the source code of Singularity OS. For the sake of readability, the proofs of subject reduction and of type soundness, which are fairly long and require a number of auxiliary lemmas, have been moved to Appendix A and B, respectively.

2. Motivating Example

To introduce the context in which we operate and the kind of problems we have to face we take a look at a real fragment of Singularity OS. In the discussion that

follows it is useful to keep in mind that Singularity channels consist of pairs of related *endpoints*, called the *peers* of the channel. Each endpoint is associated with an unbounded queue containing the messages sent to that endpoint from its peer. Communication is therefore asynchronous and send operations are nonblocking.

Figure 1 shows a Sing# function that computes the name for a newly allocated RAM disk.[1] The function has two output parameters, the computed disk name and the endpoint that links the disk to the `DirectoryService` (abbreviated `DS` in the code) which is part of the file system manager. The function begins by retrieving an endpoint `ns` for communicating with `DirectoryService` (line 3). Then the function repeatedly creates a new channel, represented as the peer endpoints `imp` and `exp` which are the output parameters of the `NewChannel` method (lines 6–8), computes a new disk name (line 9), and tries to register the chosen name along with `imp` to `DirectoryService` through `ns` (line 10). The `switch receive` construct (lines 11–24) is used to receive messages and to dispatch control to various cases depending on the type of message that is received. Each `case` block specifies the endpoint from which a message is expected and the tag of the message. In this example, one of two kinds of messages are expected from the `ns` endpoint: either an `AckRegister` message (lines 12–15) or a `NakRegister` message (lines 16–23). In the first case the registration is successful (line 12), so the output parameter `expService` is properly initialized and the function terminates correctly (line 15). In the second case the registration is unsuccessful (line 16), hence a new registration is attempted if the error is recoverable (lines 17–18), otherwise an exception is thrown to abort the execution of the function (line 20). The main loop (lines 5–25) is protected within a `try` block with a `finally` clause that is executed regardless of whether the function terminates correctly or not. In the example, the clause deallocates the `ns` endpoint (line 27).

Sing# uses *channel contracts* to detect communication errors. Figure 2 shows (part of) the `DSContract` contract associated with endpoint `ns` in Figure 1. A contract is made of *message specifications* and of *states* connected by *transitions*. Each message specification begins with the keyword `message` and is followed by the *tag* of the message and the type of its arguments. In Figure 2, `DSContract` defines the `Register` message with two arguments (a string and another endpoint)

[1] This function has been taken from `./Services/RamDisk/ClientManager/RamDiskClientManager.sg` in the Singularity OS source code available at `http://www.codeplex.com/singularity/`. Here we have shortened some identifiers to fit the available space.

```
1  void GetNextDiskPath(out string! diskName,
2                       out SPContract.Exp! expService) {
3    DSContract.Imp:Ready ns = DS.NewClientEndpoint();
4    try {
5      while (true) {
6        SPContract.Imp! imp;
7        SPContract.Exp! exp;
8        SPContract.NewChannel(out imp, out exp);
9        diskName = pathPrefix + nextDiskNumber.ToString();
10       ns.SendRegister(Bitter.FromString2(diskName), imp);
11       switch receive {
12         case ns.AckRegister():
13           nextDiskNumber++;
14           expService = exp;
15           return;
16         case ns.NakRegister(nakImp, error):
17           if (error == ErrorCode.AlreadyExists)
18             nextDiskNumber++;
19           else
20             throw new RamDiskErrorException(error);
21           delete exp;
22           delete nakImp;
23           break;
24       }
25     }
26   } finally {
27     delete ns;
28   }
29 }
```

Figure 1: Example of Sing# function.

```
contract DSContract {
  out message Success();
  in  message Register(char[]! in ExHeap path,
                       SPContract.Imp:Start! imp);
  out message AckRegister();
  out message NakRegister(SPContract.Imp:Start imp,
                          ErrorCode error);

  // ...more message types

  state Start : one { Success!  →  Ready; }

  state Ready : one {
    Register?  →  DoRegister;
    CreateDirectory?  →  ...
    // ...more transitions
  }

  state DoRegister : one {
    AckRegister!  →  Ready;
    NakRegister!  →  Ready;
  }
}
```

Figure 2: Example of Sing# contract.

and the `AckRegister` message with no arguments. The `in` and `out` qualifiers specify the direction of messages from the point of view of the process exporting the contract. The state of the contract gives information about which messages can be sent/received at every given point in time. In `DSContract` we have a `Ready` state from which `Register`, `CreateDirectory`, and other (here omitted) messages can be received. After receiving a `Register` message, the contract moves to state `DoRegister`, from which one of the `AckRegister` or `NakRegister` messages can be sent, and then the contract goes back to the `Ready` state. In fact, each contract has two complementary views – called *exporting* and *importing* views – which are associated with the two peer endpoints of the channel. By convention, a contract declaration like that in Figure 2 specifies the exporting view of the contract: a provider of `DSContract` must adhere to its exporting view. In contrast,

the function GetNextDiskPath in Figure 1 acts as a consumer of DSContract, therefore the function performs complementary actions by sending a Register message and then waiting for either an AckRegister or a NakRegister message. In the code, the importing and exporting views correspond to the types obtained by appending .Imp and .Exp suffixes to the name of the contract. For example, the declaration on line 3 specifies that ns is an endpoint having as type the importing view of DSContract in state Ready. After line 10, the type associated with ns changes to DSContract.Imp:DoRegister and then it goes back to DSContract.Imp:Ready after any of the receive operations on lines 12 and 16. Note that the changes in the state of the contract associated with ns (and therefore of the type of ns) are not explicit in the source code. They follow from the initial declaration that brings ns into scope (line 3) and from the way ns is used in the function. By keeping track of the contract state of ns, the compiler can statically check that the actions performed on ns (for sending and receiving messages) match corresponding co-actions (for receiving and sending) performed on its peer endpoint, which is in use by some other process in the system.

The code structure in Figure 1, involving channel allocation and deallocation, messaging, delegation (sending endpoints over other endpoints), and exception handling, is in fact typical throughout the whole Singularity OS and shows that these aspects are frequently mixed in non-trivial ways. We can identify two main problems caused by exceptions:

1. Since communication errors are prevented by the complementarity of actions performed by processes accessing peer endpoints, a jump in the control flow of one process, like that caused by an exception, may disrupt the alignment of the peers of a channel and compromise subsequent interactions.

2. When an exception is thrown, messages that have been sent but not yet received and other objects allocated since the beginning of a try may become unreachable and therefore turn into memory leaks.

Sing# has limited and not fully satisfactory mechanisms for dealing with these problems. Regarding the first, Sing# provides an InState method through which it is possible to query, at runtime, the actual state of an endpoint. This information can be used to attempt recovery from a possibly inconsistent state of the endpoints. This mechanism implies an overhead for preserving and maintaining typing information at runtime and is unreliable as it depends on the programmer. The second problem seems to have been neglected. For example, the function

in Figure 1 is prone to leak memory on line 20 in the case that the exception is thrown, since neither `exp` nor `nakImp` are properly deallocated (`imp` has been sent away in the call to `SendRegister` so it is not the current thread's responsibility to deallocate it). In this example it would suffice to move the `delete` instructions on lines 21 and 22 between lines 16 and 17 but, in general, it may be impossible to identify the exact point where an exception can be thrown and therefore when it is appropriate to deallocate resources. Note that it is unreasonable to assume that this clean-up code will be placed in the exception handler, if only because the handler may not be in the scope of the resources to be deallocated: in the example, `exp` and `nakImp` are not visible in the `finally` block so, by the time the exception has been thrown, it is too late to prevent the leak.

In the present paper we put forward a solution that combines static analysis (inspired by existing works on exception handling for sessions by Carbone et al. [11], Capecchi et al. [12]) with a transaction-like, all-or-nothing semantics of `try` blocks. The basic idea is that a `try` block is either executed completely, and then its effects on the heap are committed and become permanent, or it is aborted by an exception. If this happens, *all* the processes involved in the transaction are notified of the exception, so that the types of the endpoints they are using can remain aligned, and the state of the heap is restored to that at the beginning of the `try` block. Our solution relies on the following key ideas:

(A) Following Carbone et al. [11], Capecchi et al. [12], we add explicit annotations to the types of endpoints used inside a transaction so that all processes involved in the transaction are aware of all the exceptions that can be thrown (possibly by a different process) during the transaction. In addition, these annotations make sure that the queues of endpoints used in a transaction are *empty* at the beginning of the transaction so that heap restoration solely amounts to removing messages from queues.

(B) Inside `try` blocks, we "seal" the type of any endpoint whose type is not properly annotated and we forbid processes to use endpoints with a sealed type. In this way, the type system can statically ensure that well-typed processes do not modify any portion of the heap outside the restorable one.

(C) We forbid the deallocation of endpoints inside `try` blocks, unless they have been allocated within the very same block. In this way, state restoration does not involve reallocations, which are difficult to implement correctly.

To prevent memory leaks, it is necessary to dynamically keep track of the

P	$::=$		**Process**
		$\texttt{done}$	(inaction)
	$\mid$	$\texttt{open}(a,a).P$	(open channel)
	$\mid$	$\texttt{close}(u).P$	(close endpoint)
	$\mid$	$u!\mathtt{m}(u).P$	(send)
	$\mid$	$\sum_{i\in I} u?\mathtt{m}_i(x_i).P_i$	(receive)
	$\mid$	$P \oplus P$	(conditional)
	$\mid$	$P \mid P$	(parallel)
	$\mid$	$\texttt{try}(U)\ \{\mathsf{e}_i : P_i\}_{i\in I} P$	(initiate transaction)
	$\mid$	$\texttt{throw}\ \mathsf{e}$	(exception)
	$\mid$	$\texttt{commit}(U).P$	(commit transaction)
	$\mid$	$X\langle\tilde{u}\rangle$	(invocation)
D	$::=$		**Definition**
		$X(\tilde{u}) \stackrel{\text{def}}{=} P$	(rule)

Table 1: Syntax of processes and definitions.

memory allocated within a `try` block so that this memory can be properly reclaimed in case an exception is thrown. It is unsafe to deallocate an endpoint if its peer is not deallocated simultaneously: mechanism (A) guarantees that these deallocations are safe even if the type of these endpoints would not normally allow it, because transactions define a "closed scope" that includes, for each endpoint used in a transaction, also its peer. Starting with the next section we turn into the technical part of this work, in which we make precise all of the concepts informally introduced so far.

3. Language

Notation. We assume that we are given an infinite set Pointers of *heap addresses* ranged over by a, b, ..., an infinite set Variables of *variables* ranged over by x, y, ..., and a set Exceptions of *exceptions* ranged over by e, ... We let *names* u, v, ... range over elements of Pointers $\cup$ Variables. We use A, B, ... to denote sets of pointers, $\mathscr{E}$, ... to denote sets of exceptions, U to denote sets of names, and $\tilde{u}$, $\tilde{v}$ to denote sequences of names (we will sometimes use $\tilde{u}$ to denote also the set of names in $\tilde{u}$). Process variables are ranged over by X, Y, ...

Syntax. The process language is essentially a variant of the π-calculus [13], except that names represent heap pointers instead of communicating channels. *Processes* are defined by the grammar in Table 1. The term $\texttt{done}$ denotes the idle process that performs no action. The term $\texttt{open}(a,b).P$ denotes a process that allocates a new channel, represented as the two peer endpoints a and b, in the heap and continues as P. The term $u!\mathtt{m}(v).P$ denotes a process that sends the message $\mathtt{m}(v)$ on the endpoint u and then continues as P. A *message* is made of a *tag* $\mathtt{m}$ and an argument v. The term $\sum_{i\in I} u?\mathtt{m}_i(x_i).P_i$ denotes a process that waits for a message from endpoint u. According to the tag $\mathtt{m}_i$ of the received message, the variable x_i is instantiated with the argument of the message in the continuation process P_i. We assume that the set I is always finite and non-empty. The term $P \oplus Q$ denotes a process that nondeterministically decides to behave as either P or Q, while the term $P \mid Q$ denotes the standard parallel composition of P and Q. The term $\texttt{try}(U)\ \{\mathtt{e}_i : Q_i\}_{i\in I} P$ denotes a process willing to initiate a transaction involving the endpoints in U. The process P is the *body* of the transaction and is executed when the transaction is initiated, while the Q_i's are the *handlers* of the transaction which are activated if the transaction is aborted during the execution of the body by an exception $\mathtt{e}_i$. The term $\texttt{throw}\ \mathtt{e}$ denotes the throwing of the exception $\mathtt{e}$, whose effect is to abort the currently running transaction and to execute the appropriate handler. The term $\texttt{commit}(U).P$ denotes a process willing to terminate the currently running transaction (involving the endpoints in U). As soon as the transaction has ended, the process continues as P. The term $X\langle\tilde{u}\rangle$ denotes the invocation of the process associated with the process variable X. We assume that we work with a global environment of process definitions of the form

$$X(\tilde{u}) \stackrel{\text{def}}{=} P$$

defining these associations.

The binders of the language are $\texttt{open}(a,b).P$, which binds a and b in P, the input prefix $u?\mathtt{m}(x).P$, which binds x in P, and $X(\tilde{u}) \stackrel{\text{def}}{=} P$ which binds the names $\tilde{u}$ in P. The formal definitions of free and bound names of a process P, respectively denoted by $\mathsf{fn}(P)$ and $\mathsf{bn}(P)$, are standard. We identify processes modulo alpha renaming of bound names.

Syntactic conventions. We adopt some standard conventions regarding the syntax of processes: we sometimes use an infix form for receive operations and write, for example $u?\mathtt{m}_1(x_1).P_1 + \cdots + u?\mathtt{m}_n(x_n).P_n$ instead of $\sum_{i=1..n} u?\mathtt{m}_i(x_i).P_i$; we omit message arguments when they are not used; we sometimes use a prefix form for

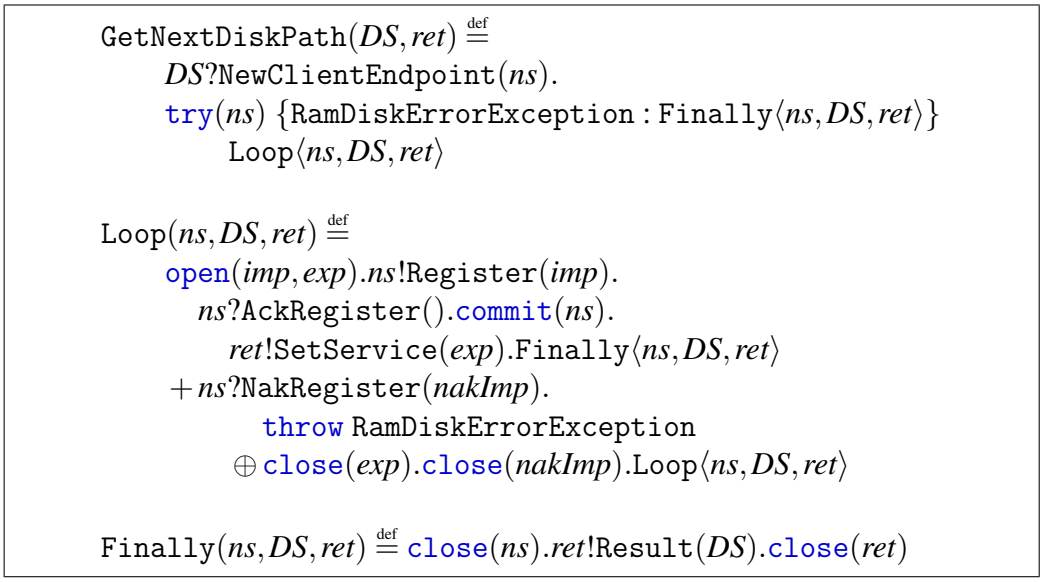

Figure 3: Encoding of the function in Figure 1.

parallel compositions and write, for example, $\prod_{i=1..n} P_i$ instead of $P_1 \mid \cdots \mid P_n$; we identify `done` with $\prod_{i \in \emptyset} P_i$ and we omit trailing occurrences of `done`.

To ease the formalization, our process language supports a minimal set of critical features: we focus only on monadic messaging (messages have exactly one endpoint argument) and exception handling, disregarding other constructs and data types of Sing#; we assume that receive operations use the same endpoint in every branch, forbidding processes like $u?\mathtt{a}(x).P + v?\mathtt{b}(y).Q$ which are allowed by the `switch receive` construct in Sing#; we work with a purely prefix-based language without sequential composition, encoding `try-catch-finally` blocks in Sing# with transaction bodies and handlers and `commit` processes within bodies; we encode `if-else` commands with the non-deterministic process $P \oplus Q$ omitting the condition that determines the chosen branch. We claim that all the results presented hereafter can be suitably extended to overcome these restrictions.

Example 3.1. Figure 3 shows the encoding of the function in Figure 1 using the syntax of our process language. The structure of the process follows quite closely that of the function, except for some details which we explain here.

The loop on lines 5–25 is encoded as a recursive process `Loop` parameterized on its free names. The `finally` block on lines 26–28 is factored out as a named process `Finally`, since it must be executed regardless of whether the `try` block

is terminated successfully (line 15) or not (line 20). Consequently, `Finally` is invoked twice in the encoding.

The main difference between the function Figure 1 and its encoding concerns parameter passing, which is encoded using explicit communication on the *ret* endpoint. In particular, the initialization of `expService` with `exp` on line 14 corresponds to the output operation *ret*!`SetService`(*exp*) in Figure 3.

Note that in Figure 1 the function uses a global name `DS` for accessing a system service. In order to obtain a closed term, in the encoding we explicitly mention a parameter *DS* of the `GetNextDiskPath` process which represents `DS`. Because our type system relies on the linear access to resources, invoking a parametric process such as `GetNextDiskPath` means transferring the ownership of the parameters to the process. To preserve linearity (of *DS* in this case), the `Finally` process sends *DS* back on *ret* before *ret* is closed (more involved examples of function modeling and ownership transfer are described in detail by Bono and Padovani [7]). ■

Operational semantics. In order to describe the operational semantics of processes, we need to represent the *heap* where channels are allocated and through which messages are exchanged. Indeed, channels are accessed through the pointers to their endpoints and message arguments are themselves pointers to heap objects. Intuitively, a heap μ is a finite map from pointers a to endpoint structures $[b, \mathfrak{Q}]$, where b is the *peer endpoint* of a and $\mathfrak{Q}$ is the queue of messages waiting to be received from a. In the model, we represent heaps and message queues as terms generated by the grammar in Table 2. The term $\emptyset$ denotes the empty heap, in which no endpoints are allocated. The term $a \mapsto [b, \mathfrak{Q}]$ denotes an endpoint allocated at a pointing to the endpoint structure $[b, \mathfrak{Q}]$. The term μ, μ' denotes the composition of the heaps μ and μ'. We write $\mathsf{dom}(\mu)$ for the *domain* of the heap μ, that is the set of pointers for which there is an allocated endpoint structure. The heap composition μ, μ' is well defined provided that $\mathsf{dom}(\mu) \cap \mathsf{dom}(\mu') = \emptyset$ (there cannot be two endpoint structures allocated at the same address). In the following, we identify queues assuming associativity of composition and the laws $\varepsilon :: \mathfrak{Q} = \mathfrak{Q} :: \varepsilon = \mathfrak{Q}$ and we identify heaps assuming associativity and commutativity of composition and the law $\emptyset, \mu = \mu$. We write $a \mapsto [b, \mathfrak{Q}] \in \mu$ to indicate that the endpoint structure $[b, \mathfrak{Q}]$ is allocated at location a in μ.

Message queues, ranged over by $\mathfrak{Q}$, are also represented as terms: ε denotes the empty queue, $\mathtt{m}(c)$ is a queue made of an `m` message with argument c, and $\mathfrak{Q} :: \mathfrak{Q}'$ is the queue composition of $\mathfrak{Q}$ and $\mathfrak{Q}'$. We identify queues modulo associativity of :: and we assume that ε is neutral for ::.

Before defining the operational semantics of processes we formalize two no-

μ	$::=$		**Heap**
		$\emptyset$	(empty heap)
	$\mid$	$a \mapsto [a, \mathfrak{Q}]$	(endpoint structure)
	$\mid$	μ, μ	(heap composition)
$\mathfrak{Q}$	$::=$		**Queue**
		ε	(empty queue)
	$\mid$	$\mathtt{m}(a)$	(message)
	$\mid$	$\mathfrak{Q} :: \mathfrak{Q}$	(queue composition)
P	$::=$		**Runtime process**
		$\cdots$	(as in Table 1)
	$\mid$	$\langle A, B, \{\mathsf{e}_i : P_i\}_{i \in I} P \rangle$	(running transaction)

Table 2: Syntax of heaps, queues, and runtime processes.

tions. The first one is that of peer endpoints:

Definition 3.1 (peer endpoints)**.** We say that a and b are *peer endpoints* in μ, written $a \overset{\mu}{\leftrightarrow} b$, if $a \neq b$ and $a \mapsto [b, \mathfrak{Q}] \in \mu$ and $b \mapsto [a, \mathfrak{Q}'] \in \mu$.

Note that $\overset{\mu}{\leftrightarrow}$ is a symmetric relation. The notion of "closed scope" that we mentioned in Section 2 is formalized as a predicate on sets of pointers:

Definition 3.2 (balanced set of pointers)**.** We say that $A \subseteq \mathsf{dom}(\mu)$ is *balanced* in μ, written μ-$\mathsf{balanced}(A)$, if, for every $a \in A$, $a \overset{\mu}{\leftrightarrow} b$ implies $b \in A$.

In words, A is balanced in μ if for every a in A, the peer of a is also in A provided that it is still allocated in μ. Since a message sent over a ends up in the queue of its peer, this means that any communication occurring on one of the endpoints in A remains within the scope identified by A.

In the operational semantics of processes, we need to distinguish between a transaction that has not started yet (and which is represented using the `try` construct of Table 1), and a *running transaction*. This need arises for two reasons: First, a running transaction generally involves more than one process, each with its own set of handlers. Therefore, it is technically convenient to devise an explicit construct that defines the *scope* of the transaction. Second, it is necessary to keep track of the part of the heap that has been allocated since the initiation of the transaction. Table 2 extends the syntax of processes with the term $\langle A, B, \{\mathsf{e}_i : P_i\}_{i \in I} P \rangle$

Structural congruence

[S-PAR IDLE] $P \mid \texttt{done} \equiv P$

[S-PAR COMM] $P \mid Q \equiv Q \mid P$

[S-PAR ASSOC] $P \mid (Q \mid R) \equiv (P \mid Q) \mid R$

Reduction relation

[R-OPEN] $\mu \fatsemi \texttt{open}(a,b).P \to \mu, a \mapsto [b,\varepsilon], b \mapsto [a,\varepsilon] \fatsemi P$

[R-PARALLEL] $\dfrac{\mu \fatsemi P \to \mu' \fatsemi P'}{\mu \fatsemi P \mid Q \to \mu' \fatsemi P' \mid Q}$

[R-CLOSE] $\mu, a \mapsto [b,\mathfrak{Q}] \fatsemi \texttt{close}(a).P \to \mu \fatsemi P$

[R-CHOICE] $\dfrac{i \in \{1,2\}}{\mu \fatsemi P_1 \oplus P_2 \to \mu \fatsemi P_i}$

[R-SEND] $\mu, a \mapsto [b,\mathfrak{Q}], b \mapsto [a,\mathfrak{Q}'] \fatsemi a!\mathtt{m}(c).P \to \mu, a \mapsto [b,\mathfrak{Q}], b \mapsto [a,\mathfrak{Q}' :: \mathtt{m}(c)] \fatsemi P$

[R-RECEIVE] $\dfrac{k \in I}{\mu, a \mapsto [b,\mathtt{m}_k(c) :: \mathfrak{Q}] \fatsemi \sum_{i \in I} a?\mathtt{m}_i(x_i).P_i \to \mu, a \mapsto [b,\mathfrak{Q}] \fatsemi P_k\{c/x_k\}}$

[R-INVOKE] $\dfrac{X(\tilde{u}) \stackrel{\text{def}}{=} P}{\mu \fatsemi X\langle \tilde{a} \rangle \to \mu \fatsemi P\{\tilde{a}/\tilde{u}\}}$

[R-STRUCT] $\dfrac{P \equiv P' \qquad \mu \fatsemi P' \to \mu' \fatsemi Q' \qquad Q' \equiv Q}{\mu \fatsemi P \to \mu' \fatsemi Q}$

Table 3: Operational semantics of processes.

where A is the set of endpoints involved in the transaction, B is the set of endpoints that have been allocated since the transaction has started, P is the (residual) body of the transaction, and the P_i's represent the handlers of the transaction. In general, P and the P_i's will be parallel compositions of the bodies and the handlers of the processes that have cooperatively initiated the transaction.

The operational semantics of processes is defined in terms of a structural congruence over processes (identifying structurally equivalent processes) and a reduction relation. Structural congruence is the least relation $\equiv$ including alpha conversion and the laws in Table 3, stating that parallel composition is commutative, associative, and has `done` as neutral element. As process interaction mostly

occurs through the heap, the reduction relation describes the evolution of *configurations* $\mu \fatsemi P$ rather than of processes alone, so that

$$\mu \fatsemi P \to \mu' \fatsemi P'$$

denotes the fact that process P evolves to P' and, in doing so, it changes the heap from μ to μ'.

Reduction is the smallest relation between configurations defined by the rules in Tables 3 and 4. We explain the rules in the following paragraphs. Rule [R-OPEN] describes the creation of a new channel, which causes the allocation of two new endpoint structures in the heap. The endpoints are initialized with empty queues and are allocated at fresh locations, for otherwise the resulting heap would be ill formed. Since we have assumed that Pointers is infinite, it is always possible to alpha rename a and b to fresh pointers using structural congruence, so that an $\texttt{open}(a,b).P$ is always able to reduce.

Rule [R-CLOSE] describes the closing of an endpoint, which deallocates its structure from the heap and discards its queue. Note that both endpoints of a channel are created simultaneously by [R-OPEN], but each is closed independently by [R-CLOSE] (this is the same semantics as the one of $\mathsf{Sing}^{\#}$).

Rule [R-CHOICE] states that a process $P \oplus Q$ nondeterministically reduces to either P or Q.

Rule [R-SEND] describes the sending of a message $\mathtt{m}(c)$ on the endpoint a. The message is enqueued at the right end of the queue associated with the peer endpoint b of a. Note that, for this rule to be applicable, it is necessary for both endpoints of a channel to still be allocated.

Rule [R-RECEIVE] describes the receiving of a message from endpoint a. In particular, the message at the left end of the queue associated with a is removed from the queue, its tag $\mathtt{m}_k$ is used to select one branch of the process, and its argument c instantiates the corresponding variable x_k.

Rule [R-PARALLEL] describes the independent evolution of parallel processes. Note how the heap is treated globally even if it is only one subprocess to reduce.

Rule [R-INVOKE] describes process invocations simply as the replacement of a process variable with the process it is associated with, modulo the substitution of its parameters. In this rule and in [R-RECEIVE], $P\{\tilde{u}/\tilde{v}\}$ denotes the capture-avoiding substitution of $\tilde{u}$ in place of $\tilde{v}$ in P.

Rule [R-START TRANSACTION] describes the initiation of a transaction by a number of processes. The transaction is identified by a set of endpoints $\bigcup_{i\in I} A_i$ which are distributed among the processes. In order for the transaction to start, this

[R-START TRANSACTION]

$$\frac{\mu\text{-balanced}(\bigcup_{i\in I} A_i)}{\mu \fatsemi \prod_{i\in I} \texttt{try}(A_i)\ \{\mathsf{e}_j : Q_{ij}\}_{j\in J} P_i \to \mu \fatsemi \langle \bigcup_{i\in I} A_i, \emptyset, \{\mathsf{e}_j : \prod_{i\in I} Q_{ij}\}_{j\in J} \prod_{i\in I} P_i\rangle}$$

[R-END TRANSACTION]

$$\mu \fatsemi \langle A, B, \{\mathsf{e}_j : Q_j\}_{j\in J} \prod_{i\in I} \texttt{commit}(A_i).P_i\rangle \to \mu \fatsemi \prod_{i\in I} P_i$$

[R-RUN TRANSACTION]

$$\frac{\mu \fatsemi P \to \mu' \fatsemi P'}{\mu \fatsemi \langle A, B, \{\mathsf{e}_j : Q_j\}_{j\in J} P\rangle \to \mu' \fatsemi \langle A, \mathsf{track}(B, \mathsf{dom}(\mu), \mathsf{dom}(\mu')), \{\mathsf{e}_j : Q_j\}_{j\in J} P'\rangle}$$

[R-CATCH EXCEPTION]

$$\frac{k \in J}{\mu_1, \{a_i \mapsto [b_i, \mathfrak{Q}_i]\}_{i\in I}, \mu_2 \fatsemi \langle \{a_i\}_{i\in I}, \mathsf{dom}(\mu_2), \{\mathsf{e}_j : Q_j\}_{j\in J} \texttt{throw}\ \mathsf{e}_k \mid P\rangle \to \mu_1, \{a_i \mapsto [b_i, \varepsilon]\}_{i\in I} \fatsemi Q_k}$$

[R-PROPAGATE EXCEPTION]

$$\frac{\forall j \in J : \mathsf{e}_j \neq \mathsf{e}}{\mu_1, \{a_i \mapsto [b_i, \mathfrak{Q}_i]\}_{i\in I}, \mu_2 \fatsemi \langle \{a_i\}_{i\in I}, \mathsf{dom}(\mu_2), \{\mathsf{e}_j : Q_j\}_{j\in J} \texttt{throw}\ \mathsf{e} \mid P\rangle \to \mu_1, \{a_i \mapsto [b_i, \varepsilon]\}_{i\in I} \fatsemi \texttt{throw}\ \mathsf{e}}$$

Table 4: Operational semantics of transactions.

set of endpoints must be balanced, so that for every endpoint in the set its peer is also in the set. The rule is nondeterministic, in the sense that there can be multiple combinations of processes that can initiate a transaction. We leave the choice of a particular strategy (for example, requiring $\bigcup_{i\in I} A_i$ to be non-empty, minimal, and μ-balanced) to the implementation. The residual process is the tuple

$$\langle \bigcup_{i\in I} A_i, \emptyset, \{\mathsf{e}_j : \prod_{i\in I} Q_{ij}\}_{j\in J} \prod_{i\in I} P_i\rangle$$

combining the bodies and the handlers of the processes involved in the transaction. The second component is $\emptyset$ indicating that at this stage no new endpoints have been allocated yet within the transaction. Note that the combined processes must be able to handle the same set $\{\mathsf{e}_j\}_{j\in J}$ of exceptions for the reduction to occur. Even if this seems to require a runtime check, the fact that all processes involved in a transaction are able to handle the same set of exceptions will be ensured by the type system (see rule [T-TRY] in Table 8).

Rule [R-END TRANSACTION] reduces a running transaction to its continuation when its body has terminated. The handlers are discarded. The sets A_i play no role in the operational semantics and are used for typing purposes only. In fact, we will see that the type system enforces the invariant that these sets coincide with the ones decorating the `try` blocks corresponding to the `commit` processes and, in particular, $A = \bigcup_{i \in I} A_i$.

Rule [R-RUN TRANSACTION] allows the reduction of a transaction according to the reductions of its body. The rule keeps track of the memory changes occurred during the reduction of the body of the transaction by updating the second component of the transaction to $\mathsf{track}(B, \mathsf{dom}(\mu), \mathsf{dom}(\mu'))$, where

$$\mathsf{track}(B, A_0, A_1) \stackrel{\text{def}}{=} (B \cup (A_1 \setminus A_0)) \setminus (A_0 \setminus A_1)$$

In practice, the pointers to objects allocated during the reduction are added to B, while the pointers to objects deallocated during the reduction are removed from B.

Rule [R-CATCH EXCEPTION] describes the abnormal termination of a running transaction when an exception is thrown and the transaction provides an handler for it. In this case, the queues of all the endpoints involved in the transactions are emptied, the memory allocated within the transaction is reclaimed, and the appropriate handler is run. In a similar way, rule [R-PROPAGATE EXCEPTION] abnormally terminates running transactions when there is no suitable handler for the thrown exception. Also in this case the queues of the endpoints involved in the transactions are emptied and the memory allocated within the transaction is reclaimed, but the exception is propagated (technically, *re-thrown*) at the outer level.

We write $\mu \fatsemi P \rightarrow$ if $\mu \fatsemi P \rightarrow \mu' \fatsemi P'$ for some μ' and P' and $\mu \fatsemi P \not\rightarrow$ if not $\mu \fatsemi P \rightarrow$; we write $\Rightarrow$ for the reflexive, transitive closure of $\rightarrow$.

Well-behaved processes. We conclude this section providing a characterization of *well-behaved processes*, those that are free from memory leaks, memory faults, and communication errors. A *memory leak* occurs when no pointer to an allocated region of the heap is retained by any process. In this case, the allocated region has no owner, it occupies space, but it is no longer accessible. A *memory fault* occurs when a pointer is accessed and the endpoint it points to is not (or no longer) allocated. A *communication error* occurs when some process receives a message of unexpected type. To formalize well-behaved processes, we need to define the reachability of a heap object with respect to a set of *root* pointers. Intuitively, a process P may directly reach any object located at some pointer in the set $\mathsf{fn}(P)$

$$\text{[ST-INACTIVE]}\quad \mu \mathbin{;} \texttt{done} \downarrow \qquad \text{[ST-INPUT]}\quad \mu, a \mapsto [b, \varepsilon] \mathbin{;} \textstyle\sum_{i \in I} a?\mathtt{m}_i(x_i).P_i \downarrow \qquad \text{[ST-COMMIT]}\quad \mu \mathbin{;} \texttt{commit}(A).P \downarrow$$

$$\text{[ST-TRY]}\quad \frac{\neg \mu\text{-balanced}(A)}{\mu \mathbin{;} \texttt{try}(A)\ \{\mathsf{e}_j : Q_j\}_{j \in J} P \downarrow} \qquad \text{[ST-PARALLEL]}\quad \frac{\mu \mathbin{;} P \downarrow \qquad \mu \mathbin{;} Q \downarrow}{\mu \mathbin{;} P \mid Q \downarrow}$$

$$\text{[ST-RUNNING TRANSACTION]}\quad \frac{\mu \mathbin{;} P \downarrow \qquad P \not\equiv \prod_{i \in I} \texttt{commit}(A_i).P_i}{\mu \mathbin{;} \langle A, B, \{\mathsf{e}_j : Q_j\}_{j \in J} P \rangle \downarrow}$$

Table 5: Stuck configurations.

(we can think of the pointers in $\mathsf{fn}(P)$ as of the local variables of the process stored on its stack); from these pointers, the process may reach other heap objects by reading messages from the endpoints it can reach, and so forth.

Definition 3.3 (reachable pointers)**.** We say that c is *reachable* from a in μ, notation $c \prec_\mu a$, if $a \mapsto [b, \mathfrak{Q} :: \mathtt{m}(c) :: \mathfrak{Q}'] \in \mu$. We write $\preccurlyeq_\mu$ for the reflexive, transitive closure of $\prec_\mu$ and we define $\mu\text{-reach}(A) = \{c \in \mathsf{Pointers} \mid \exists a \in A : c \preccurlyeq_\mu a\}$.

The last auxiliary notion we need provides a syntactic characterization of those configurations that cannot reduce but that do not represent any of the errors described above.

Definition 3.4 (stuck configuration)**.** We say that the configuration $\mu \mathbin{;} P$ is *stuck* if the judgment $\mu \mathbin{;} P \downarrow$ is inductively derivable by the rules in Table 5.

Rules [ST-INACTIVE] and [ST-PARALLEL] are obvious, while rules [ST-TRY] and [ST-COMMIT] state that transaction initiations and termination are stuck, if taken in isolation. In the former case, the set of involved endpoints must not be balanced, for otherwise the transaction could initiate. Rule [ST-RUNNING TRANSACTION] states that a running transaction is stuck if its body is stuck and different from a combination of processes willing to terminate the transaction, for otherwise the transaction could terminate. Note also that no exception can have been thrown within the body of a stuck running transaction, for the stuckness predicate is undefined for `throw` e processes. Finally, rule [ST-INPUT] states that a process waiting for a message from endpoint a is stuck only if the endpoint a is allocated and its queue is empty. Then, a configuration whose processes are all waiting for a

message corresponds to a genuine deadlock. From these rules we deduce that a process willing to send a message on a is never stuck, and so is a process willing to receive a message from a if the queue associated with a is not empty.

Definition 3.5 (well-behaved process)**.** We say that P is *well behaved* if $\emptyset \fatsemi P \Rightarrow \mu \fatsemi Q$ implies:

1. $\mathsf{dom}(\mu) \subseteq \mu\text{-}\mathsf{reach}(\mathsf{fn}(Q))$;
2. $Q \equiv Q_1 \mid Q_2$ and $\mu \fatsemi Q_1 \not\to$ imply $\mu \fatsemi Q_1 \downarrow$.

In words, a process P is well behaved if every residual Q of P is such that Q can reach every pointer in the heap and every subprocess Q_1 of Q that does not reduce is stuck (recall that the definition of $\mu \fatsemi Q_1 \downarrow$ captures also the possibility that Q_1 is in deadlock). Here are a few examples of ill-behaved processes to illustrate the sort of errors we want to spot with our type system:

- The process $\mathtt{open}(a,b).\mathtt{done}$ violates condition (1), since it leaks endpoints a and b.
- The process $\mathtt{open}(a,b).(\mathtt{close}(a).\mathtt{close}(a) \mid \mathtt{close}(b))$ tries to deallocate the same endpoint a twice. This is an example of fault.
- The process $\mathtt{open}(a,b).(a!\mathtt{a}().\mathtt{close}(a) \mid b?\mathtt{b}().\mathtt{close}(b))$ violates condition (2) since it reduces to a parallel composition of subprocesses where one has sent an `a` message, but the other one was expecting a `b` message.
- The process

$$\mathtt{open}(a,b).\mathtt{try}(\emptyset)\ \{\mathtt{e} : \mathtt{done}\}\\ \qquad \mathtt{throw}\ \mathtt{e} \oplus \mathtt{commit}(\emptyset).\mathtt{close}(a).\mathtt{close}(b)$$

  may leak a and b if the exception is thrown.

Observe that, in item (1) of Definition 3.5, the domain of μ is only required to be *included in* (instead of being *equal to*) the set of pointers reachable from the free names of Q. In particular, it may be the case that Q contains references to unallocated objects, and yet it never attempts to use them. This formulation of leak-freedom, which is slightly more general than the one used by Jakšić and Padovani [10] where equality between the two sets was required, is necessary because the type system that we are about to define allows subtyping, which was

t	$::=$		**Type**
		T	(endpoint type)
	$\mid$	$[t]$	(sealed type)
T	$::=$		**Endpoint type**
		$\mathtt{end}$	(termination)
	$\mid$	α	(type variable)
	$\mid$	$\{!\mathtt{m}_i(T_i).T_i\}_{i\in I}$	(internal choice)
	$\mid$	$\{?\mathtt{m}_i(T_i).T_i\}_{i\in I}$	(external choice)
	$\mid$	$\{\mathtt{e}_i : T_i\}_{i\in I}[\![T$	(initiate transaction)
	$\mid$	$]\!]T$	(commit transaction)
	$\mid$	$\mathtt{rec}\ \alpha : r.T$	(recursive type)
	$\mid$	$\{\mathtt{e}_i : T_i\}_{i\in I}T$	(running transaction)

Table 6: Syntax of types and endpoint types.

not considered in [10]. Also note that our notion of leak-freedom does not require a process to eventually deallocate the objects it owns, but only to guarantee the reachability of all the objects it owns. For example, the process $\mathtt{open}(a,b).X\langle a,b\rangle$ where $X(u,v) \stackrel{\text{def}}{=} X\langle u,v\rangle$, maintains the reachability of a and b without ever using them. This process is well behaved according to Definition 3.5 and is also well typed according to the type system that will be developed in Section 4.

4. Type System

We now develop a type system that enforces well-behavedness of processes: in Section 4.1 we introduce the syntax of the type language; in Section 4.2 we define a notion of *type weight* which is used for discriminating between safe and unsafe communications; Section 4.3 is devoted to extending classical subtyping for session types by Gay and Hole [14] so as to take transactions and exceptions into account; Sections 4.4, 4.5, and 4.6 define the actual typing rules; finally, Section 4.7 presents the soundness result.

4.1. Syntax of Types

We assume that we are given an infinite set of *type variables* ranged over by α; we use t, s, ... to range over types, and T, S, ... to range over endpoint types. The syntax of types and endpoint types is defined in Table 6. An endpoint

type describes the behavior of a process with respect to a particular endpoint: the process may send messages over the endpoint, receive messages from the endpoint, deallocate the endpoint, initiate and terminate transactions involving the endpoint. The endpoint type $\mathtt{end}$ denotes an endpoint that can only be deallocated. An internal choice $\{!\mathtt{m}_i(S_i).T_i\}_{i\in I}$ denotes an endpoint on which a process may send any message $\mathtt{m}_i$ for $i \in I$. The message has an argument of type S_i and, depending on the tag $\mathtt{m}_i$, the endpoint can be used thereafter according to T_i. In a dual manner, an external choice $\{?\mathtt{m}_i(S_i).T_i\}_{i\in I}$ denotes an endpoint from which a process must be ready to receive any message $\mathtt{m}_i$ for $i \in I$ and, depending on the tag $\mathtt{m}_i$ of the received message, the endpoint is to be used according to T_i. In endpoint types $\{!\mathtt{m}_i(S_i).T_i\}_{i\in I}$ and $\{?\mathtt{m}_i(S_i).T_i\}_{i\in I}$ we assume that $I \neq \emptyset$ and $\mathtt{m}_i = \mathtt{m}_j$ implies $i = j$ for every $i, j \in I$. That is, the tag $\mathtt{m}_i$ of the message that is sent or received identifies a unique continuation T_i. The endpoint type $\{\mathtt{e}_i : S_i\}_{i\in I}[\![T$ denotes an endpoint on which it is possible to initiate a transaction. The type T specifies how the endpoint is used within the body of the transaction, whereas each type S_i specifies how the endpoint is used if the transaction is aborted by the exception $\mathtt{e}_i$. The endpoint type $]\!]T$ denotes the termination of the transaction in which an endpoint with this type is involved. As soon as the transaction is properly terminated, the endpoint can be subsequently used according to T. Terms α and $\mathtt{rec}\ \alpha : r.T$ can be used to specify recursive behaviors, as usual. The annotation r associated with α represents the rank of α, which will be explained shortly. Finally, the endpoint type $\{\mathtt{e}_i : S_i\}_{i\in I}T$ is analogous to $\{\mathtt{e}_i : S_i\}_{i\in I}[\![T$, except that it specifies the type of an endpoint involved in a transaction which has already been initiated, but has not terminated yet. In fact, this type is needed for technical reasons only, and will be used in conjunction with running transaction processes $\langle A, B, \{\mathtt{e}_i : P_i\}_{i\in I}P\rangle$. In no case the programmer is supposed to deal with endpoint types of this form.

Clearly, not every endpoint type written according to the syntax in Table 6 makes sense. For example, it is possible to write terms such as $\{\mathtt{e} : \mathtt{end}\}[\![\mathtt{end}$ where a transaction is initiated but not terminated or terms where recursions do not respect the intended nesting of transactions, like in $\mathtt{rec}\ \alpha.\{\mathtt{e} : \mathtt{end}\}[\![\alpha$ or in $\{\mathtt{e} : \mathtt{end}\}[\![\mathtt{rec}\ \alpha.]\!]\alpha$. As far as our analysis is concerned, the syntax does not even prevent $\mathtt{end}$ subterms from occurring within transactions, which as we have argued in Section 2 is undesirable since endpoints involved in transactions should not be closed. For all these reasons we define a subset of *well-formed* endpoint types based on a notion of *rank*. Intuitively, the rank of a term T gives the number of transactions within which T may occur, with the proviso that $\mathtt{end}$ must have rank 0.

$$\text{[WF-End]}\quad \Theta \vdash \mathtt{end} : 0 \qquad \text{[WF-Var]}\quad \Theta, \alpha : r \vdash \alpha : r \qquad \text{[WF-Rec]}\quad \frac{\Theta, \alpha : r \vdash T : r}{\Theta \vdash \mathtt{rec}\ \alpha : r.T : r}$$

$$\text{[WF-Prefix]}\quad \frac{\dagger \in \{?, !\} \qquad \Theta \vdash S_i : 0\ {}^{(i \in I)} \qquad \Theta \vdash T_i : r\ {}^{(i \in I)}}{\Theta \vdash \{\dagger \mathtt{m}_i(S_i).T_i\}_{i \in I} : r} \qquad \text{[WF-Commit]}\quad \frac{\Theta \vdash T : r}{\Theta \vdash [\![T : r+1}$$

$$\text{[WF-Initiate]}\quad \frac{\Theta \vdash S_i : r\ {}^{(i \in I)} \qquad \Theta \vdash T : r+1}{\Theta \vdash \{\mathtt{e}_i : S_i\}_{i \in I} [\![T : r} \qquad \text{[WF-Run]}\quad \frac{\Theta \vdash S_i : r\ {}^{(i \in I)} \qquad \Theta \vdash T : r+1}{\Theta \vdash \{\mathtt{e}_i : S_i\}_{i \in I} T : r}$$

Table 7: Rank of endpoint types.

In general, we say that the endpoint type T is well formed and has rank r in Θ if $\Theta \vdash T : r$ is inductively derivable by the axioms and rules in Table 7, where Θ ranges over ranking contexts associating ranks to type variables. Then, a derivation of $\emptyset \vdash T : 0$ means that T is a closed endpoint type where transaction initiations and terminations are balanced. Rules [WF-Initiate], [WF-Run], and [WF-Commit] count the number of nested transactions. Rule [WF-Prefix] requires all branches of a choice to have the same rank, while rules [WF-Rec] and [WF-Var] deal with recursive types in a standard way, by respectively augmenting and accessing the ranking context. In the following we will omit Θ from judgments $\Theta \vdash T : r$ if Θ is empty.

As welcome side effects of well formedness, note that:

- message types have rank 0 (rule [WF-Prefix]). Thus, well-typed processes will not be able to send/receive endpoints involved in pending transactions;
- `end` cannot occur inside transactions (rule [WF-End]). Thus, well-typed processes will not be able to close endpoints involved in pending transactions.

The rank annotation r in recursive terms $\mathtt{rec}\ \alpha : r.T$ guarantees that every well-formed endpoint type has a uniquely determined rank. Without this annotation a term like $\mathtt{rec}\ \alpha.!\mathtt{m}(\mathtt{end}).\alpha$ could be given any rank. The following proposition guarantees that the rank of well-formed endpoint types is unaffected by folding/unfolding of recursions:

Proposition 4.1. *If* $\vdash \mathtt{rec}\ \alpha : r.T : r$, *then* $\vdash T\{\mathtt{rec}\ \alpha : r.T/\alpha\} : r$.

Proof. A simple induction on the derivation of $\Theta, \alpha : r \vdash T : r$. □

In what follows, we will assume that all endpoint types are closed and well formed and we will usually omit the rank annotation from recursive terms with the assumption that they can be properly annotated so that they are well formed; we will also write $\mathsf{rank}(T)$ for the rank of T. We will identify endpoint types modulo alpha renaming of bound type variables (the only binder being `rec`) and folding/unfolding of recursions knowing that this does not change their rank (Proposition 4.1). In particular, we have $\mathtt{rec}\ \alpha.T = T\{\mathtt{rec}\ \alpha.T/\alpha\}$. Finally, we will sometimes use an infix notation for internal and external choices and write $!\mathtt{m}_1(S_1).T_1 \oplus \cdots \oplus !\mathtt{m}_n(S_n).T_n$ instead of $\{!\mathtt{m}_i(S_i).T_i\}_{i\in\{1,\ldots,n\}}$ and $?\mathtt{m}_1(S_1).T_1 + \cdots + ?\mathtt{m}_n(S_n).T_n$ instead of $\{?\mathtt{m}_i(S_i).T_i\}_{i\in\{1,\ldots,n\}}$.

Types are possibly sealed endpoint types of the form

$$[\cdots[T]\cdots]$$

for some arbitrary (possibly zero) number of seals $[\cdots]$. Seals protect the endpoints not involved in a transaction: they are applied when the transaction is initiated (the `try` primitive is executed) and are stripped off when the transaction terminates (the `commit` primitive is executed). The type system prevents endpoints with a sealed type from being used, since any change to them would not be undoable in case the currently running transaction is aborted.

Example 4.1. According to the process definitions in Figure 3, the endpoint *ns* is involved in the transaction around the `Loop` process, it is used for sending a `Register` message and then for receiving either an `AckRegister` or a `NakRegister` message. The same endpoint is then closed regardless of whether the transaction completes successfully or not. We can describe the overall behavior of `GetNextDiskPath`, `Loop`, and `Finally` on *ns* with the following endpoint type:

$$T_{ns} = \{\mathtt{RamDiskErrorException} : \mathtt{end}\}[\![\mathtt{rec}\ \alpha.!\mathtt{Register}(T_{imp}).(?\mathtt{AckRegister}().[\![\mathtt{end} + ?\mathtt{NakRegister}(T_{imp}).\alpha)$$

where T_{imp} is the endpoint type associated with the *imp* and *nakImp* endpoints.

The endpoint *ret* is not used within the transaction, but its usage differs depending on whether or not the exception is thrown:

$$T_{ret} = !\mathtt{Result}(T_{DS}).\mathtt{end} \oplus !\mathtt{SetService}(T_{exp}).!\mathtt{Result}(T_{DS}).\mathtt{end}$$

If no exception is thrown, *ret* is used for sending a `SetRegister` message followed by a `Result` one; if an exception is thrown, only the `Result` message is sent. The above type T_{ret} takes into account both possibilities. ■

In order to avoid communication errors, we associate peer endpoints with endpoint types describing complementary actions: if a process sends a message of some kind on one endpoint, another process must be able to receive a message of that kind from the peer endpoint; if one process initiates a transaction involving one endpoint, the other process will do so as well on the peer endpoint; if one process has finished using an endpoint, the process owning the peer endpoint has finished too. We formalize this complementarity of actions by defining a function that, given an endpoint type, computes its dual:

Definition 4.1 (duality)**.** *Duality* is the function $\overline{\cdot}$ on endpoint types defined coinductively by the equations:

$$\begin{array}{rcl}
\overline{\mathtt{end}} & = & \mathtt{end} \\
\overline{\{?\mathtt{m}_i(S_i).T_i\}_{i\in I}} & = & \{!\mathtt{m}_i(S_i).\overline{T_i}\}_{i\in I} \\
\overline{\{!\mathtt{m}_i(S_i).T_i\}_{i\in I}} & = & \{?\mathtt{m}_i(S_i).\overline{T_i}\}_{i\in I} \\
\overline{\{\mathtt{e}_i : S_i\}_{i\in I}[\![T} & = & \{\mathtt{e}_i : \overline{S_i}\}_{i\in I}[\![\overline{T} \\
\overline{[\![T} & = & [\![\overline{T} \\
\overline{\{\mathtt{e}_i : S_i\}_{i\in I}T} & = & \{\mathtt{e}_i : \overline{S_i}\}_{i\in I}\overline{T}
\end{array}$$

Roughly speaking, the dual of an endpoint type T is obtained from T by swapping internal and external choices. For example, the dual of the endpoint type T_{ret} defined in Example 4.1 is

$$\overline{T_{ret}} = ?\mathtt{Result}(T_{DS}).\mathtt{end} + ?\mathtt{SetService}(T_{exp}).?\mathtt{Result}(T_{DS}).\mathtt{end}$$

Note that the dual $\overline{T}$ of T cannot be defined by a simple induction on the structure of T according to this intuition because the type of message arguments is *un*affected by duality. In particular we have

$$\begin{array}{rcl}
\overline{\mathtt{rec}\ \alpha.?\mathtt{m}(\alpha).\mathtt{end}} & = & \overline{?\mathtt{m}(\mathtt{rec}\ \alpha.?\mathtt{m}(\alpha).\mathtt{end}).\mathtt{end}} \\
& = & !\mathtt{m}(\mathtt{rec}\ \alpha.?\mathtt{m}(\alpha).\mathtt{end}).\mathtt{end} \\
& \neq & \mathtt{rec}\ \alpha.!\mathtt{m}(\alpha).\mathtt{end}.
\end{array}$$

The interested reader may refer to Bono and Padovani [7] for an equivalent inductive definition of duality.

We list here two important properties of duality, namely that it is an involution and it preserves ranks:

Proposition 4.2. *The following properties hold:*

1. $\overline{\overline{T}} = T$;
2. $\mathsf{rank}(\overline{T}) = \mathsf{rank}(T)$.

Proof. Item (1) is an easy consequence of the definition of duality (Definition 4.1). Item (2) follows from the fact that the rank is only affected by the nesting of transaction types in T and internal/external choices are treated in the same way by rule [WF-PREFIX]. □

4.2. Type Weight

In previous work by Bono and Padovani [7] it was observed that the delegation of endpoints having some particular types can generate memory leaks even if the delegating process appears to behave correctly with respect to the type of the endpoints it uses. For example, the process

$$P \stackrel{\text{def}}{=} \mathtt{open}(a,b).a!\mathtt{m}(b).\mathtt{close}(a) \tag{1}$$

uses a and b according to the endpoint types

$$T = !\mathtt{m}(S).\mathtt{end} \qquad \text{and} \qquad S = \mathtt{rec}\ \alpha : 0.?\mathtt{m}(\alpha).\mathtt{end} \tag{2}$$

respectively. Note that $\overline{T} = S$, therefore the complementarity of actions performed on the peer endpoints a and b is guaranteed. Now, the process P sends endpoint b over endpoint a. According to T, the process is indeed entitled to send an $\mathtt{m}$ message with argument of type S on a and b has precisely that type. After the output operation, the process no longer owns endpoint b and endpoint a is deallocated. Despite its apparent correctness, P generates a leak, as shown by the reduction:

$$\begin{array}{rcl} \emptyset \fatsemi P & \to & a \mapsto [b,\varepsilon], b \mapsto [a,\varepsilon] \fatsemi a!\mathtt{m}(b).\mathtt{close}(a) \\ & \to & a \mapsto [b,\varepsilon], b \mapsto [a,\mathtt{m}(b)] \fatsemi \mathtt{close}(a) \\ & \to & b \mapsto [a,\mathtt{m}(b)] \fatsemi \mathtt{done} \end{array}$$

In the final configuration we have $\mu\text{-}\mathsf{reach}(\mathsf{fn}(\mathtt{done})) = \emptyset$ while $\mathsf{dom}(\mu) = \{b\}$. In particular, the endpoint b is no longer reachable and therefore this configuration violates condition (1) of Definition 3.5. A closer look at the heap in the reduction above reveals that the problem lies in the cycle involving b resulting from the send operation $a!\mathtt{m}(b)$: it is as if the $b \mapsto [a,\mathtt{m}(b)]$ region of the heap

needs not be owned by any process because "it owns itself". Fortunately, it is possible to detect the situations in which these cycles may be generated by looking at the structure of the types of the endpoints that are sent as messages. More specifically, for each endpoint type we compute a value in the set $\mathbb{N} \cup \{\infty\}$, which we call *weight*, that estimates the length of any chain of pointers originating from the queue of the endpoints it denotes. A weight equal to ∞ means that this length can be infinite, in the sense that cycles such as the one shown above may be generated. Then, the type system makes sure that only endpoints having a finite-weight type can be sent as messages, and this has been shown to be enough for preventing these kinds of memory leaks (see Bono and Padovani [7]).

We proceed by recalling here the definition of weight from [7], adapted to our context where we deal also with transaction types:

Definition 4.2 (weight)**.** We say that $\mathscr{W}$ is a *coinductive weight bound* if $(T,n) \in \mathscr{W}$ implies either:

- $T = \mathtt{end}$ or $T = \{\mathtt{e}_i : S_i\}_{i\in I}[\![T'$ or $T =]\!]T'$ or $T = \{!\mathtt{m}_i(S_i).T_i\}_{i\in I}$, or
- $T = \{?\mathtt{m}_i(S_i).T_i\}_{i\in I}$ and $n > 0$ and $(S_i, n-1) \in \mathscr{W}$ and $(T_i, n) \in \mathscr{W}$ for every $i \in I$, or
- $T = \{\mathtt{e}_i : S_i\}_{i\in I}T'$ and $(T', n) \in \mathscr{W}$.

We write $T :: n$ if $(T,n) \in \mathscr{W}$ for some coinductive weight bound $\mathscr{W}$. The *weight* of an endpoint type T, denoted by $\|T\|$, is defined by

$$\|T\| = \min\{n \in \mathbb{N} \mid T :: n\}$$

where we let $\min \emptyset = \infty$. When comparing weights we extend the usual total orders $<$ and $\leq$ over natural numbers so that $n < \infty$ for every $n \in \mathbb{N}$ and $\infty \leq \infty$.

The weight of T is defined as the least of its weight bounds, or ∞ if there is no such weight bound. For example we have $\|\mathtt{end}\| = \|\{!\mathtt{m}_i(S_i).T_i\}_{i\in I}\| = 0$. Indeed, the queues of endpoints with type $\mathtt{end}$ and those in a send state are empty and therefore the chains of pointers originating from them have zero length. The same happens for endpoints whose type is $\{\mathtt{e}_i : S_i\}_{i\in I}[\![T$ and $]\!]T$, since we will enforce the invariant that when a transaction is initiated or successfully terminated, the endpoints involved in it have empty queues. Endpoint types in a receive state have a strictly positive weight. For instance we have $\|?\mathtt{m}(\mathtt{end}).\mathtt{end}\| = 1$ and $\|?\mathtt{m}(?\mathtt{m}(\mathtt{end}).\mathtt{end}).\mathtt{end}\| = 2$. Indeed, the queue of an endpoint with type

?m(end).end may contain another endpoint with an empty queue. Therefore, the chain of pointers originating from the endpoint with type ?m(end).end has at most length 1. If we go back to the endpoint types in (2) that we used to motivate this discussion, we have $\|T\| = 0$ and $\|S\| = \infty$, from which we deduce that endpoints with type S, like b in (1), are not safe to be used as messages.

4.3. Subtyping

The last notion we need before proceeding with the definition of the type system is a *subtyping relation* for endpoint types. Because of the close relationship between endpoint types and session types, the subtyping relation for endpoint types turns out to be a variant of that for session types [14]. However, the peculiar nature of exceptions has interesting consequences. The original subtyping relation for session types is based on the fundamental duality between input and output actions. In particular, it establishes that subtyping is *covariant* for external choices (inputs) and *contravariant* for internal ones (outputs). For example,

$$T = !\mathtt{a}(S_1).T_1 \oplus !\mathtt{b}(S_2).T_2 \leqslant !\mathtt{a}(S_1).T_1 = S$$

is a valid subtyping relation between T and S. The underlying intuition is based on the usual principle of safe substitution of an endpoint of type S with another endpoint of type T. If a (well-typed) process is using an endpoint c of type S, then it can only send an a message on c. So, replacing the endpoint c with another one of type T, which allows both a and b messages to be sent, does not compromise communication safety. In a dual manner,

$$T' = ?\mathtt{a}(S_1).T_1 \leqslant ?\mathtt{a}(S_1).T_1 + ?\mathtt{b}(S_2).T_2 = S'$$

is a valid subtyping relation between T' and S'. In this case, a (well-typed) process using an endpoint of type S' must be capable of handling (at least) a and b messages received from the endpoint. Replacing that endpoint with another one of type T' is safe because from the latter one only a messages can be received.

The covariance and contravariance properties of subtyping with respect to input and output operations follow from the duality of endpoint types associated with peer endpoints: when a process is entitled to send a message on an endpoint, the process using its peer must be ready to receive it, and vice-versa. By contrast, during a transaction, exceptions can be thrown on both peers of a channel. As a consequence, the two transaction types

$$\{\mathtt{e}_1 : S_1, \mathtt{e}_2 : S_2\}[\![T \qquad \text{and} \qquad \{\mathtt{e}_1 : S_1\}[\![T$$

cannot be related. Indeed, if we had $\{\mathtt{e}_1 : S_1, \mathtt{e}_2 : S_2\}[\![T \leqslant \{\mathtt{e}_1 : S_1\}[\![T$, then the process using the endpoint a with type $\{\mathtt{e}_1 : S_1\}[\![T$ might not be prepared to handle the exception $\mathtt{e}_2$ thrown by the process using the peer b of a. Similarly, the process using the peer endpoint b might be unable to handle the exception $\mathtt{e}_2$ thrown on a if we had the opposite relation. In the end, because of the bi-directional nature of exceptions thrown during a transaction, subtyping must be *invariant* for transaction types.

We now proceed to define subtyping formally, extending it to possibly sealed endpoint types in the natural way:

Definition 4.3 (subtyping)**.** *Subtyping* is the largest relation $\leqslant$ such that $t \leqslant s$ implies either:

- $t = [t']$ and $s = [s']$ and $t' \leqslant s'$, or
- $t = s = \mathtt{end}$, or
- $t = \{?\mathtt{m}_i(T_i).T_i'\}_{i\in I}$ and $s = \{?\mathtt{m}_i(S_i).S_i'\}_{i\in I\cup J}$ and $T_i \leqslant S_i$ and $T_i' \leqslant S_i'$ for every $i \in I$, or
- $t = \{!\mathtt{m}_i(T_i).T_i'\}_{i\in I\cup J}$ and $s = \{!\mathtt{m}_i(S_i).S_i'\}_{i\in I}$ and $S_i \leqslant T_i$ and $T_i' \leqslant S_i'$ for every $i \in I$, or
- either ($t = \{\mathtt{e}_i : T_i\}_{i\in I}[\![T$ and $s = \{\mathtt{e}_i : S_i\}_{i\in I}[\![S$) or ($t = \{\mathtt{e}_i : T_i\}_{i\in I}T$ and $s = \{\mathtt{e}_i : S_i\}_{i\in I}S$) and $T_i \leqslant S_i$ for every $i \in I$ and $T \leqslant S$, or
- $t = [\![T$ and $s = [\![S$ and $T \leqslant S$.

According to the definition of $\leqslant$, the covariance and contravariance properties for external and internal choices informally introduced earlier are extended to message argument types, in the usual manner. Observe that subtyping is always covariant with respect to continuations. It is easy to show that $\leqslant$ is a pre-order that is contravariant with respect to duality:

Proposition 4.3. *The following properties hold:*

1. *$\leqslant$ is reflexive and transitive;*
2. *$T \leqslant S$ if and only if $\overline{S} \leqslant \overline{T}$.*

Proof. See Gay and Hole [14]. Transaction types do not pose additional issues. □

In Section 4.2 we have introduced a notion of weight that will be used in the type system for discriminating between safe and unsafe messages. Since the weight is computed on the (static) type of endpoints and subtyping allows for the substitution of endpoints with related but possibly different types, one important question arises whether subtyping and type weight are coherent with each other. This is indeed the case:

Proposition 4.4. *$T \leqslant S$ implies $\|T\| \leq \|S\|$.*

Proof. It is easy to see that $\mathscr{W} = \{(T,n) \mid \exists S : T \leqslant S \;\&\; S :: n\}$ is a coinductive weight bound. In particular, when T is an internal choice we have $T :: 0$ regardless of the number of branches in T. □

4.4. Typing Processes

We can now proceed to defining a type system for processes. A *type environment* is a finite map $\Gamma = \{u_i : t_i\}_{i\in I}$ from names to types. We write $\mathsf{dom}(\Gamma)$ for the domain of Γ, namely the set $\{u_i\}_{i\in I}$; we write Γ,Γ' for the union of Γ and Γ' when $\mathsf{dom}(\Gamma) \cap \mathsf{dom}(\Gamma') = \emptyset$; finally, we write $\Gamma \vdash u : t$ if $\Gamma(u) = t$. An *exception environment* $\tilde{\mathscr{E}}$ is a finite sequence $\mathscr{E}_1 \cdots \mathscr{E}_n$ of sets of exceptions. We write $\mathsf{e} \in \tilde{\mathscr{E}}$ if $\mathsf{e} \in \mathscr{E}_k$ for some $k \in \{1,\ldots,n\}$. We say that a type t is *local*, written $\mathsf{local}(t)$, if t is not sealed and has a null rank, namely $t = T$ for some T such that $\mathsf{rank}(T) = 0$. Intuitively, a local type denotes an endpoint that can be modified (its type is not sealed) and is not involved in any transaction. We extend the notion of local types to type environments so that $\mathsf{local}(\Gamma)$ holds if every type in the codomain of Γ is local.

The typing rules for processes are inductively defined in Table 8. Judgments have the form

$$\tilde{\mathscr{E}}; \Gamma \vdash P$$

denoting that process P is well typed in the exception environment $\tilde{\mathscr{E}}$ and type environment Γ. In particular, P can only throw exceptions that occur in $\tilde{\mathscr{E}}$. The type system makes use of a global process environment Σ associating process variables X with pairs $(\tilde{t}, \tilde{\mathscr{E}})$ containing the type of the parameters of X as well as the exception environment $\tilde{\mathscr{E}}$ in which X is supposed to be invoked. It is understood that the process environment Σ contains associations for all the global definitions D and that the judgment $\Sigma \vdash D$ defined by

$$\frac{\Sigma(X) = (\tilde{t}, \tilde{\mathscr{E}}) \qquad \tilde{\mathscr{E}}; \tilde{u} : \tilde{t} \vdash P}{\Sigma \vdash X(\tilde{u}) \stackrel{\text{def}}{=} P}$$

[T-Inaction]
$$\tilde{\mathscr{E}};\emptyset \vdash \texttt{done}$$

[T-Throw]
$$\frac{\mathsf{e} \in \tilde{\mathscr{E}}}{\tilde{\mathscr{E}};\Gamma \vdash \texttt{throw}\ \mathsf{e}}$$

[T-Close]
$$\frac{\tilde{\mathscr{E}};\Gamma \vdash P}{\tilde{\mathscr{E}};\Gamma, u : \texttt{end} \vdash \texttt{close}(u).P}$$

[T-Invoke]
$$\frac{\Sigma(X) = (\tilde{s}, \tilde{\mathscr{E}}) \qquad \tilde{t} \leqslant \tilde{s}}{\tilde{\mathscr{E}};\tilde{u} : \tilde{t} \vdash X\langle \tilde{u} \rangle}$$

[T-Open]
$$\frac{\vdash T : 0 \qquad \tilde{\mathscr{E}};\Gamma, a : T, b : \overline{T} \vdash P}{\tilde{\mathscr{E}};\Gamma \vdash \texttt{open}(a,b).P}$$

[T-Send]
$$\frac{k \in I \qquad S \leqslant S_k \qquad \|S\| < \infty \qquad \tilde{\mathscr{E}};\Gamma, u : T_k \vdash P}{\tilde{\mathscr{E}};\Gamma, u : \{!\mathtt{m}_i(S_i).T_i\}_{i\in I}, v : S \vdash u!\mathtt{m}_k(v).P}$$

[T-Choice]
$$\frac{\tilde{\mathscr{E}};\Gamma \vdash P \qquad \tilde{\mathscr{E}};\Gamma \vdash Q}{\tilde{\mathscr{E}};\Gamma \vdash P \oplus Q}$$

[T-Receive]
$$\frac{S_i \leqslant S_i'^{\ (i\in I)} \qquad \tilde{\mathscr{E}};\Gamma, u : T_i, x_i : S_i' \vdash P_i^{\ (i\in I)}}{\tilde{\mathscr{E}};\Gamma, u : \{?\mathtt{m}_i(S_i).T_i\}_{i\in I} \vdash \sum_{i\in I\cup J} u?\mathtt{m}_i(x_i).P_i}$$

[T-Parallel]
$$\frac{\tilde{\mathscr{E}};\Gamma_1 \vdash P \qquad \tilde{\mathscr{E}};\Gamma_2 \vdash Q}{\tilde{\mathscr{E}};\Gamma_1,\Gamma_2 \vdash P \mid Q}$$

[T-Try]
$$\frac{\tilde{\mathscr{E}}\{\mathsf{e}_j\}_{j\in J};[\Gamma],\{u_i : T_i\}_{i\in I} \vdash P \qquad \tilde{\mathscr{E}};\Gamma,\{u_i : S_{ij}\}_{i\in I} \vdash Q_j^{\ (j\in J)}}{\tilde{\mathscr{E}};\Gamma,\{u_i : \{\mathsf{e}_j : S_{ij}\}_{j\in J}[\![T_i]\!]_{i\in I} \vdash \texttt{try}(\{u_i\}_{i\in I})\ \{\mathsf{e}_j : Q_j\}_{j\in J} P}$$

[T-Commit]
$$\frac{\mathsf{local}(\Gamma_2) \qquad \tilde{\mathscr{E}};\Gamma_1,\{u_i : T_i\}_{i\in I},\Gamma_2 \vdash P}{\tilde{\mathscr{E}}\mathscr{E};[\Gamma_1],\{u_i : [\![T_i]\!]\}_{i\in I},\Gamma_2 \vdash \texttt{commit}(\{u_i\}_{i\in I}).P}$$

Table 8: Typing rules for processes.

holds. In particular, *all* of the free names of P must occur in its binding variable $X(\tilde{u})$.

We describe the typing rules for processes in the following paragraphs. Rule [T-INACTION] states that the idle process is well typed only in the empty type environment. This is a standard rule for linear type systems implying, in our case, that the terminated process has no leaks.

Rule [T-CLOSE] states that a process $\texttt{close}(u).P$ is well typed provided that u corresponds to an endpoint with type `end`, on which no further interaction is possible, and P is well typed in the remaining type environment.

Rule [T-OPEN] deals with the creation of a new channel, which is visible in the continuation process as two peer endpoints typed by dual endpoint types. The premise $\vdash T : 0$ means that newly created endpoints have no pending transactions on them.

Rule [T-SEND] states that a process $u!\mathtt{m}(v).P$ is well typed if u is associated with an endpoint type T that permits the output of `m` messages. The type S of the argument v must be unsealed, finite-weight, and has to be a subtype of the expected type in the endpoint type. Finally, the continuation P must be well typed in a type environment where the endpoint u is typed according to the continuation T_k of T and the endpoint v is no longer visible. This models the fact that the ownership of v is transferred to the process that receives the message.

Rule [T-RECEIVE] deals with inputs: a process waiting for a message from an endpoint $u : \{?\mathtt{m}_i(S_i).T_i\}_{i\in I}$ is well typed if it can deal with all of the messages $\mathtt{m}_i$. The continuation processes may use the endpoint u according to the endpoint type T_i and can access the message argument x_i of some supertype S_i' of S_i.

Rules [T-CHOICE] and [T-PARALLEL] are standard. In the latter, the type environment is split into two disjoint environments to type the processes being composed.

Rule [T-INVOKE] declares that a process invocation $X\langle\tilde{u}\rangle$ is well typed provided that the number and type of actual parameters $\tilde{u}$ match the number and type of formal parameters in $\Sigma(X)$ and that the process is invoked in the correct exception environment. In this rule we write $\tilde{t} \leqslant \tilde{s}$ for the pointwise extension of $\leqslant$ to sequences of types.

We now turn our attention to the constructs dealing with transactions and exceptions.

Rule [T-THROW] states that the process $\texttt{throw}\ \mathtt{e}$ is well typed in *any* type environment, provided that it occurs within a transaction (the exception being thrown must be among the ones occurring in the exception environment). For this reason, the violation of linearity for the assumptions in the type environment

is only apparent, as control will be transferred at runtime to some appropriate exception handler.

Rule [T-TRY] deals with transaction initiations. All the endpoints in the decoration U must have a type allowing them to be involved in a transaction, while the types of other names are sealed so that P is prevented from using them until the transaction is terminated. Seals are not applied in the type environment for the handlers since they execute only if and when the transaction is aborted and therefore act outside of the transaction. Note that the admitted exceptions are augmented in P but not in Q.

Rule [T-COMMIT] is almost the dual of rule [T-TRY] and deals with transaction termination. Again, the endpoints in the decoration U must have a matching type in the context indicating the end of the transaction. Names with a sealed type must have been inherited from the context surrounding the transaction being terminated, so a seal is stripped off them in the continuation P. Names with a local type must have been created within the transaction being terminated, and can be used in the continuation as well. Note that the rightmost set $\mathscr{E}$ in the exception environment is stripped off when type checking P, since P executes after the transaction has terminated hence outside of the scope where the exceptions in $\mathscr{E}$ can be thrown.

Observe that the type system requires the endpoints specified in a `commit` process to be exactly the same as the ones in the corresponding `try`. This is a consequence of the properties of well-formed endpoint types: endpoints involved in a transaction have a type with a strictly positive rank (see [WF-INITIATE] in Table 7) meaning that they cannot be closed (because `end` has null rank) and they cannot be sent as messages (again because [WF-PREFIX] requires messages to have a type with null rank). For the same reason they cannot be qualified as local, because local endpoints have a type with a null rank. Therefore, the set $\{u_i\}_{i\in I}$ associated with a given `try` process will be exactly the same set associated with the corresponding `commit` process.

Example 4.2. Using the types defined in Example 4.1, the reader can verify that the bodies of the process definitions in Figure 3 for `GetNextDiskPath`, `Loop`, and `Finally` are respectively well typed according to the type environments

$$\begin{array}{lcl} \Gamma_1 & = & DS : ?\texttt{NewClientEndpoint}(T_{ns}).T_{DS}, ret : T_{ret} \\ \Gamma_2 & = & ns : T'_{ns}, DS : T_{DS}, ret : T_{ret} \\ \Gamma_3 & = & ns : \texttt{end}, DS : T_{DS}, ret : T_{ret} \end{array}$$

where

$$T'_{ns} = !\texttt{Register}(T_{imp}).(?\texttt{AckRegister}().]\!]\texttt{end} + ?\texttt{NakRegister}(T_{imp}).T_{ns})$$

is an appropriate residual of the unfolding of T_{ns}. Note the role played by subtyping in this example: *ret* is used according to the type $!\texttt{Result}(T_{DS}).\texttt{end}$ in `Finally` and according to the type $!\texttt{SetService}(T_{exp}).!\texttt{Result}(T_{DS}).\texttt{end}$ in `Loop`. Since T_{ret} is a subtype of both these types, *ret* can be passed to `Finally` and `Loop` thanks to the subtyping relation in [T-INVOKE]. ■

4.5. Typing the Heap

The typing rules in Table 8 are not sufficient for proving the soundness of the type system, because they are solely concerned with the static syntax of processes. At runtime, we must take into account running transaction processes (see Tables 2 and 4) as well as the heap. Indeed, since inter-process communication relies on heap-allocated structures, several properties of well-behaved processes depend on properties of the heap saying that its content is consistent with a given type environment. In this section and in the following one we develop a type system for the runtime components of our process language. We remark that the programmer is solely concerned with the typing rules for static processes presented in Section 4.4, while the technical material presented hereafter, which builds on and extends the previous one, is only required for proving that the type system is sound.

Just as we have type checked a process P against a type environment that associates types with the names occurring in P, we also need to check that the heap is consistent with respect to the same environment. This leads to a notion of well-typed heap that we develop in this section. More precisely, well-typedness of a heap μ is checked with respect to a pair $\Gamma_0;\Gamma$ of type environments: the context Γ_0,Γ must provide type information for *all* the allocated structures in μ (that is, $\mathsf{dom}(\Gamma_0,\Gamma) = \mathsf{dom}(\mu)$); the splitting $\Gamma_0;\Gamma$ distinguishes the pointers in $\mathsf{dom}(\Gamma)$ from the pointers in $\mathsf{dom}(\Gamma_0)$ so that Γ contains the *roots* of μ, namely the pointers that are not referenced from any endpoint structure in the heap, while Γ_0 contains pointers that are referenced from some endpoint structure.

Among the properties that must be enforced is the complementarity between the endpoint types associated with peer endpoints. This notion of complementarity does not coincide with duality because the communication model is asynchronous: since messages can accumulate in the queue of an endpoint before they are received, the types of peer endpoints can be misaligned. The two peers are guaranteed to have dual types only when their queues are both empty. In general, we need to compute the actual endpoint type of an endpoint by taking into account the messages in its queue. To this aim we introduce a $\mathsf{tail}(\cdot,\cdot)$ function for

endpoint types such that

$$\mathsf{tail}(T, \mathtt{m}_1(S_1) \cdots \mathtt{m}_n(S_n)) = T'$$

indicates that messages having tag $\mathtt{m}_i$ and an argument of type S_i can be received in the specified order from an endpoint with type T, which can be used according to type T' thereafter. The function is inductively defined by the following rules:

$$\mathsf{tail}(T, \varepsilon) = T$$

$$\frac{k \in I \qquad S \leqslant S_k}{\mathsf{tail}(\{?\mathtt{m}_i(S_i).T_i\}_{i \in I}, \mathtt{m}_k(S)) = T_k} \qquad \frac{\mathsf{tail}(T, \mathtt{m}(S)) = T'}{\mathsf{tail}(\{\mathsf{e}_i : S_i\}_{i \in I} T, \mathtt{m}(S)) = T'}$$

$$\frac{\mathsf{tail}(T, \mathtt{m}_1(S_1)) = T' \qquad \mathsf{tail}(T', \mathtt{m}_2(S_2) \cdots \mathtt{m}_n(S_n)) = T''}{\mathsf{tail}(T, \mathtt{m}_1(S_1)\mathtt{m}_2(S_2) \cdots \mathtt{m}_n(S_n)) = T''}$$

Note that $\mathsf{tail}(T, \mathtt{m}(S))$ is undefined when $T = \mathtt{end}$ or T is an internal choice or T denotes the initiation or the termination of a transaction. This will enforce the property that the queue of endpoints having these types must be empty. In the particular case of transaction initiation, this makes sure that, if an exception is thrown, heap restoration simply amounts to emptying the queues of the endpoints involved in the transaction (mechanism (A) in Section 2). The fact that the queues of the endpoints involved in the transaction are guaranteed to be empty at the end of a transaction is solely motivated by our notion of duality (Definition 4.1), which demands a perfect correspondence between the actions on such endpoints during a transaction. In principle, it would be possible to relax duality in such a way that a message sent within a transaction is received only after the transaction is terminated. However, it would still be necessary for the receive operation to first wait for the actual termination of the transaction, for otherwise the soundness of the transaction would be compromised. This means that this increased flexibility in the syntax of programs would bear no concrete advantage in their semantics.

We now have all the notions to express the well-typedness of a heap μ with respect to a pair $\Gamma_0; \Gamma$ of type environments.

Definition 4.4 (well-typed heap)**.** Let $\mathsf{dom}(\Gamma_0) \cap \mathsf{dom}(\Gamma) = \emptyset$. We write $\Gamma_0; \Gamma \Vdash \mu$ if all of the following conditions hold:

1. If $a \mapsto [b, \mathfrak{Q}] \in \mu$ and $b \mapsto [a, \mathfrak{Q}'] \in \mu$, then either $\mathfrak{Q} = \varepsilon$ or $\mathfrak{Q}' = \varepsilon$.

2. If $a \mapsto [b, \mathtt{m}_1(c_1) :: \cdots :: \mathtt{m}_n(c_n)] \in \mu$, then

$$\mathsf{tail}(T, \mathtt{m}_1(S_1) \cdots \mathtt{m}_n(S_n)) = S$$

where $\Gamma_0, \Gamma \vdash a : [\cdots[T]\cdots]$ and $\Gamma_0 \vdash c_i : S_i$ and $\|S_i\| < \infty$ and $\vdash S_i : 0$ for $1 \leq i \leq n$ and $b \mapsto [a, \varepsilon] \in \mu$ implies $\Gamma_0, \Gamma \vdash b : [\cdots[\overline{S}]\cdots]$ and $b \notin \mathsf{dom}(\mu)$ implies $S = \mathtt{end}$.

3. $\mathsf{dom}(\mu) = \mathsf{dom}(\Gamma_0, \Gamma) = \mu\text{-}\mathsf{reach}(\mathsf{dom}(\Gamma))$;

4. $A \cap B = \emptyset$ implies $\mu\text{-}\mathsf{reach}(A) \cap \mu\text{-}\mathsf{reach}(B) = \emptyset$ for every $A, B \subseteq \mathsf{dom}(\Gamma)$.

Condition (1) requires that at least one of the queues of peer endpoints in a well-typed heap is empty. This invariant corresponds to half-duplex communication and is ensured by duality of endpoint types associated with peer endpoints, since a well-typed process cannot send messages on an endpoint until it has read all the pending messages from the corresponding queue (we will see in Example 4.4 how to safely circumvent half-duplex communication thanks to transactions). Condition (2) requires that the content of the queue of an endpoint must be consistent with the type of the endpoint, in the sense that the messages in the queue have the expected tag and an argument with the expected type. In addition, the endpoint types of message arguments must all have finite weight and null rank. Finally, the endpoint types of peer endpoints are dual of each other, modulo the content of the non-empty queue. Condition (3) states that the type environment Γ_0, Γ must specify a type for all of the allocated objects in the heap and, in addition, every object (located at) a in the heap must be reachable from a root $b \in \mathsf{dom}(\Gamma)$. Finally, condition (4) requires the uniqueness of the root for every allocated object. Overall, since the roots are distributed linearly among the processes of the system, conditions (3) and (4) guarantee that every allocated object belongs to one and only one process.

There are a few subtleties regarding conditions (1) and (2) and the fact that, in condition (2), the property $b \mapsto [a, \varepsilon] \in \mu$ is the head of an implication. First of all, condition (2) must hold for both peers of a channel, therefore if a is the peer with the empty queue ($n = 0$) while b has messages in its queue, then the type of a is not necessarily the dual of the type of b. The correct dual correspondence is checked when the symmetric pair of endpoints is considered. Second, it is possible that at some point only one endpoint of a channel is allocated. For example, the well-typed process $\mathtt{open}(a, b).\mathtt{close}(b).\mathtt{close}(a)$ reduces to $\mathtt{close}(a)$ in a configuration where the heap contains only $a \mapsto [b, \varepsilon]$. When this happens,

[T-RUNNING PROCESS]
$$\frac{\Gamma_0;\Gamma_R,\Gamma \Vdash \mu \qquad \tilde{\mathscr{E}};\Gamma \vdash P}{\tilde{\mathscr{E}};\Gamma_0;\Gamma_R;\Gamma \vdash \mu \wr P}$$

[T-RUNNING PARALLEL]
$$\frac{\tilde{\mathscr{E}};\Gamma_0;\Gamma_R,\Gamma_2;\Gamma_1 \vdash \mu \wr P \qquad \tilde{\mathscr{E}};\Gamma_0;\Gamma_R,\Gamma_1;\Gamma_2 \vdash \mu \wr Q}{\tilde{\mathscr{E}};\Gamma_0;\Gamma_R;\Gamma_1,\Gamma_2 \vdash \mu \wr P \mid Q}$$

[T-RUNNING TRANSACTION]
$$\frac{\begin{array}{c}\mu\text{-balanced}(\{a_i : S_{ij}\}_{i\in I})^{\,(j\in J)} \qquad \mu\text{-balanced}(B) \qquad \mathsf{local}(\Gamma_2) \\ \{a_i\}_{i\in I} \cup B = \mu\text{-reach}(\{a_i\}_{i\in I} \cup \mathsf{dom}(\Gamma_2)) \\ \tilde{\mathscr{E}}\{\mathsf{e}_j\}_{j\in J};\Gamma_0;\Gamma_R;[\Gamma_1],\{a_i : T_i\}_{i\in I},\Gamma_2 \vdash \mu \wr P \qquad \tilde{\mathscr{E}};\Gamma_1,\{a_i : S_{ij}\}_{i\in I} \vdash Q_j\,{}^{(j\in J)}\end{array}}{\tilde{\mathscr{E}};\Gamma_0;\Gamma_R;\Gamma_1,\{a_i : \{\mathsf{e}_j : S_{ij}\}_{j\in J} T_i\}_{i\in I},\Gamma_2 \vdash \mu \wr \langle \{a_i\}_{i\in I}, B, \{\mathsf{e}_j : Q_j\}_{j\in J} P\rangle}$$

Table 9: Typing rules for configurations.

the type of the remaining endpoint forbids any send operation (last property of condition (2)). Note that condition (1) is not implied by condition (2) and both conditions are necessary.

4.6. Typing Configurations

Table 9 defines typing rules for configurations $\mu \wr P$ as an extension of the typing rules for processes. Judgments have the form

$$\tilde{\mathscr{E}};\Gamma_0;\Gamma_R;\Gamma \vdash \mu \wr P$$

and state that the configuration $\mu \wr P$ is well typed with respect to the exception environment $\tilde{\mathscr{E}}$ and the triple $\Gamma_0;\Gamma_R;\Gamma$ of type environments. Intuitively, Γ is the type environment used to type check P, Γ_R is the type environment describing the type of root pointers owned by processes that are running in parallel with P, and Γ_0 describes the type of pointers that occur in some queue.

Rule [T-RUNNING PROCESS] lifts well-typed processes to well-typed configurations by requiring the heap to be well typed with respect to the pair of environments $\Gamma_0;\Gamma_R,\Gamma$ where Γ_R,Γ represents the whole set of roots obtained from those owned by the process being typed (in Γ) and those owned by processes in parallel with it (in Γ_R).

Rule [T-RUNNING PARALLEL] is similar to [T-PARALLEL], except that it deals with three type environments which are appropriately rearranged for keeping track of the roots of the heap.

Rule [T-RUNNING TRANSACTION] captures the basic properties regarding running transactions $\langle \{a_i\}_{i\in I}, B, \{\mathsf{e}_j : Q_j\}_{j\in J} P\rangle$, which we describe here. The rule

makes use of a balancing predicate over type environments that generalizes the notion of balancing for sets of pointers (Definition 3.2):

Definition 4.5. We say that Γ is *balanced* in μ, written μ-balanced(Γ), if $a \in \mathsf{dom}(\Gamma)$ and $a \overset{\mu}{\leftrightarrow} b$ imply $b \in \mathsf{dom}(\Gamma)$ and $\Gamma(a) = \overline{\Gamma(b)}$.

First of all, it must be possible to partition the type environment into three parts Γ_1, $\{a_i : \{\mathsf{e}_j : S_{ij}\}_{j\in J} T_i\}_{i\in I}$, and Γ_2 such that the environment Γ_1 corresponds to the endpoints owned by P but which are not involved in the transaction. Consequently, the types of these endpoints are sealed in the judgment corresponding to the typing of P. The environment $\{a_i : \{\mathsf{e}_j : S_{ij}\}_{j\in J} T_i\}_{i\in I}$ corresponds to the endpoints involved in the transaction (the first component of the running transaction process), and their type indicates that the transaction is in progress. The environment Γ_2 corresponds to the endpoints that have been allocated inside the transaction. Their type is not sealed in the judgment corresponding to the typing of P. The premises μ-balanced($\{a_i : S_{ij}\}_{i\in I}$) for every $j \in J$ and μ-balanced(B) indicate that the set of all the endpoints to which P has full access is balanced. Therefore, the transaction operates in a closed scope and cannot have "side effects" from the point of view of other processes. The first premise indicates, in addition, that the types S_{ij} associated with peer endpoints are dual of each other (this property is a consequence of well-typedness of the heap before the transaction initiates, but it must be explicitly stated in [T-RUNNING TRANSACTION] where the heap is checked against a type environment where the S_i's do not occur any more). The premise local(Γ_2) identifies the Γ_2 partition of the context corresponding to the endpoints that have been created inside the transaction. The premise $\{a_i\}_{i\in I} \cup B = \mu$-reach($\{a_i\}_{i\in I} \cup \mathsf{dom}(\Gamma_2)$) states that all the endpoints allocated within the transaction have not escaped the scope of the transaction. The last two premises correspond to the premises of rule [T-TRY]. In particular, note that the exception environment is properly augmented when typing the body of the transaction.

Since running transaction processes appear only at runtime as the result of [R-START TRANSACTION] reductions, they can never occur behind a prefix and therefore the three rules in Table 9 cover all possible forms of runtime configurations.

4.7. Type Soundness

We can now formulate the two main results about our framework: well-typedness is preserved by reduction, and well-typed processes are well behaved. Subject

reduction takes into account the possibility that types in the environment may change as the process reduces, which is common in behavioral type theories.

Theorem 4.1 (subject reduction)**.** *Let* $\tilde{\mathscr{E}}; \Gamma_0; \Gamma_R; \Gamma \vdash \mu \fatsemi P$ *and* $\mu \fatsemi P \to \mu' \fatsemi P'$. *Then* $\tilde{\mathscr{E}}; \Gamma_0'; \Gamma_R; \Gamma' \vdash \mu' \fatsemi P'$ *for some* Γ_0' *and* Γ'.

Proof. See Appendix A. □

In fact, the proof of this theorem requires to specify a number of additional properties showing the precise relationship between Γ and Γ_0 (before the reduction) and Γ' and Γ_0' (after the reduction).

Theorem 4.2 (safety)**.** *Let* $\emptyset \vdash_0 P$. *Then* P *is well behaved.*

Proof. See Appendix B. □

We conclude this section with two examples showing how transaction types increase the expressiveness of session types by allowing the safe modeling of timeouts and mixed choices.

Example 4.3 (timeouts)**.** In some cases it is possible and desirable to establish a timeout for a receive operation to succeed. Using transaction types and exceptions, it is easy to model timeouts in our process language. As an example, suppose one is interested in modeling a process

$$a?\mathtt{m}(x).P + \tau.Q$$

which behaves as P as soon as it receives an $\mathtt{m}$ message from endpoint a and reduces to Q through an internal τ move if no message is received after some unspecified amount of time. This can be modeled by means of the process

$$\begin{array}{l} \mathtt{try}(a)\ \{\mathtt{timeout} : Q\} \\ \qquad a?\mathtt{m}(x).\mathtt{commit}(a).P \\ \qquad \mid\ \mathtt{try}(\{\})\ \{\mathtt{ok} : \mathtt{done}\}(\mathtt{throw}\ \mathtt{ok} \oplus \mathtt{throw}\ \mathtt{timeout}) \end{array}$$

where the nondeterministic choice between throwing $\mathtt{ok}$ or $\mathtt{timeout}$ is the abstract representation of the mechanism that activates the timeout. Note that the modeling uses exception propagation to trigger the timeout. The type associated with a is $\{\mathtt{timeout} : S\}[\![?\mathtt{m}(t).]\!]T$ where T describes the behavior of P on a if the $\mathtt{m}$ message is received within the timeout, while S describes the behavior of Q on a if the timeout expires. Note that the modeling in this example is not meant to suggest

an actual implementation of the timeout mechanism, but rather to describe its effect in abstract terms. In particular, the `m` message may have already been sent by the time the timeout expires (and the exception `timeout` is thrown). Yet, the sender of the message should be aware of this eventuality, it should be notified in case the timeout expires, and it should provide a suitable recovery action for this eventuality, possibly involving the resending of the `m` message. This is all guaranteed by the fact that the sender, which uses the peer endpoint of a, must do so according to the endpoint type $\{\mathtt{timeout} : \overline{S}\}[\![\,!\mathtt{m}(t).]\!]\overline{T}$. ■

Example 4.4 (mixed choices)**.** The Sing$^\#$ implementation allows the definition of contracts with so-called *mixed choices*, namely states in which there are two (or more) alternative operations involving both inputs and outputs. If mixed choices were allowed in our type language we would have, for example, endpoint types of the form $\{!\mathtt{a}(t).T, ?\mathtt{b}(s).S\}$ allowing either sending an `a` message or receiving a `b` message. Mixed choices break the half-duplex communication modality and are known to make protocols less robust and prone to deadlock [5, 6]. Yet, they can be safely modeled using transaction types as, for example

$$T_m = \{\mathtt{a} : !\mathtt{a}(t).T, \mathtt{b} : ?\mathtt{b}(s).S\}[\![\,]\!]\mathtt{end}$$

The intuition is that a process behaving as T_m may throw either an `a` or a `b` exception to notify its party as to which operation (output an `a` message or input a `b` message) it will perform. An example of such process is

$$\mathtt{try}(c)\{\mathtt{a} : c!\mathtt{a}(u).P, \mathtt{b} : c?\mathtt{b}(x).Q\}(\mathtt{commit}(c).\mathtt{close}(c) \mid X\langle u\rangle)$$

where

$$X(y) \stackrel{\text{def}}{=} X\langle y\rangle \oplus \mathtt{throw}\ \mathtt{a}$$

represents an internal computation that may eventually throw the `a` exception and cause the sending of the `a` message.

Below is part of the proof derivation showing that the process is well typed in the environment $c : T_m, u : t$. The branch related to the `b` exception has been omitted, but it is analogous to the one for the `a` exception; where necessary, the exception environment $\mathscr{E} = \{\mathtt{a}, \mathtt{b}\}$ is used:

$$\dfrac{\dfrac{\dfrac{\mathscr{E};\emptyset \vdash \texttt{done}}{\mathscr{E};c:\texttt{end} \vdash \texttt{close}(c)}}{\mathscr{E};c:[\![]\!]\texttt{end} \vdash \texttt{commit}(c).\texttt{close}(c)} \quad \mathscr{E};u:[t] \vdash X\langle u\rangle \quad \dfrac{\vdots}{c:T \vdash P}}{\dfrac{\mathscr{E};c:[\![]\!]\texttt{end},u:[t] \vdash \texttt{commit}(c).\texttt{close}(c) \mid X\langle u\rangle \quad c:\,!\texttt{a}(t).T,u:t \vdash c!\texttt{a}(u).P}{c:T_m,u:t \vdash \texttt{try}(c)\{\texttt{a}:c!\texttt{a}(u).P,\texttt{b}:c?\texttt{b}(x).Q\}(\texttt{commit}(c).\texttt{close}(c) \mid X\langle u\rangle)}\ \vdots$$

Finally,

$$\dfrac{\mathscr{E};u:[t] \vdash X\langle u\rangle \quad \mathscr{E};u:[t] \vdash \texttt{throw a}}{\mathscr{E};u:[t] \vdash X\langle u\rangle \oplus \texttt{throw a}}$$

is the proof tree proving that the definition of $X\langle u\rangle$ is well typed. ■

5. Related Work

Our research fits in the broad spectrum of works developing hybrid static/dynamic techniques for the controlled management of the heap, and relies on invariants regarding the configuration of heap-allocated structures for enabling the efficient implementation of transactional mechanisms. Controlled heap management has fostered the development of a wide spectrum of techniques aimed at the most diverse purposes, of which we provide here a small account.

Pure functional programming languages are excellent candidates for the implementation of implicitly parallel computations. Parallelism is usually achieved by allowing multiple processing units to independently reduce disjoint parts of a program (represented as a graph) stored in a (possibly virtual) shared memory. Both hardware [15] and software [16] architectures have been explored. In these architectures, the crucial aspect is to achieve an optimal distribution of tasks, taking into account the fact that each processing unit often has its own local memory where subgraphs to be reduced must be copied, and that for improved efficiency it is necessary to take into proper account locality properties of the program [17].

Type-based approaches for enforcing and reasoning on properties of the heap are also popular. The seminal work by Tofte and Talpin [18] describes an *effect type system* for region-based memory management that allows for efficient allocation and deallocation of related heap structures. Interestingly, effects can be seen as primitive forms of behavioral types. There exist type systems with resource annotations that allow computing bounds on the heap space usage of functions and methods. Examples of such type systems are given in [19] for (first-order)

functional programs and in [20] for object-oriented programs. These approaches are usually motivated by the need to provide firm guarantees in code meant to be executed in embedded systems with constrained resources. Such forms of analyses can be carried out also for untyped programs. For instance, Albert et al. [21] work directly on Java bytecode, while Hammond [22] studies bounded time/space semantics for a functional and concurrent language.

Suenaga et al. [23] study a type discipline for safe resource deallocation in concurrent programs with shared memory. The idea is to associate each shared resource with a *fractional ownership*, namely a rational number in the interval $[0,1]$, which denotes the level of sharing of the resource: 0 means that the resource is not owned, 1 means that the resource is owned exclusively, while any intermediate rational number indicates a shared ownership of the resource (often constraining the operations allowed on it). The type system allows a thread to deallocate a resource only if the thread is the only owner of the resource *and* if all of the other resources contained in it have been deallocated or transferred to other threads. In our type system, a similar effect is achieved by the combination of the type rule for the `close` primitive (which allows for the deallocation of endpoints only when they are owned exclusively) and the notion of well-typed heap (which guarantees endpoints with type `end` to have an empty queue).

There are strong analogies between our endpoint types and *usage expressions* defined by Iwama et al. [24], which are used for controlling access to resources (such as files and memory) in a functional language with exceptions. Usage expressions are akin to behavioral types and describe the valid sequence of operations allowed on some resource. In particular, [24] defines a usage constructor $U_1;_E U_2$ where U_1 describes how the resource is accessed under "nominal" conditions while U_2 describes how the resource is accessed if an exception is thrown. The structure of this usage constructor resembles that of a transaction type $\{E : U_2\}[\![U_1$. Because usage expressions can be composed in sequence, there is no need to explicitly mark the points where the scope of an exception ends.

The present work continues the type-based formalization of Singularity OS described by Bono and Padovani [7]. To simplify the formal development of the present paper we dropped polymorphism and non-linear types from the type system in [7]. These are orthogonal features that are independent of exception handling and can be added without affecting the results we have presented here. A radically different approach for the static analysis of Singularity processes is explored by Villard et al. [6, 25], where the authors develop a proof system based on a variant of *separation logic*. Exceptions are not taken into account in these works.

The works more closely related to ours, and which we used as starting points, are by Carbone et al. [11] and Capecchi et al. [12]. In [11], which was the first to investigate exceptions in calculi for session-oriented interactions and to propose type constructs to describe explicitly, at the type level, the handling of exceptional events, it is possible to associate an exception handler to a whole (dyadic) session; [12] generalizes this idea to multiparty sessions (those with multiple participants) and allows the same channel to be involved, at different times, in different `try` blocks, each with its own dedicated exception handler. In both [11] and [12] it is possible that messages already present in channel queues at the time an exception occurs are forgotten. In our context, this would immediately yield memory leaks, which we avoid by keeping track of the resources allocated during a transaction and by restoring the system to a consistent configuration in case an exception is thrown. State restoration is made possible in our context because the system is not distributed and the heap is shared by the communicating processes. Neither [11] nor [12] consider session delegation, namely the communication of channels. Also, in [12] the type system forces inner `try` blocks to use a subset of the channels involved in outer blocks. We relax these restrictions and allow locally created channels to be involved in inner transactions. The most notable difference between [12] and the present work regards the semantics of exceptions in nested transactions: in [12], an exception thrown in one transaction is suspended as long as there are active handlers in the nested ones. This semantics is motivated by the observation that, in a distributed setting, it may be desirable to complete the execution of potentially critical handlers before outermost handlers take control. Our semantics allows handlers of outer transactions to take control at any time following the throwing of an exception. As a consequence, more constrained policies, such as the one adopted in [12], can be implemented without invalidating the results presented in our work.

The recent interest on Web services has spawned a number of works investigating (long running) transactions in a distributed setting; a detailed survey with many references is provided by Ferreira et al. [26]. In our context, the components Q_i of a process $\langle A, B, \{\mathsf{e}_i : Q_i\}_{i \in I} P \rangle$ are analogous to *compensation handlers*. The main difference between our handlers and those used for compensations is that, in the latter case, it is usually made the assumption that it is not possible to restore the state of the system as it was at the beginning of the transaction. In our case, state restoration is made possible by the fact that the system is local and all the interactions occur through shared memory. In this context, we can rely on some native support from the runtime system to properly cleanup the state of the system and avoid memory leaks.

The operational semantics of exceptions and exception handling in the present paper has been loosely inspired by that of Haskell memory transactions described by Harris et al. [27]. In particular, our semantics describes *what* happens when an exception is thrown but not *how* exception notification and state restoration are implemented. In this sense our semantics is somewhat more abstract than the semantics given in similar works [12]. The semantics of [27] uses a clever combination of small- and big-step reduction rules and is even more abstract than ours. Technically, having an abstract semantics is an advantage because it allows the meaning of programs to be expressed more concisely and in an implementation-independent way. However, because the big-step semantics is affected by diverging programs, it is more appropriate in a functional setting where non-termination is typically considered as a misbehavior. In our context, where non-terminating processes are useful, we had to resort to a more detailed operational semantics that dynamically keeps track of the allocated memory within a transaction.

Donnelly and Fluet [28] put forward a programming abstraction called *transactional events* for the modular composition of communication events into transactions with an all-or-nothing semantics. Their approach focuses on finding synchronization paths between threads communicating synchronously, while in our case transactions are required for preserving type consistency of endpoints and for undoing the effects of asynchronous communication.

Inadequacy of the standard error handling mechanisms provided by mainstream programming languages has already been recognized, even in sequential and communication-free scenarios. Weimer and Necula [29], Weimer [30], Weimer and Necula [31] develop a static analysis technique that spots error handling mistakes concerning proper resource release. Their technique is based on finite-state automata (in other words, a basic form of behavioral type) for keeping track of the state of resources along all possible execution paths. They also propose a more effective mechanism for preventing runtime errors. The basic idea is to accumulate compensation actions regarding resources on a *compensation stack* as resources are allocated. This technique closely resembles dynamic compensations in [26]. Because of their dynamic nature, compensation stacks do not provide any assistance as far as type consistency is concerned.

6. Conclusions and Future Work

We have formalized a core language of processes that communicate and synchronize through the copyless message passing paradigm and can throw exceptions. In this context, where the sharing of data and explicit memory allocation require

controlled policies on the ownership of heap-allocated objects, special care must be taken when exceptions are thrown to prevent communication errors (arising from misaligned states of channel endpoints) and memory leaks (resulting from messages forgotten in endpoint queues). We have studied a type system guaranteeing some safety properties, in particular that well-typed processes are free from communication errors and do no leak memory even in presence of (caught) exceptions. We have taken advantage of invariants guaranteed by the type system for taming the implementation costs of exception handling: the queues of endpoints involved in a transaction are guaranteed to be empty when the transaction starts, so that state restoration in case of exception simply means emptying such queues; also, only endpoints local to a transaction can be freed inside the transaction, so that state restoration in case of exception does not involve re-allocations.

The choice of Sing$^\#$ as our reference language has been motivated by the fact that the Singularity code base provides concrete programming patterns that the formal model is supposed to cover (for example, previous works on exception handling for session-oriented languages by Carbone et al. [11], Capecchi et al. [12] do not consider delegation inside `try`-blocks, which instead is ubiquitous in Sing$^\#$). In addition, Sing$^\#$ already accommodates channel contracts, which play a crucial role in our formalization. However, we claim that our approach is abstract enough to be applicable to other programming languages and paradigms, provided that suitable type information (possibly in the form of code annotations) is attached to channel endpoints.

To measure the practical impact of our mechanism of exception handling, one can take advantage of the availability of Singularity's source code for verifying whether the constraints imposed by the type discipline are reasonable in practice. Even without thorough investigations, however, we are able to provide some favorable arguments to our type discipline, in particular with respect to the weight and rank restrictions on types. As regards type weights, one has to consider that messages allocated on the exchange heap are explicitly managed by means of reference counting [2], which notoriously falls short in handling cyclic data structures, and that the finite-weight restriction on the type of communicated messages is just aimed at preventing cycles in the exchange heap. Regarding ranks, the subclass of well-formed, null-ranked types are just those in which transactions are properly balanced. In fact, the notion of well formedness arises solely because of our choice of modeling transactions using two matching constructs `try` and `commit` marking their beginning and end. Their balancing arises naturally in a structured language such as Sing$^\#$. We also plan to work on a prototype implementation of the exception handling so as to explore its practical costs.

Even if the type system has been tailored for a process calculus with a minimal set of critical features, it can be easily extended to incorporate commonly used programming constructs. Some restrictions of the type system can also be relaxed without endangering its soundness. For example, according to rule [T-SEND], only local endpoints (those that have an unsealed and null-ranked type) can be sent as messages inside transactions. This restriction results from the syntax of endpoint types (requiring that message arguments must have an unsealed type) and from rule [WF-PREFIX] regarding well-formed endpoint types (requiring that message argument types must have null rank). Endpoints with a sealed type cannot be accessed from within a transaction, because all the operations that modify the state of the heap require the access to endpoints with unsealed type. Therefore, we claim that endpoints with a sealed type are also safe to be sent as messages. In case an exception is thrown, the only thing that must be restored is their ownership at the beginning of the transaction. Nonetheless, the proof of this relaxed discipline seems to require a non-trivial modification of rule [T-RUNNING TRANSACTION], which is already quite elaborate in the present state, to account for the fact that the state of endpoints with a sealed type can change as the result of concurrent threads that execute *outside* of the transaction.

A. Proof of Theorem 4.1 (Subject Reduction)

The two following lemmas are standard and say that typing is preserved by structural congruence and by substitutions. In the case of substitutions, subtyping may be applied without compromising well-typedness.

Lemma A.1. *Let* $\tilde{\mathscr{E}};\Gamma \vdash P$ *and* $P \equiv Q$. *Then* $\tilde{\mathscr{E}};\Gamma \vdash Q$.

Proof. By case analysis on the derivation of $P \equiv Q$. □

Lemma A.2 (substitution)**.** *If* $\tilde{\mathscr{E}};\Gamma,u:t \vdash P$ *and* $v \notin \mathsf{dom}(\Gamma) \cup \mathsf{bn}(P)$ *and* $s \leqslant t$, *then* $\tilde{\mathscr{E}};\Gamma,v:s \vdash P\{v/u\}$.

Proof. For notational simplicity we prove the result when $u = x$ and $v = a$. We proceed by induction on the derivation of $\tilde{\mathscr{E}};\Gamma,x:t \vdash P$ and by cases on the last rule applied. We only prove a few interesting cases.

[T-CLOSE] In this case $P = \texttt{close}(u).Q$ and $\Gamma,x:t = \Gamma',u:\texttt{end}$ and $\tilde{\mathscr{E}};\Gamma' \vdash Q$.

If $x \in \mathsf{dom}(\Gamma')$, then $\Gamma' = \Gamma'',x:t$ and $\Gamma = \Gamma'',x:t,u:\texttt{end}$. By induction hypothesis we deduce $\tilde{\mathscr{E}};\Gamma'',a:s \vdash Q\{a/x\}$. Since we know that $a \neq u$, from rule [T-CLOSE] we conclude $\tilde{\mathscr{E}};\Gamma,a:s \vdash \texttt{close}(u).Q\{a/x\}$.

If $x = u$, then $\Gamma = \Gamma', t = s = \mathtt{end}$, and $Q = Q\{a/x\}$ because x does not occur free in Q. From rule [T-CLOSE] we conclude $\tilde{\mathscr{E}}; \Gamma, a : \mathtt{end} \vdash \mathtt{close}(a).Q\{a/x\}$.

[T-SEND] In this case $\Gamma, x : t = \Gamma', u : \{!\mathtt{m}_i(S_i).T_i\}_{i\in I}, v : S'$ and $P = u!\mathtt{m}_k(v).Q$ where $k \in I$ and $S' \leqslant S_k$ and $\|S'\| < \infty$ and $\tilde{\mathscr{E}}; \Gamma', u : T_k \vdash Q$.

If $x \in \mathsf{dom}(\Gamma')$, then $\Gamma' = \Gamma'', x : t$ and $\Gamma = \Gamma'', u : \{!\mathtt{m}_i(S_i).T_i\}_{i\in I}, v : S'$. From $\tilde{\mathscr{E}}; \Gamma', u : T_k \vdash Q$ and by induction hypothesis we obtain $\tilde{\mathscr{E}}; \Gamma'', a : s, u : T_k \vdash Q\{a/x\}$. Since $a \notin \mathsf{dom}(\Gamma)$ we know that $a \notin \{u, v\}$ and from rule [T-SEND] we conclude $\tilde{\mathscr{E}}; \Gamma, a : s \vdash u!\mathtt{m}_k(v).Q\{a/x\}$.

If $x = u$, then $\Gamma = \Gamma', v : S'$ and $t = \{!\mathtt{m}_i(S_i).T_i\}_{i\in I}$ and $s = \{!\mathtt{m}_i(S'_i).T'_i\}_{i\in I\cup J}$ and $S_i \leqslant S'_i$ and $T'_i \leqslant T_i$ for every $i \in I$. Then $S' \leqslant S_k \leqslant S'_k$. From $\tilde{\mathscr{E}}; \Gamma', u : T_k \vdash Q$ and the induction hypothesis we obtain $\tilde{\mathscr{E}}; \Gamma', a : T'_k \vdash Q\{a/x\}$. We conclude $\tilde{\mathscr{E}}; \Gamma', a : \{!\mathtt{m}_i(S'_i).T'_i\}_{i\in I\cup J}, v : S' \vdash a!\mathtt{m}_k(v).Q\{a/x\}$ with an application of rule [T-SEND].

If $x = v$, then $t = S'$ and $x \notin \mathsf{dom}(\Gamma') \cup \{u\}$ and $Q = Q\{a/x\}$. From Proposition 4.4 we deduce $\|s\| \leq \|t\| < \infty$. We conclude with an application of rule [T-SEND].

[T-TRY] In this case $\Gamma, x : t = \Gamma', \{u_i : \{\mathtt{e}_j : S_{ij}\}_{j\in J}[\![T_i\}_{i\in I}$ and $P = \mathtt{try}(\{u_i\}_{i\in I})\ \{\mathtt{e}_j : Q_j\}_{j\in J} Q$ and $\tilde{\mathscr{E}}; \{\mathtt{e}_j\}_{j\in J}; [\Gamma'], \{u_i : T_i\}_{i\in I} \vdash Q$ and $\tilde{\mathscr{E}}; \Gamma', \{u_i : S_{ij}\}_{i\in I} \vdash Q_j$ for every $j \in J$.

If $x \in \mathsf{dom}(\Gamma')$, then $\Gamma' = \Gamma'', x : t$ for some Γ''. From the hypothesis $s \leqslant t$ we deduce $[s] \leqslant [t]$. From $\tilde{\mathscr{E}}; \{\mathtt{e}_j\}_{j\in J}; [\Gamma'], \{u_i : T_i\}_{i\in I} \vdash Q$ and by induction hypothesis we deduce $\tilde{\mathscr{E}}; \{\mathtt{e}_j\}_{j\in J}; [\Gamma''], a : [s], \{u_i : T_i\}_{i\in I} \vdash Q\{a/x\}$. From $\tilde{\mathscr{E}}; \Gamma', \{u_i : S_{ij}\}_{i\in I} \vdash Q_j$ and by induction hypothesis we deduce $\tilde{\mathscr{E}}; \Gamma'', a : s, \{u_i : S_{ij}\}_{i\in I} \vdash Q_j\{a/x\}$ for every $j \in J$. We conclude with an application of rule [T-TRY].

If $x = u_k$ for some $k \in I$, then $t = \{\mathtt{e}_j : S_{kj}\}_{j\in J}[\![T_k$ and $s = \{\mathtt{e}_j : S'_{kj}\}_{j\in J}[\![T'_k$ where $S'_{kj} \leqslant S_{kj}$ for every $j \in J$ and $T'_k \leqslant T_k$. By induction hypothesis we deduce $\tilde{\mathscr{E}}; \{\mathtt{e}_j\}_{j\in J}; [\Gamma'], \{u_i : T_i\}_{i\in I\setminus\{k\}}, a : T'_k \vdash Q\{a/x\}$ and $\tilde{\mathscr{E}}; \Gamma', \{u_i : S_{ij}\}_{i\in I\setminus\{k\}}, a : S'_{kj} \vdash Q_j\{a/x\}$ for every $j \in J$. We conclude with an application of rule [T-TRY]. □

The next lemma connects well-typed configurations to well-typed heaps and shows the irrelevance of the first and second components Γ_0 and Γ_R in typing processes.

Lemma A.3. *Let* $\tilde{\mathscr{E}}; \Gamma_0; \Gamma_R; \Gamma \vdash \mu \triangleright P$. *Then:*

(1) $\Gamma_0; \Gamma_R, \Gamma \Vdash \mu$;

(2) $\Gamma'_0; \Gamma'_R, \Gamma \Vdash \mu'$ *implies* $\tilde{\mathscr{E}}; \Gamma'_0; \Gamma'_R; \Gamma \vdash \mu' \triangleright P$.

Proof. By induction on the derivation of $\Gamma_0;\Gamma_R;\Gamma \vdash \mu \fatsemi P$ and by cases on the last rule applied. The only interesting case is [T-RUNNING TRANSACTION], from which we deduce:

- $P = \langle \{a_i\}_{i\in I}, B, \{\mathsf{e}_j : R_j\}_{j\in J} Q\rangle$;
- $\Gamma = \Gamma_1, \{a_i : \{\mathsf{e}_j : S_{ij}\}_{j\in J} T_i\}_{i\in I}, \Gamma_2$;
- μ-balanced$(\{a_i : S_{ij}\}_{i\in I})$ for every $j \in J$;
- $\tilde{\mathscr{E}}\{\mathsf{e}_j\}_{j\in J};\Gamma_0;\Gamma_R;[\Gamma_1],\{a_i : T_i\}_{i\in I},\Gamma_2 \vdash \mu \fatsemi Q$.

Regarding (1), from $\tilde{\mathscr{E}}\{\mathsf{e}_j\}_{j\in J};\Gamma_0;\Gamma_R;[\Gamma_1],\{a_i : T_i\}_{i\in I},\Gamma_2 \vdash \mu \fatsemi Q$ by induction hypothesis we obtain $\Gamma_0;\Gamma_R,[\Gamma_1],\{a_i : T_i\}_{i\in I},\Gamma_2 \Vdash \mu$ and then from the definition of function tail and μ-balanced$(\{a_i : S_{ij}\}_{i\in I})$ for $j \in J$ we conclude $\Gamma_0;\Gamma_R,\Gamma \Vdash \mu$.

Regarding (2), from $\Gamma'_0;\Gamma'_R,\Gamma \Vdash \mu'$ and the definition of function tail we deduce $\Gamma'_0;\Gamma'_R,[\Gamma_1],\{a_i : T_i\}_{i\in I},\Gamma_2 \Vdash \mu'$ and then from $\tilde{\mathscr{E}}\{\mathsf{e}_j\}_{j\in J};\Gamma_0;\Gamma_R;[\Gamma_1],\{a_i : T_i\}_{i\in I},\Gamma_2 \vdash \mu \fatsemi Q$ by induction hypothesis we obtain $\tilde{\mathscr{E}}\{\mathsf{e}_j\}_{j\in J};\Gamma'_0;\Gamma'_R;[\Gamma_1],\{a_i : T_i\}_{i\in I},\Gamma_2 \vdash \mu' \fatsemi Q$. We conclude with an application of rule [T-RUNNING TRANSACTION]. □

The next lemma shows that the rule [T-RUNNING PARALLEL] for configurations subsumes the rule [T-PARALLEL] for processes. It is used for simplifying some cases in the proof of subject reduction (Theorem 4.1).

Lemma A.4. *If* $\tilde{\mathscr{E}};\Gamma_0;\Gamma_R;\Gamma \vdash \mu \fatsemi P_1 \mid P_2$ *is derivable using* [T-PARALLEL] *and* [T-RUNNING PROCESS]*, then it is also derivable using* [T-RUNNING PROCESS] *and* [T-RUNNING PARALLEL].

Proof. From $\tilde{\mathscr{E}};\Gamma_0;\Gamma_R;\Gamma \vdash \mu \fatsemi P_1 \mid P_2$ and rule [T-RUNNING PROCESS] we obtain (H.1) $\tilde{\mathscr{E}};\Gamma \vdash P_1 \mid P_2$ and (H.2) $\Gamma_0;\Gamma_R,\Gamma \Vdash \mu$. From (H.1) and rule [T-PARALLEL] we obtain (T.1) $\Gamma = \Gamma_1,\Gamma_2$ and (P.*i*) $\tilde{\mathscr{E}};\Gamma_i \vdash P_i$ for $i \in \{1,2\}$. From (H.2), (T.1), (P.*i*) and rule [T-RUNNING PROCESS] we obtain $\tilde{\mathscr{E}};\Gamma_0;\Gamma_R,\Gamma_{3-i};\Gamma_i \vdash \mu \fatsemi P_i$ for $i \in \{1,2\}$. We conclude with an application of rule [T-RUNNING PARALLEL]. □

When a new channel is allocated in the heap, it always comes as a pair of peer endpoints. This easy property is formalized thus:

Proposition A.1. *If* $\mu \fatsemi P \to \mu' \fatsemi P'$ *then* μ'-balanced$(\mathsf{dom}(\mu') \setminus \mathsf{dom}(\mu))$.

Proof. Simple induction on the reduction that occurs. □

The next result is the complete version of subject reduction proving that reductions preserve well typedness and showing the relationship between the contexts used for typing the two configurations involved. In particular, item (1) below says that reductions preserve well typedness; item (2) says that the rank of the type of endpoints are preserved by reductions, and that (de)allocated endpoints have a type with null rank; finally, item (3) formally expresses the concept of process isolation, saying that any portion of the heap that is not reachable by the process being reduced is not affected by the reduction. In this theorem and in its proof, we write $\mathsf{unsealed}(\Gamma)$ if all types in the range of Γ are unsealed. We also write $\bigsqcup_{i\in I}\Gamma_i$ for the disjoint union of the contexts Γ_i.

Theorem 4.1. *Let* $\tilde{\mathscr{E}};\Gamma_0;\Gamma_R;[\Gamma_S],\Gamma \vdash \mu \fatsemi P$ *where* $\mathsf{unsealed}(\Gamma)$ *and* $\mu \fatsemi P \to \mu' \fatsemi P'$. *Then there exist* Γ_0' *and* Γ' *such that:*

(1) $\tilde{\mathscr{E}};\Gamma_0';\Gamma_R;[\Gamma_S],\Gamma' \vdash \mu' \fatsemi P'$, *and*

(2) $\mathsf{unsealed}(\Gamma')$ *and for every* $a \in \mathsf{dom}(\Gamma)\cap\mathsf{dom}(\Gamma')$ *we have* $\mathsf{rank}(\Gamma(a)) = \mathsf{rank}(\Gamma'(a))$ *and for every* $a \in \mathsf{dom}(\Gamma)\setminus\mathsf{dom}(\Gamma')$ *we have* $\mathsf{rank}(\Gamma(a)) = 0$ *and for every* $a \in \mathsf{dom}(\Gamma')\setminus\mathsf{dom}(\Gamma)$ *we have* $\mathsf{rank}(\Gamma'(a)) = 0$, *and*

(3) for every $\Gamma_I \subseteq \Gamma_R,[\Gamma_S]$ *such that* $\mu\text{-}\mathsf{balanced}(\mu\text{-}\mathsf{reach}(\mathsf{dom}(\Gamma_I,\Gamma)))$ *we have*

$$\mu\text{-}\mathsf{reach}(\mathsf{dom}(\Gamma_R,\Gamma_S)\setminus\mathsf{dom}(\Gamma_I)) = \mu'\text{-}\mathsf{reach}(\mathsf{dom}(\Gamma_R,\Gamma_S)\setminus\mathsf{dom}(\Gamma_I)).$$

Proof. By induction on the derivation of $\mu \fatsemi P \to \mu' \fatsemi P'$ and by cases on the last rule applied. We omit trivial and symmetric cases.

[R-OPEN] Then $P = \mathtt{open}(a,b).P'$ and $\mu' = \mu, a \mapsto [b,\varepsilon], b \mapsto [a,\varepsilon]$. From rule [T-RUNNING PROCESS] we obtain:

- (H.1) $\tilde{\mathscr{E}};[\Gamma_S],\Gamma \vdash \mathtt{open}(a,b).P'$;
- (H.2) $\Gamma_0;\Gamma_R,[\Gamma_S],\Gamma \Vdash \mu$.

From (H.1) and rule [T-OPEN] we obtain:

- $\vdash T : 0$;
- (C.1) $\tilde{\mathscr{E}};[\Gamma_S],\Gamma,a : T, b : \overline{T} \vdash P'$.

Let $\Gamma_0' = \Gamma_0$ and $\Gamma' = \Gamma, a : T, b : \overline{T}$. The proof of (C.2) $\Gamma_0'; \Gamma_R, [\Gamma_S], \Gamma' \Vdash \mu'$ is trivial. From (C.1), (C.2) and [T-RUNNING PROCESS] we obtain (1). We conclude by noting that items (2) and (3) hold trivially.

[R-CLOSE] In this case $P = \texttt{close}(a).P'$ and $\mu = \mu', a \mapsto [b, \mathfrak{Q}]$. From rule [T-RUNNING PROCESS] we obtain:

- (H.1) $\tilde{\mathscr{E}}; [\Gamma_S], \Gamma \vdash \texttt{close}(a).P'$;
- (H.2) $\Gamma_0; \Gamma_R, [\Gamma_S], \Gamma \Vdash \mu', a \mapsto [b, \mathfrak{Q}]$.

From the hypothesis (H.1) and rule [T-CLOSE] we obtain:

- (L.1) $\Gamma = \Gamma', a : \texttt{end}$;
- (C.1) $\tilde{\mathscr{E}}; [\Gamma_S], \Gamma' \vdash P'$.

Let $\Gamma_0' = \Gamma_0$. We only have to show that (C.2) $\Gamma_0'; \Gamma_R, [\Gamma_S], \Gamma' \Vdash \mu'$ and the only interesting case in Definition 4.4 is item 3. The relation $\mathsf{dom}(\mu') = \mathsf{dom}(\Gamma_0', \Gamma_R, [\Gamma_S], \Gamma')$ is obvious. We need to prove $\mathsf{dom}(\Gamma_0', \Gamma_R, [\Gamma_S], \Gamma') = \mu'\text{-}\mathsf{reach}(\mathsf{dom}(\Gamma_R', [\Gamma_S], \Gamma'))$. First we show that $\mathfrak{Q}$ is empty. Suppose by contradiction that this is not the case. Then the endpoint type associated with a before the reduction occurs must begin with an external choice or running transaction, which contradicts (L.1). So we obtain

$$\begin{array}{lcll} \mu'\text{-}\mathsf{reach}(\mathsf{dom}(\Gamma_R, [\Gamma_S], \Gamma')) & = & \mu\text{-}\mathsf{reach}(\mathsf{dom}(\Gamma_R, [\Gamma_S], \Gamma)) \setminus \{a\} & \\ & = & \mathsf{dom}(\Gamma_0, \Gamma_R, [\Gamma_S], \Gamma) \setminus \{a\} & \text{(H.2)} \\ & = & \mathsf{dom}(\Gamma_0', \Gamma_R, \Gamma') & \end{array}$$

From (C.1), (C.2) and [T-RUNNING PROCESS] we conclude (1). Item (2) is obvious while item (3) holds because $\mu\text{-}\mathsf{reach}(\mathsf{dom}(\Gamma_R, \Gamma_S)) = \mu'\text{-}\mathsf{reach}(\mathsf{dom}(\Gamma_R, \Gamma_S))$.

[R-PARALLEL] In this case $P = P_1 \mid P_2$ and $\mu \triangleright P_1 \to \mu' \triangleright P_1'$ and $P' = P_1' \mid P_2$. By Lemma A.4 we can assume that $\tilde{\mathscr{E}}; \Gamma_0; \Gamma_R, [\Gamma_S], \Gamma \vdash \mu \triangleright P$ was derived by an application of rule [T-RUNNING PARALLEL]. Then:

- $\Gamma = \Gamma_1, \Gamma_2$ and $\Gamma_S = \Gamma_{S1}, \Gamma_{S2}$;
- (P.i) $\tilde{\mathscr{E}}; \Gamma_0; \Gamma_R, [\Gamma_{S3-i}], \Gamma_{3-i}; [\Gamma_{Si}], \Gamma_i \vdash \mu \triangleright P_i$ for $i \in \{1, 2\}$.

From (P.1) by induction hypothesis we deduce that there exist Γ_0' and Γ_1' such that:

(1') $\tilde{\mathscr{E}}; \Gamma_0'; \Gamma_R, [\Gamma_{S2}], \Gamma_2; [\Gamma_{S1}], \Gamma_1' \vdash \mu' \wr P_1'$, and

(2') $\mathsf{unsealed}(\Gamma_1')$ and for every $a \in \mathsf{dom}(\Gamma_1) \cap \mathsf{dom}(\Gamma_1')$ we have $\mathsf{rank}(\Gamma_1(a)) = \mathsf{rank}(\Gamma_1'(a))$ and for every $a \in \mathsf{dom}(\Gamma_1) \setminus \mathsf{dom}(\Gamma_1')$ we have $\mathsf{rank}(\Gamma_1(a)) = 0$ and for every $a \in \mathsf{dom}(\Gamma_1') \setminus \mathsf{dom}(\Gamma_1)$ we have $\mathsf{rank}(\Gamma_1'(a)) = 0$, and

(3') for every $\Gamma_I \subseteq \Gamma_R, [\Gamma_S], \Gamma_2$ such that $\mu\text{-}\mathsf{balanced}(\mu\text{-}\mathsf{reach}(\mathsf{dom}(\Gamma_I, \Gamma_1)))$ we have

$$\mu\text{-}\mathsf{reach}(\mathsf{dom}(\Gamma_R, \Gamma_S, \Gamma_2) \setminus \mathsf{dom}(\Gamma_I)) = \mu'\text{-}\mathsf{reach}(\mathsf{dom}(\Gamma_R, \Gamma_S, \Gamma_2) \setminus \mathsf{dom}(\Gamma_I)).$$

Let $\Gamma' = \Gamma_1', \Gamma_2$. From (1') and Lemma A.3(1) we obtain (N.1) $\Gamma_0'; \Gamma_R, [\Gamma_S], \Gamma' \Vdash \mu'$. From (P.2), (N.1), and Lemma A.3(2) we deduce (P.2') $\tilde{\mathscr{E}}; \Gamma_0'; \Gamma_R, [\Gamma_{S1}], \Gamma_1'; [\Gamma_{S2}], \Gamma_2 \vdash \mu' \wr P_2$. From (1'), (P.2'), and rule [T-RUNNING PARALLEL] we conclude (1). Regarding (2), just notice that $\mathsf{unsealed}(\Gamma')$. Regarding (3), let $\Gamma_J \subseteq \Gamma_R, [\Gamma_S]$ be such that $\mu\text{-}\mathsf{balanced}(\mu\text{-}\mathsf{reach}(\mathsf{dom}(\Gamma_J, \Gamma)))$. Take $\Gamma_I = \Gamma_J, \Gamma_2$ and observe that $\Gamma_I \subseteq \Gamma_R, [\Gamma_S], \Gamma_2$ and $\mu\text{-}\mathsf{balanced}(\mu\text{-}\mathsf{reach}(\mathsf{dom}(\Gamma_I, \Gamma_1)))$. From (3') we are able to deduce $\mu\text{-}\mathsf{reach}(\mathsf{dom}(\Gamma_R, \Gamma_S, \Gamma_2) \setminus \mathsf{dom}(\Gamma_I)) = \mu'\text{-}\mathsf{reach}(\mathsf{dom}(\Gamma_R, \Gamma_S, \Gamma_2) \setminus \mathsf{dom}(\Gamma_I))$ and we conclude (3) by observing that $\mathsf{dom}(\Gamma_R, \Gamma_S) \setminus \mathsf{dom}(\Gamma_J) = \mathsf{dom}(\Gamma_R, \Gamma_S, \Gamma_2) \setminus \mathsf{dom}(\Gamma_I)$.

[R-SEND] In this case $P = a!\mathtt{m}(c).P'$ and $\mu = \mu'', a \mapsto [b, \mathfrak{Q}], b \mapsto [a, \mathfrak{Q}']$ and $\mu' = \mu'', a \mapsto [b, \mathfrak{Q}], b \mapsto [a, \mathfrak{Q}' :: \mathtt{m}(c)]$. From [T-RUNNING PROCESS] we obtain:

- (H.1) $\tilde{\mathscr{E}}; [\Gamma_S], \Gamma \vdash a!\mathtt{m}(c).P'$;
- (H.2) $\Gamma_0; \Gamma_R, [\Gamma_S], \Gamma \Vdash \mu'', a \mapsto [b, \mathfrak{Q}], b \mapsto [a, \mathfrak{Q}']$.

From (H.1) and rule [T-SEND] we deduce:

- (L.1) $\Gamma = \Gamma'', a : \{!\mathtt{m}_i(S_i).T_i\}_{i \in I}, c : S$;
- $\mathtt{m} = \mathtt{m}_k$ for some $k \in I$;
- $S \leqslant S_k$ and $\|S\| < \infty$;
- (C.1) $\tilde{\mathscr{E}}; [\Gamma_S], \Gamma'', a : T_k \vdash P'$.

Let $\Gamma_0' = \Gamma_0, c : S_k$ and $\Gamma' = \Gamma'', a : T_k$. We show (C.2) $\Gamma_0'; \Gamma_R, [\Gamma_S], \Gamma' \Vdash \mu'$ by proving the items of Definition 4.4 in order.

1. We only need to show that $\mathfrak{Q}$ is empty. Suppose by contradiction that this is not the case. Then the endpoint type associated with a before the reduction must begin with an external choice or running transaction, which contradicts (L.1).

2. Let $\mathfrak{Q}' = \mathtt{m}_1(c_1) :: \cdots :: \mathtt{m}_p(c_p)$. From hypothesis (H.2) we deduce $\Gamma_0, \Gamma_R, [\Gamma_S], \Gamma \vdash b : T_b$ and $\Gamma_0, \Gamma_R, [\Gamma_S], \Gamma \vdash c_i : S'_i$ for $1 \leq i \leq p$ where

$$\mathsf{tail}(T_b, \mathtt{m}_1(S'_1) \cdots \mathtt{m}_p(S'_p)) = \overline{\{!\mathtt{m}_i(S_i).T_i\}_{i \in I}} = \{?\mathtt{m}_i(S_i).\overline{T_i}\}_{i \in I}$$

   and from $S \leqslant S_k$ we conclude $\overline{T_k} = \mathsf{tail}(T_b, \mathtt{m}_1(S'_1) \cdots \mathtt{m}_p(S'_p)\mathtt{m}(S))$.

3. From hypothesis (H.2) we have $\mathsf{dom}(\mu) = \mathsf{dom}(\Gamma_0, \Gamma_R, \Gamma)$ and for every $a' \in \mathsf{dom}(\mu)$ there exists $b' \in \mathsf{dom}(\Gamma_R, \Gamma_S, \Gamma)$ such that $a' \preccurlyeq_\mu b'$. Clearly $\mathsf{dom}(\mu') = \mathsf{dom}(\Gamma'_0, \Gamma_R, \Gamma_S, \Gamma')$ since $\mathsf{dom}(\mu') = \mathsf{dom}(\mu)$ and $\mathsf{dom}(\Gamma'_0) \cup \mathsf{dom}(\Gamma') = \mathsf{dom}(\Gamma_0) \cup \mathsf{dom}(\Gamma)$. Let $b \preccurlyeq_\mu b_0$ and $\Gamma_R, \Gamma_S, \Gamma \vdash b_0 : T_0$. We have $c \prec_{\mu'} b \preccurlyeq_{\mu'} b_0$, namely $c \preccurlyeq_{\mu'} b_0$. Now

$$\|S\| \leq \|S_k\| < \|\mathsf{tail}(T_b, \mathtt{m}_1(S'_1) \cdots \mathtt{m}_p(S'_p))\| \leq \|T_b\| \leq \|T_0\|$$

   and $\|S\| < \infty$, therefore $c \neq b_0$. We conclude $b_0 \in \mathsf{dom}(\Gamma_R, \Gamma_S, \Gamma')$.

4. Immediate from hypothesis (H.2).

Item (2) holds trivially. Regarding item (3), let $\Gamma_I \subseteq \Gamma_R, [\Gamma_S]$ be such that μ-balanced(μ-reach($\mathsf{dom}(\Gamma_I, \Gamma)$)). From $a \in \mathsf{dom}(\Gamma)$ we are able to deduce that $b \in \mu$-reach($\mathsf{dom}(\Gamma_I, \Gamma)$) and $c \preccurlyeq_{\mu'} b$, therefore μ-reach($\mathsf{dom}(\Gamma_R, \Gamma_S) \setminus \mathsf{dom}(\Gamma_I)$) = μ'-reach($\mathsf{dom}(\Gamma_R, \Gamma_S) \setminus \mathsf{dom}(\Gamma_I)$).

[R-RECEIVE] In this case $P = \sum_{i \in I} a?\mathtt{m}_i(x_i).P_i$ and $\mu = \mu'', a \mapsto [b, \mathtt{m}(c) :: \mathfrak{Q}]$ where $\mathfrak{Q} = \mathtt{m}_1(c_1) :: \cdots :: \mathtt{m}_p(c_p)$ and $\mathtt{m} = \mathtt{m}_k$ for some $k \in I$ and $P' = P_k\{c/x_k\}$ and $\mu' = \mu'', a \mapsto [b, \mathfrak{Q}]$. From rule [T-RUNNING PROCESS] we obtain:

- (H.1) $\tilde{\mathscr{E}}; [\Gamma_S], \Gamma \vdash \sum_{i \in I} a?\mathtt{m}_i(x_i).P_i$;
- (H.2) $\Gamma_0; \Gamma_R, [\Gamma_S], \Gamma \Vdash \mu$.

From (H.1) and rule [T-RECEIVE] we obtain:

- $\Gamma = \Gamma'', a : \{?\mathtt{m}_i(S_i).T_i\}_{i \in J}$ with $J \subseteq I$;
- (N.1) $\tilde{\mathscr{E}}; [\Gamma_S], \Gamma'', a : T_k, x_k : S_k \vdash P_k$.

From (H.2) we deduce $\Gamma_0 = \Gamma_0', c : S$ where $S \leqslant S_k$ and $k \in J$. Let $\Gamma' = \Gamma'', a : T_k, c : S$. From (N.1) and Lemma A.2 we deduce (C.1) $\tilde{\mathscr{E}}; [\Gamma_S], \Gamma' \vdash P_k\{c/x_k\}$. Now we only have to show (C.2) $\Gamma_0'; \Gamma_R, [\Gamma_S], \Gamma' \Vdash \mu'$ and we do it by proving the items of Definition 4.4 in order.

1. Since the queue associated with a is not empty in μ, the queue associated with its peer endpoint b must be empty. The reduction does not change the queue associated with b, therefore condition (1) of Definition 4.4 is satisfied.

2. From (H.2) we deduce $\Gamma_0, \Gamma_R, [\Gamma_S], \Gamma \vdash b : T_b$ and

   $$\overline{T_b} = \mathsf{tail}(\{?\mathtt{m}_i(S_i).T_i\}_{i\in J}, \mathtt{m}(S)\mathtt{m}_1(S_1')\cdots\mathtt{m}_p(S_p')) = \mathsf{tail}(T_k, \mathtt{m}_1(S_1')\cdots\mathtt{m}_p(S_p'))$$

   where $\Gamma_0, \Gamma_R, [\Gamma_S], \Gamma \vdash c_i : S_i'$ for $1 \leq i \leq p$.

3. Straightforward by definition of Γ_0' and Γ'.

4. Immediate from (H.2).

Therefore, from (C.1), (C.2), and rule [T-RUNNING PROCESS] we conclude $\tilde{\mathscr{E}}; \Gamma_0'; \Gamma_R, [\Gamma_S], \Gamma' \vdash \mu' \wr P'$. Regarding (2), observe from (H.2) and condition (2) of Definition 4.4 that $\mathsf{rank}(S_k) = 0$. Regarding (3), it suffices to observe that the only region of the heap that changes is the queue associated with a and that $a \notin \mathsf{dom}(\Gamma_R, \Gamma_S)$.

[R-START TRANSACTION] In this case $P = \prod_{i\in I} \mathtt{try}(A_i)\ \{\mathtt{e}_j : Q_{ij}\}_{j\in J} P_i$ where μ-$\mathsf{balanced}(\bigcup_{i\in I} A_i)$ and $P' = \langle \bigcup_{i\in I} A_i, \emptyset, \{\mathtt{e}_j : \prod_{i\in I} Q_{ij}\}_{j\in J} \prod_{i\in I} P_i\rangle$ and $\mu' = \mu$. According to Lemma A.4 we can assume that $\tilde{\mathscr{E}}; \Gamma_0; \Gamma_R; [\Gamma_S], \Gamma \vdash \mu \wr P$ was derived by rule [T-RUNNING PARALLEL]. Then:

- (L.1) $\Gamma_S = \bigsqcup_{i\in I} \Gamma_{Si}$ and $\Gamma = \bigsqcup_{i\in I} \Gamma_i$;
- (P.i) $\tilde{\mathscr{E}}; \Gamma_0; \Gamma_R, \bigsqcup_{j\in I\setminus\{i\}} [\Gamma_{Sj}], \bigsqcup_{j\in I\setminus\{i\}} \Gamma_j; [\Gamma_{Si}], \Gamma_i \vdash \mu \wr \mathtt{try}(A_i)\ \{\mathtt{e}_j : Q_{ij}\}_{j\in J} P_i$ for every $i \in I$.

From (L.1), (P.i) and rule [T-RUNNING PROCESS] we obtain:

- (H.1) $\tilde{\mathscr{E}}; [\Gamma_{Si}], \Gamma_i \vdash \mathtt{try}(A_i)\ \{\mathtt{e}_j : Q_{ij}\}_{j\in J} P_i$ where $\mathsf{unsealed}(\Gamma_i)$ for every $i \in I$;
- (H.2) $\Gamma_0; \Gamma_R, [\Gamma_S], \Gamma \Vdash \mu$.

From (H.1) and rule [T-TRY] we obtain, for every $i \in I$:

- (L.2) $\Gamma_i = \Gamma_i', \{a : \{\mathsf{e}_j : S_{aj}\}_{j\in J}[\![T_a\}_{a\in A_i}$;
- $\tilde{\mathscr{E}}\{\mathsf{e}_j\}_{j\in J}; [[\Gamma_{Si}], \Gamma_i'], \{a : T_a\}_{a\in A_i} \vdash P_i$;
- $\tilde{\mathscr{E}}; [\Gamma_{Si}], \Gamma_i', \{a : S_{aj}\}_{a\in A_i} \vdash Q_{ij}$ for every $j \in J$.

By rule [T-PARALLEL] we derive:

- (P.1) $\tilde{\mathscr{E}}\{\mathsf{e}_j\}_{j\in J}; [[\Gamma_S], \bigsqcup_{i\in I}\Gamma_i'], \{a : T_a\}_{i\in I, a\in A_i} \vdash \prod_{i\in I} P_i$;
- (P.2) $\tilde{\mathscr{E}}; [\Gamma_S], \bigsqcup_{i\in I}\Gamma_i', \{a : S_{aj}\}_{i\in I, a\in A_i} \vdash \prod_{i\in I} Q_{ij}$ for every $j \in J$.

From (L.2) and the definition of tail it is easy to see that the queue associated with a is empty for every $i \in I$ and $a \in A_i$. Hence, from (H.2), (L.1), and (L.2) and the fact that μ-balanced$(\bigcup_{i\in I} A_i)$ we have μ-balanced$(\{a : S_{aj}\}_{i\in I, a\in A_i})$ for every $j \in J$ and $\bigcup_{i\in I} A_i = \mu$-reach$(\bigcup_{i\in I} A_i)$. Also, from (H.2) we have

- (H.2') $\Gamma_0; \Gamma_R, [[\Gamma_S], \bigsqcup_{i\in I}\Gamma_i'], \{a : T_a\}_{i\in I, a\in A_i} \Vdash \mu$.

From (H.2'), (P.1), and rule [T-RUNNING PROCESS] we obtain:

- (T.1) $\tilde{\mathscr{E}}\{\mathsf{e}_j\}_{j\in J}; \Gamma_0; \Gamma_R; [[\Gamma_S], \bigsqcup_{i\in I}\Gamma_i'], \{a : T_a\}_{i\in I, a\in A_i} \vdash \mu \fatsemi \prod_{i\in I} P_i$.

We conclude (1) with an application of rule [T-RUNNING TRANSACTION], (T.1), (P.2), and the facts proven above by taking $\Gamma_0' = \Gamma_0$ and $\Gamma' = \bigsqcup_{i\in I}\Gamma_i', \{a : \{\mathsf{e}_j : S_{aj}\}_{j\in J} T_a\}_{i\in I, a\in A_i}$. Item (2) is trivial and (3) holds since $\mu' = \mu$.

[R-END TRANSACTION] Then $P = \langle A, B, \{\mathsf{e}_j : Q_j\}_{j\in J} \prod_{i\in I} \mathtt{commit}(A_i).P_i\rangle$ and $P' = \prod_{i\in I} P_i$ and $\mu' = \mu$. From rule [T-RUNNING TRANSACTION] we obtain:

- $\Gamma = \Gamma_1, \{a : \{\mathsf{e}_j : S_{aj}\}_{j\in J} T_a\}_{a\in A}, \Gamma_2$;
- (T.1) $\tilde{\mathscr{E}}\{\mathsf{e}_j\}_{j\in J}; \Gamma_0; \Gamma_R; [[\Gamma_S], \Gamma_1], \{a : T_a\}_{a\in A}, \Gamma_2 \vdash \mu \fatsemi \prod_{i\in I} \mathtt{commit}(A_i).P_i$;
- μ-balanced$(\{a : S_{aj}\}_{a\in A})$ for every $j \in J$;
- μ-balanced(B);
- local(Γ_2);
- $A \cup B = \mu$-reach$(A \cup \mathsf{dom}(\Gamma_2))$.

From (T.1), [T-RUNNING PARALLEL], and [T-RUNNING PROCESS] we deduce:

- $\Gamma_S = \bigsqcup_{i\in I}\Gamma_{Si}$ and $\Gamma_1 = \bigsqcup_{i\in I}\Gamma_{1i}$ and $\{a:T_a\}_{a\in A} = \bigsqcup_{i\in I}\{a:T_a\}_{a\in B_i}$ and $\Gamma_2 = \bigsqcup_{i\in I}\Gamma_{2i}$ where $\mathsf{rank}(\Gamma_{2i}) = 0$ for every $i \in I$;
- $\tilde{\mathscr{E}}\{\mathsf{e}_j\}_{j\in J}; [[\Gamma_{Si}], \Gamma_{1i}], \{a:T_a\}_{a\in B_i}, \Gamma_{2i} \vdash \texttt{commit}(A_i).P_i$ for every $i \in I$.

From rule [T-COMMIT] we deduce:

- $B_i = A_i$;
- $T_a = \rrbracket T'_a$ for every $i \in I$ and $a \in A_i$;
- $\tilde{\mathscr{E}}; [\Gamma_{Si}], \Gamma_{1i}, \{a:T'_a\}_{a\in A_i}, \Gamma_{2i} \vdash P_i$ for every $i \in I$.

From rule [T-PARALLEL] we deduce (H.1) $\tilde{\mathscr{E}}; [\Gamma_S], \Gamma_1, \{a:T'_a\}_{i\in I, a\in A_i}, \Gamma_2 \vdash P'$. Let $\Gamma'_0 = \Gamma_0$ and $\Gamma' = \Gamma_1, \{a:T'_a\}_{a\in A}, \Gamma_2$. From $\tilde{\mathscr{E}}; \Gamma_0; \Gamma_R; [\Gamma_S], \Gamma \vdash \mu \fatsemi P$ and Lemma A.3(1) we obtain (H.2) $\Gamma_0; \Gamma_R, [\Gamma_S], \Gamma \Vdash \mu$. From (2) we deduce that the queues associated with the pointers $a \in A$ are empty, because Γ includes $\{a : \{\mathsf{e}_j : S_{aj}\}_{j\in J}\rrbracket T'_a\}_{a\in A}$. Hence we deduce $\Gamma_0; \Gamma_R, [\Gamma_S], \Gamma' \Vdash \mu'$. From (H.1) and rule [T-RUNNING PROCESS] we conclude (1). We observe that (2) holds since $\mathsf{rank}(\{\mathsf{e}_j : S_{aj}\}_{j\in J} T_a) = \mathsf{rank}(T'_a)$ for $a \in A$ and (3) holds trivially since the heap has not changed.

[R-RUN TRANSACTION] In this case: $P = \langle A, B, \{\mathsf{e}_j : R_j\}_{j\in J} Q\rangle$ and $P' = \langle A, B', \{\mathsf{e}_j : R_j\}_{j\in J} Q'\rangle$ where $B' = \mathsf{track}(B, \mathsf{dom}(\mu), \mathsf{dom}(\mu'))$ and $\mu \fatsemi Q \to \mu' \fatsemi Q'$. Let $A = \{a_i\}_{i\in I}$. From rule [T-RUNNING TRANSACTION] we deduce:

- $\Gamma = \Gamma_1, \{a_i : \{\mathsf{e}_j : S_{ij}\}_{j\in J} T_i\}_{i\in I}, \Gamma_2$;
- (T.1) $\tilde{\mathscr{E}}\{\mathsf{e}_j\}_{j\in J}; \Gamma_0; \Gamma_R; [[\Gamma_S], \Gamma_1], \{a_i : T_i\}_{i\in I}, \Gamma_2 \vdash \mu \fatsemi Q$;
- (T.2) $\tilde{\mathscr{E}}; \Gamma_1, \{a_i : S_{ij}\}_{i\in I} \vdash R_j$ for every $j \in J$;
- (T.3) μ-$\mathsf{balanced}(\{a_i : S_{ij}\}_{i\in I})$;
- (T.4) μ-$\mathsf{balanced}(B)$;
- (T.5) $\mathsf{local}(\Gamma_2)$;
- (T.6) $\{a_i\}_{i\in I} \cup B = \mu$-$\mathsf{reach}(\{a_i\}_{i\in I} \cup \mathsf{dom}(\Gamma_2))$.

Let $\Gamma_3 = \{a_i : T_i\}_{i\in I}, \Gamma_2$. From (T.1) and $\mathsf{unsealed}(\Gamma)$ by induction hypothesis we obtain that there exist Γ'_0 and Γ'_3 such that:

(1') $\tilde{\mathscr{E}}; \Gamma'_0; \Gamma_R; [[\Gamma_S], \Gamma_1], \Gamma'_3 \vdash \mu' \fatsemi Q'$, and

(2') $\mathsf{unsealed}(\Gamma_3')$ and for every $a \in \mathsf{dom}(\Gamma_3) \cap \mathsf{dom}(\Gamma_3')$ we have $\mathsf{rank}(\Gamma_3(a)) = \mathsf{rank}(\Gamma_3'(a))$ and for every $a \in \mathsf{dom}(\Gamma_3) \setminus \mathsf{dom}(\Gamma_3')$ we have $\mathsf{rank}(\Gamma_3(a)) = 0$ and for every $a \in \mathsf{dom}(\Gamma_3') \setminus \mathsf{dom}(\Gamma_3)$ we have $\mathsf{rank}(\Gamma_3'(a)) = 0$, and

(3') for every $\Gamma_I \subseteq \Gamma_R, [[\Gamma_S], \Gamma_1]$ such that $\mu\text{-}\mathsf{balanced}(\mu\text{-}\mathsf{reach}(\mathsf{dom}(\Gamma_I, \Gamma_3)))$ we have

$$\mu\text{-}\mathsf{reach}(\mathsf{dom}(\Gamma_R, \Gamma_S, \Gamma_1) \setminus \mathsf{dom}(\Gamma_I)) = \mu'\text{-}\mathsf{reach}(\mathsf{dom}(\Gamma_R, \Gamma_S, \Gamma_1) \setminus \mathsf{dom}(\Gamma_I)).$$

Since $\mathsf{rank}(T_i) > 0$ for all $i \in I$, from (2') we deduce that all the a_i's are still in the environment for Q' therefore we have $\Gamma_3' = \{a : T_i'\}_{i \in I}, \Gamma_2'$. Let $\Gamma' = \Gamma_1, \{a_i : \{\mathsf{e}_j : S_{ij}\}_{j \in J} T_i'\}_{i \in I}, \Gamma_2'$.

Regarding (1), from (T.4) and Proposition A.1 we obtain (T.4') $\mu'\text{-}\mathsf{balanced}(B')$. From (2') we deduce (T.5') $\mathsf{local}(\Gamma_2')$. In order to prove (T.6') $\{a_i\}_{i \in I} \cup B' = \mu'\text{-}\mathsf{reach}(\{a_i\}_{i \in I} \cup \mathsf{dom}(\Gamma_2'))$ observe that:

$$\begin{aligned}
(*)\ & \mathsf{dom}(\mu') \\
&= (\mathsf{dom}(\mu) \cup (B' \setminus B)) \setminus (B \setminus B') && \text{by definition of } B' \\
&= (\mu\text{-}\mathsf{reach}(\mathsf{dom}(\Gamma_R, [\Gamma_S], \Gamma_1, \Gamma_2) \cup \{a_i\}_{i \in I}) \cup (B' \setminus B)) \setminus (B \setminus B') \\
& && \text{by item (3) of Definition 4.4} \\
&= (\mu\text{-}\mathsf{reach}(\mathsf{dom}(\Gamma_R, [\Gamma_S], \Gamma_1)) \\
&\qquad \uplus \mu\text{-}\mathsf{reach}(\mathsf{dom}(\Gamma_2) \cup \{a_i\}_{i \in I}) \cup (B' \setminus B)) \setminus (B \setminus B') \\
& && \text{by item (4) of Definition 4.4} \\
&= (\mu\text{-}\mathsf{reach}(\mathsf{dom}(\Gamma_R, [\Gamma_S], \Gamma_1)) \cup \{a_i\}_{i \in I} \cup B \cup (B' \setminus B)) \setminus (B \setminus B') && \text{by (T.6)} \\
&= \mu\text{-}\mathsf{reach}(\mathsf{dom}(\Gamma_R, [\Gamma_S], \Gamma_1)) \cup \{a_i\}_{i \in I} \cup B' && \text{by set theory}
\end{aligned}$$

where $\uplus$ denotes disjoint union. In addition, from (1') and Lemma A.3(1) we obtain $\Gamma_0'; \Gamma_R; [[\Gamma_S], \Gamma_1], \Gamma_3' \Vdash \mu'$, so we have:

$$\begin{aligned}
(**)\ \mathsf{dom}(\mu') &= \mu'\text{-}\mathsf{reach}(\mathsf{dom}(\Gamma_R, [\Gamma_S], \Gamma_1, \Gamma_2') \cup \{a_i\}_{i \in I}) \\
&= \mu'\text{-}\mathsf{reach}(\mathsf{dom}(\Gamma_R, [\Gamma_S], \Gamma_1)) \uplus \mu'\text{-}\mathsf{reach}(\mathsf{dom}(\Gamma_2') \cup \{a_i\}_{i \in I})
\end{aligned}$$

where the two equalities are respectively justified by items (3) and (4) of Definition 4.4. From (T.3), (T.4) we obtain $\mu\text{-}\mathsf{balanced}(\{a_i\}_{i \in I} \cup B)$, and then from (T.6) we get $\mu\text{-}\mathsf{balanced}(\mu\text{-}\mathsf{reach}(\{a_i\}_{i \in I} \cup \mathsf{dom}(\Gamma_2)))$. Therefore, by taking $\Gamma_I = \emptyset$ in (3') we obtain $\mu\text{-}\mathsf{reach}(\mathsf{dom}(\Gamma_R, [\Gamma_S], \Gamma_1)) = \mu'\text{-}\mathsf{reach}(\mathsf{dom}(\Gamma_R, [\Gamma_S], \Gamma_1))$ and then from (*) and (**) we obtain (T.6'). We conclude this part of the proof with an application of rule [T-RUNNING TRANSACTION] to (1'), (T.2), (T.3), (T.4'), (T.5') and (T.6').

Regarding (2), we conclude from (2') and rule [WF-RUN] of Table 7. Regarding (3), let $\Gamma_J \subseteq \Gamma_R, [\Gamma_S]$ be such that μ-balanced(μ-reach(dom(Γ_J, Γ))). Now take $\Gamma_I = \Gamma_J, [\Gamma_1]$ and observe that μ-balanced(μ-reach(dom(Γ_I, Γ_3))). From (3') we conclude

$$\mu\text{-reach}(\mathsf{dom}(\Gamma_R, \Gamma_S, \Gamma_1) \setminus \mathsf{dom}(\Gamma_I)) = \mu'\text{-reach}(\mathsf{dom}(\Gamma_R, \Gamma_S, \Gamma_1) \setminus \mathsf{dom}(\Gamma_I))$$

which is (3) because $\mathsf{dom}(\Gamma_R, \Gamma_S, \Gamma_1) \setminus \mathsf{dom}(\Gamma_I) = \mathsf{dom}(\Gamma_R, \Gamma_S) \setminus \mathsf{dom}(\Gamma_J)$.

[R-CATCH EXCEPTION] In this case $P = \langle \{a_i\}_{i\in I}, \mathsf{dom}(\mu_2), \{\mathsf{e}_j : Q_j\}_{j\in J} \texttt{throw}\ \mathsf{e}_k \mid P'' \rangle$ and $\mu = \mu_1, \{a_i \mapsto [b_i, \mathfrak{Q}_i]\}_{i\in I}, \mu_2$ where $k \in J$ and $P' = Q_k$ and $\mu' = \mu_1, \{a_i \mapsto [b_i, \varepsilon]\}_{i\in I}$. From rule [T-RUNNING TRANSACTION] we deduce:

- (L.1) $\Gamma = \Gamma_1, \{a_i : \{\mathsf{e}_j : S_{ij}\}_{j\in J} T_i\}_{i\in I}, \Gamma_2$;
- (H.1) $\tilde{\mathscr{E}}; [\Gamma_S], \Gamma_1, \{a_i : S_{ik}\}_{i\in I} \vdash P'$;
- (T.1) μ-balanced($\{a_i : S_{ik}\}_{i\in I}$);
- (T.2) local(Γ_2);
- (T.3) $\{a_i\}_{i\in I} \cup \mathsf{dom}(\mu_2) = \mu$-reach($\{a_i\}_{i\in I} \cup \mathsf{dom}(\Gamma_2)$).

Let $\Gamma_0' = \Gamma_0 \setminus \mathsf{dom}(\mu_2)$ and $\Gamma' = \Gamma_1, \{a_i : S_{ik}\}_{i\in I}$. We only have to show that (H.2) $\Gamma_0'; \Gamma_R, [\Gamma_S], \Gamma' \Vdash \mu'$ and we prove the items of Definition 4.4. Items (1), (2), and (4) are trivial because μ' has no more pointers than μ, some queues in μ have been emptied in μ', and duality of endpoint types associated with peer endpoints is preserved by (T.1). Regarding item (3), we have to show that $\mathsf{dom}(\mu') = \mathsf{dom}(\Gamma_0', \Gamma_R, [\Gamma_S], \Gamma') = \mu'\text{-reach}(\mathsf{dom}(\Gamma_R, [\Gamma_S], \Gamma'))$. The first equality is easy. Regarding the second equality, we derive:

$$\begin{aligned}
&\mathsf{dom}(\mu) \\
&\quad = \mu\text{-reach}(\{a_i\}_{i\in I} \uplus \mathsf{dom}(\Gamma_R, \Gamma_1, \Gamma_2)) && \text{by item (3) of Definition 4.4} \\
&\quad = \mu\text{-reach}(\mathsf{dom}(\Gamma_R, \Gamma_1)) \uplus \mu\text{-reach}(\{a_i\}_{i\in I} \cup \mathsf{dom}(\Gamma_2)) \\
&\qquad && \text{by item (4) of Definition 4.4} \\
&\quad = \mu\text{-reach}(\mathsf{dom}(\Gamma_R, \Gamma_1)) \uplus \{a_i\}_{i\in I} \uplus \mathsf{dom}(\mu_2) && \text{by (T.3)} \\
&\quad = \mathsf{dom}(\mu_1) \uplus \{a_i\}_{i\in I} \uplus \mathsf{dom}(\mu_2) && \text{by definition of } \mu
\end{aligned}$$

where we write $\uplus$ for disjoint union. From the last equality we deduce

$$(*) \qquad \mathsf{dom}(\mu_1) = \mu\text{-reach}(\mathsf{dom}(\Gamma_R, [\Gamma_S], \Gamma_1)) = \mu_1\text{-reach}(\mathsf{dom}(\Gamma_R, [\Gamma_S], \Gamma_1))$$

and now we have

$$\begin{aligned}
\mathsf{dom}(\mu') &= \mathsf{dom}(\mu_1) \uplus \{a_i\}_{i\in I} && \text{by definition of } \mu' \\
&= \mu_1\text{-}\mathsf{reach}(\mathsf{dom}(\Gamma_R, [\Gamma_S], \Gamma_1)) \cup \{a_i\}_{i\in I} && \text{from (*)} \\
&= \mu'\text{-}\mathsf{reach}(\mathsf{dom}(\Gamma_R, [\Gamma_S], \Gamma')) && \text{by definition of } \Gamma' \text{ and } \mu'
\end{aligned}$$

We conclude (1) from (H.1), (H.2) and rule [T-RUNNING PROCESS]. Regarding (2), from (L.1) we know that ranks of all pointers a_i are preserved and from (T.2) that that the rank of all pointers that are no more in the environment is 0. Regarding (3), it holds trivially.

[R-PROPAGATE EXCEPTION] Then $P = \langle \{a_i\}_{i\in I}, \mathsf{dom}(\mu_2), \{\mathsf{e}_j : Q_j\}_{j\in J} \texttt{throw}\ \mathsf{e} \mid P'' \rangle$ and $\mu = \mu_1, \{a_i \mapsto [b_i, \mathfrak{Q}_i]\}_{i\in I}, \mu_2$ and (E.1) $\mathsf{e}_j \neq \mathsf{e}$ for every $j \in J$ and $P' = \texttt{throw}\ \mathsf{e}$ and $\mu' = \mu_1, \{a_i \mapsto [b_i, \varepsilon]\}_{i\in I}$. From rule [T-RUNNING TRANSACTION] we deduce:

- (L.1) $\Gamma = \Gamma_1, \{a_i : \{\mathsf{e}_j : S_{ij}\}_{j\in J} T_i\}_{i\in I}, \Gamma_2$;
- (T.1) $\tilde{\mathscr{E}}\{\mathsf{e}_j\}_{j\in J}; \Gamma_0; \Gamma_R; [[\Gamma_S], \Gamma_1], \{a_i : T_i\}_{i\in I}, \Gamma_2 \vdash \mu \fatsemi \texttt{throw}\ \mathsf{e} \mid P''$
- (T.2) $\mu\text{-}\mathsf{balanced}(\{a_i : S_{ij}\}_{i\in I})$;
- (T.3) $\mathsf{local}(\Gamma_2)$;
- (T.4) $\{a_i\}_{i\in I} \cup \mathsf{dom}(\mu_2) = \mu\text{-}\mathsf{reach}(\{a_i\}_{i\in I} \cup \mathsf{dom}(\Gamma_2))$.

From (T.1), [T-RUNNING PARALLEL], [T-RUNNING PROCESS], and [T-THROW] we deduce $\mathsf{e} \in \tilde{\mathscr{E}}$. Let $\Gamma_0' = \Gamma_0 \setminus \mathsf{dom}(\mu_2)$ and $\Gamma' = \Gamma_1, \{a_i : S_{ik}\}_{i\in I}$ for some arbitrary $k \in J$. By rule [T-THROW] we derive $\tilde{\mathscr{E}}; [\Gamma_S], \Gamma' \vdash \texttt{throw}\ \mathsf{e}$. The proof that $\Gamma_0'; \Gamma_R, [\Gamma_S], \Gamma' \Vdash \mu'$ and that items (2) and (3) hold is the same as for the case [R-CATCH EXCEPTION]. □

B. Proof of Theorem 4.2 (Type Soundness)

The next two lemmas show the relationship between the free names of a process and the names occurring in the context used for typing it.

Lemma B.1. *If* $\tilde{\emptyset}; \Gamma \vdash P$, *then* $\mathsf{dom}(\Gamma) \subseteq \mathsf{fn}(P)$.

Proof. By induction on the derivation of $\tilde{\emptyset}; \Gamma \vdash P$ and by cases on the last rule applied. Most cases are trivial or follow by a simple induction. Case [T-THROW] is impossible because the exception environment consists of a sequence of empty

sets of exceptions, hence all the exceptions thrown in P are caught. The only interesting case is [T-TRY], when $P = \texttt{try}(\{u_i\}_{i\in I})\ \{\mathsf{e}_j : R_j\}_{j\in J} Q$, $\Gamma = \Gamma', \{u_i : \{\mathsf{e}_j : S_{ij}\}_{j\in J}[\![T_i]\!]_{i\in I}$, $\tilde{\emptyset}\{\mathsf{e}_j\}_{j\in J}; [\Gamma'], \{u_i : T_i\}_{i\in I} \vdash Q$, and $\tilde{\emptyset}; \Gamma', \{u_i : S_{ij}\}_{i\in I} \vdash R_j$ for all $j \in J$. We distinguish two subcases, according to whether J is empty or not. If $J = \emptyset$, then by induction hypothesis we obtain $\mathsf{dom}([\Gamma'], \{u_i : T_i\}_{i\in I}) \subseteq \mathsf{fn}(Q)$ and then we conclude $\mathsf{dom}(\Gamma) = \mathsf{dom}(\Gamma') \cup \{u_i\}_{i\in I} \subseteq \mathsf{fn}(Q) = \mathsf{fn}(P)$. If $J \neq \emptyset$, then by induction hypothesis we obtain $\mathsf{dom}(\Gamma', \{u_i : S_{ij}\}_{i\in I}) \subseteq \mathsf{fn}(R_j)$ for every $j \in J$. We conclude $\mathsf{dom}(\Gamma) = \mathsf{dom}(\Gamma') \cup \{u_i\}_{i\in I} \subseteq \bigcup_{j\in J} \mathsf{fn}(R_j) \subseteq \mathsf{fn}(P)$. □

Lemma B.2. *If* $\tilde{\emptyset}; \Gamma_0; \Gamma_R; \Gamma \vdash \mu \wr P$, *then* $\mathsf{dom}(\Gamma) \subseteq \mathsf{fn}(P)$.

Proof. By induction on the derivation of $\tilde{\emptyset}; \Gamma_0; \Gamma_R; \Gamma \vdash \mu \wr P$ and by cases on the last rule applied. Cases [T-RUNNING PROCESS] and [T-RUNNING PARALLEL] are easily solved by Lemma B.1 and the induction hypothesis, respectively. Regarding [T-RUNNING TRANSACTION], we have:

- $P = \langle \{a_i\}_{i\in I}, B, \{\mathsf{e}_j : R_j\}_{j\in J} Q \rangle$;
- $\Gamma = \Gamma_1, \{a_i : \{\mathsf{e}_j : S_{ij}\}_{j\in J} T_i\}_{i\in I}, \Gamma_2$;
- (*) $\{a_i\}_{i\in I} \cup B = \mu\text{-}\mathsf{reach}(\{a_i\}_{i\in I} \cup \mathsf{dom}(\Gamma_2))$;
- $\tilde{\emptyset}\{\mathsf{e}_j\}_{j\in J}; \Gamma_0; \Gamma_R; [\Gamma_1], \{a_i : T_i\}_{i\in I}, \Gamma_2 \vdash \mu \wr Q$;
- $\tilde{\emptyset}; \Gamma_1, \{a_i : S_{ij}\}_{i\in I} \vdash R_j$ for all $j \in J$.

We distinguish two cases, according to whether J is empty or not. If $J = \emptyset$, then by induction hypothesis we obtain $\mathsf{dom}([\Gamma_1], \{a_i : T_i\}_{i\in I}, \Gamma_2) \subseteq \mathsf{fn}(Q)$ and we conclude $\mathsf{dom}(\Gamma) \subseteq \mathsf{fn}(Q) \subseteq \mathsf{fn}(P)$. If $J \neq \emptyset$, then by induction hypothesis we obtain $\mathsf{dom}(\Gamma_1, \{a_i : S_{ij}\}_{i\in I}) \subseteq \mathsf{fn}(R_j)$ for every $j \in J$. From (*) we deduce $\mathsf{dom}(\Gamma_2) \subseteq \{a_i\}_{i\in I} \cup B$. We conclude $\mathsf{dom}(\Gamma) = \mathsf{dom}(\Gamma_1) \cup \{a_i\}_{i\in I} \cup \mathsf{dom}(\Gamma_2) \subseteq \bigcup_{j\in J} \mathsf{fn}(R_j) \cup \{a_i\}_{i\in I} \cup B \subseteq \mathsf{fn}(P)$. □

We conclude with the soundness proof of the type system.

Theorem 4.2. *Let* $\vdash P$. *Then* P *is well behaved.*

Proof. From the hypothesis $\vdash P$ we deduce $\tilde{\emptyset}; \emptyset; \emptyset; \emptyset \vdash \emptyset \wr P$. Consider an arbitrary derivation $\emptyset \wr P \Rightarrow \mu \wr Q$. From Theorem 4.1 we deduce that there exist Γ_0 and Γ such that $\tilde{\emptyset}; \Gamma_0; \emptyset; \Gamma \vdash \mu \wr Q$ and, from Lemma A.3, we obtain $\Gamma_0; \Gamma \Vdash \mu$.

Regarding condition (1) of Definition 3.5, from $\Gamma_0; \Gamma \Vdash \mu$ and Definition 4.4 we know $\mathsf{dom}(\mu) = \mu\text{-}\mathsf{reach}(\mathsf{dom}(\Gamma))$. By Lemma B.2 we have $\mathsf{dom}(\Gamma) \subseteq \mathsf{fn}(Q)$.

We conclude $\mathsf{dom}(\mu) = \mu\text{-}\mathsf{reach}(\mathsf{dom}(\Gamma)) \subseteq \mu\text{-}\mathsf{reach}(\mathsf{fn}(Q))$ because $\mu\text{-}\mathsf{reach}(\cdot)$ is monotone.

Regarding condition (2) of Definition 3.5, suppose that $Q \equiv Q_1 \mid Q_2$ and $\mu \fatsemi Q_1 \not\to$ and $Q_1 \not\equiv \texttt{throw}\ \mathsf{e} \mid Q_1'$ (the last hypothesis being granted by the fact that Q is well typed in the $\tilde{\emptyset}$ exception environment). We prove $\mu \fatsemi Q_1 \downarrow$ by induction on Q_1.

- ($Q_1 = \texttt{done}$) We conclude with an application of rule [ST-INACTIVE].
- ($Q_1 = \texttt{open}(a,b).R$) Because we have assumed that there are infinitely many pointers and structural congruence includes alpha renaming, we may assume $a, b \notin \mathsf{dom}(\mu)$. Then $\mu \fatsemi Q_1 \to$, which contradicts the hypothesis, therefore this case is impossible.
- ($Q_1 = \texttt{close}(a).R$) From [T-RUNNING PARALLEL], [T-RUNNING PROCESS], [T-CLOSE] and item (3) of Definition 4.4 we obtain $a \in \mathsf{dom}(\Gamma) \subseteq \mathsf{dom}(\Gamma_0, \Gamma) = \mathsf{dom}(\mu)$, and now $\mu \fatsemi Q_1 \to$ which contradicts the hypothesis, therefore this case is impossible.
- ($Q_1 = R_1 \oplus R_2$) This case is impossible because $\mu \fatsemi R_1 \oplus R_2$ always reduces.
- ($Q_1 = a!\mathtt{m}(c).R$) From rules [T-RUNNING PROCESS], [T-RUNNING PARALLEL] and [T-SEND] we obtain $\Gamma \vdash a : T$ where T is an internal choice and then from item (3) of Definition 4.4 we have $a \in \mathsf{dom}(\mu)$. From item (2) of Definition 4.4 we deduce that the queue associated with a is empty and also that the peer of a, say b, is still allocated in μ for otherwise T would have to be $\texttt{end}$. Then $\mu \fatsemi Q_1 \to$ which contradicts the hypothesis, therefore this case is impossible.
- ($Q_1 = \sum_{i \in I} a?\mathtt{m}_i(x_i).R_i$) Then $a \mapsto [b, \mathfrak{Q}] \in \mu$ and the messages and arguments in $\mathfrak{Q}$ are consistent with the type of endpoint a. The only case when $\mu \fatsemi \sum_{i \in I} a?\mathtt{m}_i(x_i).R_i$ does not reduce is when $\mathfrak{Q} = \varepsilon$, therefore we conclude $\mu \fatsemi Q_1 \downarrow$ by an application of rule [ST-INPUT].
- ($Q_1 = R_1 \mid R_2$) From the hypothesis $\mu \fatsemi Q_1 \not\to$ we deduce $\mu \fatsemi R_i \not\to$ for $i \in \{1,2\}$. From the hypothesis $Q_1 \not\equiv \texttt{throw}\ \mathsf{e} \mid Q_1'$ we deduce $R_i \not\equiv \texttt{throw}\ \mathsf{e} \mid R_i'$ for $i \in \{1,2\}$. By induction hypothesis we obtain $\mu \fatsemi R_i \downarrow$ for $i \in \{1,2\}$. We conclude with an application of rule [ST-PARALLEL].
- ($Q_1 = \texttt{try}(A)\ \{\mathsf{e}_j : R_j\}_{j \in J} R'$) From the hypothesis $\mu \fatsemi Q_1 \not\to$ we deduce $\neg \mu\text{-}\mathsf{balanced}(A)$. We conclude with an application of rule [ST-TRY].

- ($Q_1 = \texttt{throw e}$) This case is impossible by hypothesis.
- ($Q_1 = \texttt{commit}(A).R$) We conclude immediately with an application of rule [ST-COMMIT].
- ($Q_1 = X\langle\tilde{a}\rangle$) From rule [T-INVOKE] we deduce $X(\tilde{u}) \stackrel{\text{def}}{=} R$ is a definition and $\tilde{a}$ and $\tilde{u}$ have the same length. Then $\mu \fatsemi X\langle\tilde{a}\rangle \rightarrow$ which contradicts the hypothesis, therefore this case is impossible.
- ($Q_1 = \langle A, B, \{\texttt{e}_j : R_j\}_{j\in J} R'\rangle$) From the hypothesis $\mu \fatsemi Q_1 \not\rightarrow$ we deduce $\mu \fatsemi R' \not\rightarrow$ and $R' \not\equiv \prod_{i\in I} \texttt{commit}(A_i).R_i$ and $R' \not\equiv \texttt{throw e} \,|\, R''$ since these are the cases when $\mu \fatsemi Q_1$ does reduce. By induction hypothesis we deduce $\mu \fatsemi R' \downarrow$ and we conclude with an application of rule [ST-RUNNING TRANSACTION]. □

References

[1] G. Hunt, J. R. Larus, M. Abadi, M. Aiken, P. Barham, M. Fähndrich, C. Hawblitzel, O. Hodson, S. Levi, N. Murphy, B. Steensgaard, D. Tarditi, T. Wobber, B. Zill, An Overview of the Singularity Project, Technical Report MSR-TR-2005-135, Microsoft Research, 2005.

[2] G. Hunt, J. R. Larus, Singularity: Rethinking the Software Stack, SIGOPS Operating Systems Review 41 (2007) 37–49.

[3] S. Finley, Ext2 on Singularity, Technical Report, University of Wisconsin, 2008. Available at `http://pages.cs.wisc.edu/~remzi/Classes/736/Spring2008/Projects/Scott/Ext2%20On%20Singularity.pdf`.

[4] M. Fähndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. Hunt, J. R. Larus, S. Levi, Language Support for Fast and Reliable Message-based Communication in Singularity OS, in: Proceedings of EuroSys'06, ACM, 2006, pp. 177–190.

[5] Z. Stengel, T. Bultan, Analyzing Singularity Channel Contracts, in: Proceedings of the 18th International Symposium on Software Testing and Analysis (ISSTA'09), ACM, 2009, pp. 13–24.

[6] J. Villard, E. Lozes, C. Calcagno, Proving Copyless Message Passing, in: Proceedings of the 7th Asian Symposium on Programming Languages and Systems (APLAS'09), LNCS 5904, Springer, 2009, pp. 194–209.

[7] V. Bono, L. Padovani, Typing Copyless Message Passing, Logical Methods in Computer Science 8 (2012) 1–50.

[8] K. Honda, Types for Dyadic Interaction, in: Proceedings of the 4th International Conference on Concurrency Theory (CONCUR'93), LNCS 715, Springer, 1993, pp. 509–523.

[9] K. Honda, V. T. Vasconcelos, M. Kubo, Language Primitives and Type Disciplines for Structured Communication-based Programming, in: Proceedings of the 7th European Symposium on Programming (ESOP'98), LNCS 1381, Springer, 1998, pp. 122–138.

[10] S. Jakšić, L. Padovani, Exception Handling for Copyless Messaging, in: Proceedings of the 14th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP'12), ACM, 2012, pp. 151–162.

[11] M. Carbone, K. Honda, N. Yoshida, Structured Interactional Exceptions in Session Types, in: Proceedings of the 19th International Conference on Concurrency Theory (CONCUR'08), LNCS 5201, Springer, 2008, pp. 402–417.

[12] S. Capecchi, E. Giachino, N. Yoshida, Global Escape in Multiparty Sessions, in: Proceedings of the IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'10), 2010, pp. 338–351.

[13] R. Milner, Communicating and mobile systems: the π-calculus, Cambridge University Press, New York, NY, USA, 1999.

[14] S. Gay, M. Hole, Subtyping for Session Types in the π-calculus, Acta Informatica 42 (2005) 191–225.

[15] S. L. Peyton Jones, C. D. Clack, J. Salkild, M. Hardie, GRIP - A High-Performance Architecture for Parallel Graph Reduction, in: Proceedings of Functional Programming Languages and Computer Architecture (FPCA'87), LNCS 274, Springer, 1987, pp. 98–112.

[16] P. W. Trinder, K. Hammond, J. S. Mattson Jr., A. Partridge, S. L. Peyton Jones, GUM: A Portable Parallel Implementation of Haskell, in: Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation (PLDI'96), ACM, 1996, pp. 79–88.

[17] H.-W. Loidl, The Virtual Shared Memory Performance of a Parallel Graph Reduce, in: Proceedings of the 2nd IEEE International Symposium on Cluster Computing and the Grid (CCGRID'02), 2002, pp. 311–318.

[18] M. Tofte, J.-P. Talpin, Region-based memory management, Information and Computation 132 (1997) 109 – 176.

[19] M. Hofmann, S. Jost, Static Prediction of Heap Space Usage for First-Order Functional Programs, in: Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'03), ACM, 2003, pp. 185–197.

[20] M. Hofmann, S. Jost, Type-Based Amortised Heap-Space Analysis, in: Proceedings of the 15th European Symposium on Programming (ESOP'06), LNCS 3924, Springer, 2006, pp. 22–37.

[21] E. Albert, S. Genaim, M. Gómez-Zamalloa, Heap Space Analysis for Java Bytecode, in: Proceedings of the 6th International Symposium on Memory Management (ISMM'07), ACM, 2007, pp. 105–116.

[22] K. Hammond, The Dynamic Properties of Hume: A Functionally-Based Concurrent Language with Bounded Time and Space Behaviour, in: Proceedings of the 12th International Workshop on the Implementation of Functional Languages (IFL'00), LNCS 2011, Springer, 2000, pp. 122–139.

[23] K. Suenaga, R. Fukuda, A. Igarashi, Type-Based Safe Resource Deallocation for Shared-Memory Concurrency, in: Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'12), ACM, 2012, pp. 1–20.

[24] F. Iwama, A. Igarashi, N. Kobayashi, Resource Usage Analysis for a Functional Language with Exceptions, in: Proceedings of the 2006 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation (PEPM'06), ACM, 2006, pp. 38–47.

[25] J. Villard, E. Lozes, C. Calcagno, Tracking Heaps That Hop with Heap-Hop, in: Proceedings of the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'10), LNCS 6015, Springer, 2010, pp. 275–279.

[26] C. Ferreira, I. Lanese, A. Ravara, H. T. Vieira, G. Zavattaro, Advanced Mechanisms for Service Combination and Transactions, in: Rigorous Software Engineering for Service-Oriented Systems, LNCS 6582, Springer, 2011, pp. 302–325.

[27] T. Harris, S. Marlow, S. Peyton Jones, M. Herlihy, Composable Memory Transactions, in: Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'05), ACM, 2005, pp. 48–60.

[28] K. Donnelly, M. Fluet, Transactional Events, in: Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming (ICFP'06), ACM, 2006, pp. 124–135.

[29] W. Weimer, G. C. Necula, Finding and Preventing Run-Time Error Handling Mistakes, in: Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'04), ACM, 2004, pp. 419–431.

[30] W. Weimer, Exception-Handling Bugs in Java and a Language Extension to Avoid Them, in: Advanced Topics in Exception Handling Techniques, LNCS 4119, Springer, 2006, pp. 22–41.

[31] W. Weimer, G. C. Necula, Exceptional situations and program reliability, ACM Trans. Program. Lang. Syst. 30 (2008) 1–51.