Exception Handling for Copyless Messaging

Svetlana Jakšić
Univerzitet u Novom Sadu, Fakultet tehničkih nauka
sjaksic@uns.ac.rs

Luca Padovani
Università di Torino, Dipartimento di Informatica
luca.padovani@unito.it

Abstract

Copyless messaging is a communication mechanism in which only pointers to messages are exchanged between sender and receiver processes. Because of its intrinsically low overhead, copyless messaging can be profitably adopted for the development of complex software systems where processes have access to a shared address space. However, the very same mechanism 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.

Categories and Subject Descriptors F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs—Control primitives, Type structure; D.3.3 [*Programming Languages*]: Language Constructs and Features; F.1.2 [*Computation by Abstract Devices*]: Modes of Computation—Parallelism and concurrency

General Terms Languages, Theory

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 for short) [10, 11] is a notable example of a system that heavily relies on the copyless paradigm. In Singularity OS, processes have access to a shared region called the *exchange heap* that is explicitly managed (for practical reasons, objects on the exchange heap cannot be garbage collected, but must be explicitly allocated and deallocated by processes). 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.

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* (the access to unallocated/deallocated objects in the heap); *memory leaks* (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.

Earlier works [1, 5, 12, 13] have studied and formalized some aspects of Sing$^{\#}$. In particular, in [1] 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 *exception handling*. In particular, we contribute an extension of Sing$^{\#}$ types together with an enhancement in the semantics of exception handling to prevent the aforementioned programming errors even in presence of exceptions, if suitable exception handlers are provided. Copyless messaging and exceptions are 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 surprisingly, these two aspects can be reconciled only with some native support from the runtime system.

Structure of the paper. Section 2 illustrates the problem we are attacking and informally sketches 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 discusses similarities and differences between the present work and related ones. Section 6 concludes with a brief summary of the work and hints at possible extensions of the type system, in light of the common pattern usage of exception handling mechanisms as found in the source code of Singularity OS. *Appendices A and B contain additional technical material and the proofs of the presented results.*

```
 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 Exception(error);
21            delete exp;
22            delete nakImp;
23            break;
24        }
25      }
26    } finally {
27      delete ns;
28    }
29  }
```

Figure 1. Example of Sing# function.

2. Motivating Example

To introduce the context in which we operate and the kind of problems we have to face let us 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. Messages sent over one peer are received from the other peer, and vice versa. Each peer is associated with a FIFO buffer containing the messages sent to that peer that have not been received yet.

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` (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 the control flow to various cases depending on the kind 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`-tagged message (lines 12–15) or a `NakRegister`-tagged 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).

[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.

```
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.

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 `message` keyword 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) 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 `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. On the contrary, 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 sudden jump in the control flow of one of these processes, like that caused by an exception, may disrupt the alignment of the two peers of a channel and compromise the correctness of subsequent interactions. Therefore, exceptions cannot be handled locally within a single `try` block, but must be propagated to all the processes affected so that they can move in a coordinated way to a new stage of the interaction.

(2) Messages that have been sent but not yet received and other objects allocated since the beginning of a `try` block cannot be simply forgotten if an exception is thrown, for they would immediately turn into memory leaks. In general, it is necessary to keep track of the allocated memory and of all the messages that have been circulating since the beginning of a `try` block so that these are properly deallocated or moved back to their original owner in case an exception is thrown.

Sing# provides limited and not fully satisfactory solutions for these problems. Regarding the first one, Sing# compensates the lack of a coordinated recovery for the processes affected by an exception by means of dynamic typing: endpoints have 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. The second problem seems to have been neglected. For example, the function in Figure 1 is prone to leak memory on line 20 in case the exception is thrown, since neither `exp` nor `nakImp` are properly deallocated. 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. At the same time it is unreasonable to require the code in the exception handler to take care of deallocations, if only because the handler may not be in the scope of these resources: in the example, `exp` and `nakImp` are not visible in `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 an alternative solution that combines static analysis (inspired by existing works on exception handling for sessions [2, 3]) and a transaction-like, all-or-nothing semantics of `try` blocks: either a `try` block is executed completely by all processes affected, and then its effects are committed and become permanent, or its execution is aborted by an exception and the state of the affected processes is restored to the one they had at the beginning of the `try` block, except that control is passed to their exception handlers. To keep the cost of state restoration reasonable, we devise the following mechanisms:

(A) We decorate `try` blocks with the set of endpoints used in them and we synchronize the initiation of these blocks so that any message sent through one of these endpoints will be received from another endpoint from the same set. In this way, we identify a (small) portion of the heap that needs to be restored in case an exception is thrown.

(B) Inside `try` blocks, we "seal" the type of any endpoint that is not in the decoration of the block 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, we need to dynamically keep track of the memory allocated within a `try` block so that this memory is 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.

P	$::=$		**Process**
		$\texttt{done}$	(inaction)
	$\mid$	$\texttt{open}(a,a).P$	(open channel)
	$\mid$	$\texttt{close}(u).P$	(close endpoint)
	$\mid$	$u!\texttt{m}(u).P$	(send)
	$\mid$	$\sum_{i\in I} u?\texttt{m}_i(x_i).P_i$	(receive)
	$\mid$	$P \oplus P$	(conditional)
	$\mid$	$P \mid P$	(parallel)
	$\mid$	$\texttt{try}(U)\ \{P\}P$	(initiate transaction)
	$\mid$	$\texttt{throw}$	(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.

3. Language

Syntax. We assume given an infinite set Pointers ranged over by a, b, ... representing heap addresses and an infinite set Variables ranged over by x, y, ... We let *names* u, v, ... range over elements of Pointers ∪ Variables. We use A, B, ... to denote sets of pointers, 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, ...

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 $u!\texttt{m}(v).P$ denotes a process that sends the message $\texttt{m}(v)$ on the endpoint u and then continues as P. A *message* is made of a *tag* $\texttt{m}$ and an argument v. The term $\sum_{i\in I} u?\texttt{m}_i(x_i).P_i$ denotes a process that waits for a message from endpoint u. According to the tag $\texttt{m}_i$ 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 $\texttt{try}(U)\ \{Q\}P$ denotes a process willing to initiate a transaction involving the endpoints U. The process P is the *body* of the transaction and is executed when the transaction is initiated, while Q is the *handler* of the transaction

$$
\begin{array}{l}
\mathrm{GetNextDiskPath}(DS, ret) \stackrel{\text{def}}{=} \\
\quad DS?\texttt{NewClientEndpoint}(ns). \\
\quad \texttt{try}(ns)\ \{\mathrm{Finally}\langle ns, DS, ret\rangle\}\mathrm{Loop}\langle ns, DS, ret\rangle
\end{array}
$$

$$
\begin{array}{l}
\mathrm{Loop}(ns, DS, ret) \stackrel{\text{def}}{=} \\
\quad \texttt{open}(imp, exp).ns!\texttt{Register}(imp). \\
\quad\quad ns?\texttt{AckRegister}().\texttt{commit}(ns). \\
\quad\quad\quad ret!\texttt{SetService}(exp).\mathrm{Finally}\langle ns, DS, ret\rangle \\
\quad\ + ns?\texttt{NakRegister}(nakImp). \\
\quad\quad\quad \texttt{throw} \oplus \texttt{close}(exp).\texttt{close}(nakImp). \\
\quad\quad\quad\quad\quad \mathrm{Loop}\langle ns, DS, ret\rangle
\end{array}
$$

$$
\mathrm{Finally}(ns, DS, ret) \stackrel{\text{def}}{=} \texttt{close}(ns).ret!\texttt{Result}(DS).\texttt{close}(ret)
$$

Figure 3. Encoding of the function in Figure 1.

which is executed if the transaction is aborted during the execution of the body. The term `throw` denotes the throwing of an exception, whose effect is to abort the currently running transaction and to execute its handler. The term $\texttt{commit}(U).P$ denotes a process willing to terminate the currently running transaction (involving the endpoints 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 to 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 $u?\texttt{m}(x).P$, 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 $\mathsf{bn}(P)$, can be found in Table 8 of Appendix A. 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 $u_1?\texttt{m}_1(x_1).P_1 + \cdots + u_n?\texttt{m}_n(x_n).P_n$ instead of $\sum_{i=1..n} u_i?\texttt{m}_i(x_i).P_i$; we omit message arguments when they are not used; we sometimes use a prefix form for 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 sports 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 $u?\texttt{a}(x).P + v?\texttt{b}(y).Q$ 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 assume there is only one kind of exception which is implicitly thrown by a `throw` process, whereas Sing$^\#$ supports multiple kinds. We claim that none of the choices we have made affects the results presented hereafter in a significant way.

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.

μ	$::=$		**Heap**
		$\emptyset$	(empty heap)
	$\mid$	$a \mapsto [a, \mathfrak{Q}]$	(endpoint structure)
	$\mid$	μ, μ	(heap composition)
$\mathfrak{Q}$	$::=$		**Queue**
		ε	(empty queue)
	$\mid$	$\texttt{m}(a)$	(message)
	$\mid$	$\mathfrak{Q} :: \mathfrak{Q}$	(queue composition)
P	$::=$		**Runtime process**
		$\dots$	(as in Table 1)
	$\mid$	$\langle A, A, \{P\}P\rangle$	(running transaction)

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

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 $ret!\texttt{SetService}(exp)$ in Figure 3.

Finally, note that in Figure 1 the function uses a free name `DS` for accessing a system service. In the encoding we explicitly mention DS as a parameter of the GetNextDiskPath process, implying that its ownership is transferred to GetNextDiskPath upon invocation. To preserve the linear usage of resources (of DS in this case), the Finally process sends DS back on ret before ret is closed (a more detailed example of ownership transfer can be found in [1]). ■

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 work modulo commutativity and associativity of heap composition and assume that $\emptyset$ is neutral with respect to composition. 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, $\texttt{m}(c)$ is a queue made of an $\texttt{m}$-tagged 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 notions. 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 $a \mapsto [b, \mathfrak{Q}] \in \mu$ and $b \mapsto [a, \mathfrak{Q}'] \in \mu$.

The notion of "closed scope" that we mentioned in the introduction 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 $\mu\text{-}\mathsf{balanced}(A)$, if, for every $a \in A$, $a \overset{\mu}{\leftrightarrow} b$ implies $b \in A$.

In words, A is balanced if for every a in A, the peer of a is also in A provided that it is still allocated. 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 handler. 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 $\langle A, B, \{Q\}P\rangle$ 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, and P and Q respectively represent the (residual) body and the handler of the transaction. In general, P and Q 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 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* $\mu \,⨾\, P$ rather than of processes alone, so that

$$\mu \,⨾\, P \rightarrow \mu' \,⨾\, P'$$

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

We devote the following paragraphs to an informal description of the reduction rules of Table 3. 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 $\mathsf{Pointers}$ is infinite, it is always possible to alpha rename a and b to fresh pointers, so that an $\mathtt{open}(a,b).P$ 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).

Rule (R-Choice) (and its symmetric, omitted) states that a process $P \oplus Q$ nondeterministically reduces to 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. The operation may change the ownership of c, if b is owned by a process different from the sender. Note that, for this rule to be applicable, it is necessary for both endpoints of a channel to 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 when it is only one subprocess to reduce.

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 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

$$\langle \textstyle\bigcup_{i\in I} A_i, \emptyset, \{\prod_{i\in I} Q_i\} \prod_{i\in I} P_i\rangle$$

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.

Rule (R-End Transaction) reduces a running transaction to its continuation when its body has terminated. The handler is discarded.

Rule (R-Run Transaction) allows the reduction of a transaction according to the reductions of its body. The rule keeps track of the memory (de)allocated by the body of the transaction by updating the B set according to the changes of the heap.

Rule (R-Abort Transaction) describes the abnormal termination of a running transaction when an exception is thrown within it. In this case, the queues of all the endpoints involved in the transactions are emptied, the memory allocated within the transaction is deallocated, and the handler is run.

Finally, rule (R-Invoke) describe 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.

We write $\mu \,⨾\, P \rightarrow$ if $\mu \,⨾\, P \rightarrow \mu' \,⨾\, P'$ for some μ' and P' and $\mu \,⨾\, P \not\rightarrow$ if not $\mu \,⨾\, P \rightarrow$.

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 $\mathsf{fn}(P)$ (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 $\mu\text{-}\mathsf{reach}(A) = \{c \in \mathsf{Pointers} \mid \exists a \in A : c \preccurlyeq_\mu a\}$.

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.

Structural congruence

$$\begin{array}{lll}
\text{(S-PAR IDLE)} & \text{(S-PAR COMM)} & \text{(S-PAR ASSOC)} \\
P \mid \mathtt{done} \equiv P & P \mid Q \equiv Q \mid P & P \mid (Q \mid R) \equiv (P \mid Q) \mid R
\end{array}$$

Reduction relation

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

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

(R-CHOICE)
$$\mu \fatsemi P \oplus Q \to \mu \fatsemi P$$

(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-END TRANSACTION)
$$\mu \fatsemi \langle A, B, \{Q\} \textstyle\prod_{i\in I} \mathtt{commit}(A_i).P_i \rangle \to \mu \fatsemi \textstyle\prod_{i\in I} P_i$$

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

(R-PARALLEL)
$$\frac{\mu \fatsemi P \to \mu' \fatsemi P'}{\mu \fatsemi P \mid Q \to \mu' \fatsemi P' \mid Q}$$

(R-START TRANSACTION)
$$\frac{\mu\text{-balanced}(\bigcup_{i\in I} A_i)}{\mu \fatsemi \prod_{i\in I} \mathtt{try}(A_i)\ \{Q_i\} P_i \to \mu \fatsemi \langle \bigcup_{i\in I} A_i, \emptyset, \{\prod_{i\in I} Q_i\} \prod_{i\in I} P_i \rangle}$$

(R-STRUCT)
$$\frac{P \equiv P' \qquad \mu \fatsemi P' \to \mu' \fatsemi Q' \qquad Q' \equiv Q}{\mu \fatsemi P \to \mu' \fatsemi Q}$$

(R-RUN TRANSACTION)
$$\frac{\mu \fatsemi P \to \mu' \fatsemi P'}{\mu \fatsemi \langle A, B, \{Q\} P \rangle \to \mu' \fatsemi \langle A, (B \cup (\mathsf{dom}(\mu') \setminus \mathsf{dom}(\mu))) \setminus (\mathsf{dom}(\mu) \setminus \mathsf{dom}(\mu')), \{Q\} P' \rangle}$$

(R-INVOKE)
$$\frac{X(\tilde{u}) \overset{\text{def}}{=} P}{\mu \fatsemi X\langle \tilde{a} \rangle \to \mu \fatsemi P\{\tilde{a}/\tilde{u}\}}$$

(R-ABORT TRANSACTION)
$$\mu_1, \{a_i \mapsto [b_i, \mathfrak{Q}_i]\}_{i\in I}, \mu_2 \fatsemi \langle \{a_i\}_{i\in I}, \mathsf{dom}(\mu_2), \{Q\} \mathtt{throw} \mid P \rangle \to \mu_1, \{a_i \mapsto [b_i, \varepsilon]\}_{i\in I}, \fatsemi Q$$

Table 3. Operational semantics of processes.

Definition 3.4 (stuck configuration)**.** We say that the configuration $\mu \fatsemi P$ is *stuck* if the judgment $\mu \fatsemi P \downarrow$ is inductively derivable by the rules:

(ST-INPUT)
$$\mu, a \mapsto [b,\varepsilon] \fatsemi \textstyle\sum_{i\in I} a?\mathtt{m}_i(x_i).P_i \downarrow$$

(ST-COMMIT)
$$\mu \fatsemi \mathtt{commit}(A).P \downarrow$$

(ST-TRY)
$$\frac{\neg\mu\text{-balanced}(A)}{\mu \fatsemi \mathtt{try}(A)\ \{Q\} P \downarrow}$$

(ST-PARALLEL)
$$\frac{\mu \fatsemi P \downarrow \qquad \mu \fatsemi Q \downarrow}{\mu \fatsemi P \mid Q \downarrow}$$

(ST-IDLE)
$$\mu \fatsemi \mathtt{done} \downarrow$$

(ST-RUNNING TRANSACTION)
$$\frac{\mu \fatsemi P \downarrow \qquad P \not\equiv \prod_{i\in I} \mathtt{commit}(A_i).P_i}{\mu \fatsemi \langle A, B, \{Q\} P \rangle \downarrow}$$

Rules (ST-IDLE) 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. 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. $\mathsf{dom}(\mu) = \mu\text{-}\mathsf{reach}(\mathsf{fn}(Q))$;
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. 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-tagged message, but the other one was expecting a b-tagged message.
- The process

  $$\begin{array}{l}\mathtt{open}(a,b).\mathtt{try}(\emptyset)\ \{\mathtt{done}\} \\ \qquad \mathtt{throw} \oplus \mathtt{commit}(\emptyset).\mathtt{close}(a).\mathtt{close}(b)\end{array}$$

  may leak a and b if the exception is thrown.

4. Type System

4.1 Syntax of Types

We assume 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 4. 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 end-

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

Table 4. Syntax of types and endpoint types.

(WF-END)
$$\Theta \vdash \texttt{end} : 0$$

(WF-VAR)
$$\Theta, \{\alpha : r\} \vdash \alpha : r$$

(WF-REC)
$$\frac{\Theta, \{\alpha : r\} \vdash T : r}{\Theta \vdash \texttt{rec}\ \alpha : r.T : r}$$

(WF-PREFIX)
$$\frac{\dagger \in \{?, !\} \qquad \Theta \vdash S_i : 0 \ ^{(i\in I)} \qquad \Theta \vdash T_i : r \ ^{(i\in I)}}{\Theta \vdash \{\dagger\texttt{m}_i(S_i).T_i\}_{i\in I} : r}$$

(WF-COMMIT)
$$\frac{\Theta \vdash T : r}{\Theta \vdash]\!]T : r+1}$$

(WF-INITIATE)
$$\frac{\Theta \vdash S : r \qquad \Theta \vdash T : r+1}{\Theta \vdash \{S\}[\![T : r}$$

(WF-RUN)
$$\frac{\Theta \vdash S : r \qquad \Theta \vdash T : r+1}{\Theta \vdash \{S\}T : r}$$

Table 5. Rank of endpoint types.

point. The endpoint type end denotes an endpoint that can only be deallocated. An internal choice $\{!\texttt{m}_i(S_i).T_i\}_{i\in I}$ denotes an endpoint on which a process may send any message with tag $\texttt{m}_i$ for $i \in I$. The message has an argument of type S_i and, depending on the tag $\texttt{m}_i$ of the message, the endpoint can be used thereafter according to T_i. In a dual manner, an external choice $\{?\texttt{m}_i(S_i).T_i\}_{i\in I}$ denotes an endpoint from which a process must be ready to receive any message with tag $\texttt{m}_i$ for $i \in I$ and, depending on the tag $\texttt{m}_i$ of the received message, the endpoint is to be used according to T_i. In endpoint types $\{!\texttt{m}_i(S_i).T_i\}_{i\in I}$ and $\{?\texttt{m}_i(S_i).T_i\}_{i\in I}$ we assume that $I \neq \emptyset$ and $\texttt{m}_i = \texttt{m}_j$ implies $i = j$ for every $i, j \in I$. That is, the tag $\texttt{m}_i$ of the message that is sent or received identifies a unique continuation T_i. The endpoint type $\{S\}[\![T$ denotes an endpoint on which it is possible to initiate a transaction. The types T and S respectively specify how the endpoint is used within the transaction and if an exception aborts the transaction. 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 $\texttt{rec}\ \alpha : r.T$ 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 $\{S\}T$ is analogous to $\{S\}[\![T$, 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 $\langle A, B, \{Q\}P\rangle$. The programmer is in no case supposed to deal with endpoint types of this form.

Clearly, not every endpoint type written according to the syntax in Table 4 makes sense. For example, it is possible to write unbalanced terms such as $]\!]\texttt{end}$ or $\{\texttt{end}\}[\![\texttt{end}$ or terms where recursions do not respect the intended nesting of transactions, like in $\texttt{rec}\ \alpha.\{\texttt{end}\}[\![\alpha$ or in $\{\texttt{end}\}[\![\texttt{rec}\ \alpha.]\!]\alpha$. As far as our analysis is concerned, the syntax does not even prevent 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 is the number of transactions in which T is supposed to occur to make sense, with the proviso that end and, in general, well-formed endpoint types must have rank 0.

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 5, 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 properly nested. 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)). Then, well-typed processes will not be able to send/receive endpoints involved in pending transactions;
- end cannot occur inside transactions (rule (WF-END)). Then, well-typed processes will not be able to close endpoints involved in pending transactions.

The rank annotation r in recursive terms $\texttt{rec}\ \alpha : r.T$ guarantees that every well-formed endpoint type has a uniquely determined rank. Without this annotation a term like $\texttt{rec}\ \alpha.!\texttt{m}(\texttt{end}).\alpha$ 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* $\vdash \texttt{rec}\ \alpha : r.T : r$, *then* $\vdash T\{\texttt{rec}\ \alpha : r.T/\alpha\} : r$.

In what follows, we will assume that all endpoint types are 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 $\texttt{rank}(T)$ 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 $\texttt{rec}\ \alpha.T = T\{\texttt{rec}\ \alpha.T/\alpha\}$. Finally, we will sometimes use an infix notation for internal and external choices and write $!\texttt{m}_1(S_1).T_1 \oplus \cdots \oplus !\texttt{m}_n(S_n).T_n$ instead of $\{!\texttt{m}_i(S_i).T_i\}_{i\in\{1,\dots,n\}}$ and $?\texttt{m}_1(S_1).T_1 + \cdots + ?\texttt{m}_n(S_n).T_n$ instead of $\{?\texttt{m}_i(S_i).T_i\}_{i\in\{1,\dots,n\}}$.

Types are possibly sealed endpoint types of the form $[\cdots[T]\cdots]$ for some arbitrary 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 seal 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-tagged message and then for receiving either an AckRegister- or a NakRegister-tagged 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:

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

where T_{imp} is the (unspecified) 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:

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

If no exception is thrown, *ret* is used for sending a `SetRegister`-tagged message followed by a `Result`-tagged one; if an exception is thrown, only the `Result`-tagged message is sent. The above type T_{ret} takes into account both possibilities using conventional features of behavioral types (choices, sequentiality, and recursion). ■

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 is 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:

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

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

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

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

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

The interested reader may refer to [1] 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. $\mathsf{rank}(\overline{T}) = \mathsf{rank}(T)$.

4.2 Type Weight

In previous work [1] it was observed that the delegation of endpoints having some particular type 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

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

uses a and b according to the endpoint types

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

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 m-tagged 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:

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

In the final configuration we have $\mu\text{-}\mathsf{reach}(\mathsf{fn}(\texttt{done})) = \emptyset$ 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: it is as if the $b \mapsto [a,\texttt{m}(b)]$ region of the heap needs not be owned by any process because "it owns itself". To avoid these cycles we compute, for each endpoint type, a value in the set $\mathbb{N}\cup\{\infty\}$, that we call *weight*, estimating 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.

We proceed by recalling here the definition of weight from [1], 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:

- $T = \texttt{end}$ or $T = \{S\}[\![T'$ or $T =]\!]T'$ or $T = \{!\texttt{m}_i(S_i).T_i\}_{i\in I}$, or
- $T = \{?\texttt{m}_i(S_i).T_i\}_{i\in I}$ and $n > 0$ and $(S_i, n-1) \in \mathscr{W}$ and $(T_i, n) \in \mathscr{W}$ for every $i \in I$, or
- $T = \{S\}T'$ 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 $\|\texttt{end}\| = \|\{!\texttt{m}_i(S_i).T_i\}_{i\in I}\| = 0$. Indeed, the queues of endpoints with type `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 $\{S\}[\![T$ 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 $\|?\texttt{m}(\texttt{end}).\texttt{end}\| = 1$ and $\|?\texttt{m}(?\texttt{m}(\texttt{end}).\texttt{end}).\texttt{end}\| = 2$. 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 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 $\Gamma(u) = t$. We say that a type t is *local*, written

(T-INACTION) $\emptyset \vdash_n \texttt{done}$

(T-THROW) $\Gamma \vdash_{n+1} \texttt{throw}$

(T-CLOSE) $$\frac{\Gamma \vdash_n P}{\Gamma, u : \texttt{end} \vdash_n \texttt{close}(u).P}$$

(T-INVOKE) $$\frac{\Sigma(X) = (\tilde{t}, n)}{\tilde{u} : \tilde{t} \vdash_n X\langle \tilde{u} \rangle}$$

(T-OPEN) $$\frac{\vdash T : 0 \qquad \Gamma, a : T, b : \overline{T} \vdash_n P}{\Gamma \vdash_n \texttt{open}(a,b).P}$$

(T-SEND) $$\frac{k \in I \qquad \|S_k\| < \infty \qquad \Gamma, u : T_k \vdash_n P}{\Gamma, u : \{!\mathtt{m}_i(S_i).T_i\}_{i \in I}, v : S_k \vdash_n u!\mathtt{m}_k(v).P}$$

(T-CHOICE) $$\frac{\Gamma \vdash_n P \qquad \Gamma \vdash_n Q}{\Gamma \vdash_n P \oplus Q}$$

(T-RECEIVE) $$\frac{\Gamma, u : T_i, x_i : S_i \vdash_n P_i \ ^{(i \in I)}}{\Gamma, u : \{?\mathtt{m}_i(S_i).T_i\}_{i \in I} \vdash_n \sum_{i \in I} u?\mathtt{m}_i(x_i).P_i}$$

(T-PARALLEL) $$\frac{\Gamma_1 \vdash_n P \qquad \Gamma_2 \vdash_n Q}{\Gamma_1, \Gamma_2 \vdash_n P \mid Q}$$

(T-TRY) $$\frac{[\Gamma], \{u_i : T_i\}_{i \in I} \vdash_{n+1} P \qquad \Gamma, \{u_i : S_i\}_{i \in I} \vdash_n Q}{\Gamma, \{u_i : \{S_i\}\llbracket T_i\}_{i \in I} \vdash_n \texttt{try}(\{u_i\}_{i \in I})\ \{Q\}P}$$

(T-COMMIT) $$\frac{\mathsf{local}(\Gamma_2) \qquad \Gamma_1, \{u_i : T_i\}_{i \in I}, \Gamma_2 \vdash_n P}{[\Gamma_1], \{u_i : \rrbracket T_i\}_{i \in I}, \Gamma_2 \vdash_{n+1} \texttt{commit}(\{u_i\}_{i \in I}).P}$$

Table 6. Typing rules for processes.

$\mathsf{local}(t)$, 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 6. Judgments have the form $\Gamma \vdash_n P$ and state that process P within n nested transactions is well typed in the type environment Γ. The type system makes use of a global process environment Σ associating process variables X with pairs $(\tilde{t}, n)$ containing the type of the parameters of X as well as the nesting level n at 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

$$\frac{\Sigma(X) = (\tilde{t}, n) \qquad \tilde{u} : \tilde{t} \vdash_n P}{\Sigma \vdash X(\tilde{u}) \overset{\text{def}}{=} P}$$

holds. In particular, *all* of the free names of P must occur in its binding variable X.

We describe the typing rules for processes in the following paragraphs. Rule (T-IDLE) 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 $\texttt{end}$, 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 it is not possible to create endpoints with 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 $\mathtt{m}$-tagged messages. The type S of the argument v must be unsealed, finite-weight, and has to match 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.

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 message tags $\mathtt{m}_i$. The continuation processes may use the endpoint u according to the endpoint type T_i and can access the message argument x_i of type S_i.

Rules (T-CHOICE) and (T-PARALLEL) are standard. In the latter, the type environment is linearly split into two 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 at the correct nesting level.

All the rules discussed so far can be applied at arbitrary nesting levels and do not change it. We now turn our attention to the constructs dealing with transactions and exceptions.

Rule (T-THROW) states that the process $\texttt{throw}$ is well typed in *any* type environment, provided that it occurs within a transaction (the nesting level must be strictly positive). 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 type of other names is sealed so that P is prevented from using them until the transaction is terminated. Seals are not applied in the type environment for the handler since Q executes only if and when the transaction is aborted and therefore acts outside of the transaction. Note that the nesting level is increased inside P but does not change 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 nesting level is decreased in P, since it executes after the transaction has terminated.

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

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

where

$$\begin{array}{rl}
T'_{ns} = & !\mathtt{Register}(T_{imp}).(?\mathtt{AckRegister}().\rrbracket\texttt{end} + \\
& \qquad ?\mathtt{NakRegister}(T_{imp}).T_{ns})
\end{array}$$

is an appropriate residual of the unfolding of T_{ns}. ■

4.4 Typing the Heap

The typing rules in Table 6 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 care of running transaction processes (see Table 2) as well as of 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 4.3, 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 a well-typed heap must enjoy 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:

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

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

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

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.

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 4.3 (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. $a \mapsto [b, \mathfrak{Q}] \in \mu$ and $b \mapsto [a, \mathfrak{Q}'] \in \mu$ implies either $\mathfrak{Q} = \varepsilon$ or $\mathfrak{Q}' = \varepsilon$.
2. $a \mapsto [b, \mathtt{m}_1(c_1) :: \cdots :: \mathtt{m}_n(c_n)] \in \mu$ implies
$$\mathsf{tail}(T, \mathtt{m}_1(S_1)\cdots\mathtt{m}_n(S_n)) = S$$
where $\Gamma_0,\Gamma \vdash a : T$ and $\Gamma_0 \vdash c_i : S_i$ and $\|S_i\| < \infty$ and $\vdash S_i : 0$ for $1 \le i \le n$ and $b \mapsto [a, \varepsilon] \in \mu$ implies $\Gamma_0,\Gamma \vdash b : \overline{S}$ 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 $A, B \subseteq \mathsf{dom}(\Gamma)$.

(T-RUNNING PROCESS)
$$\frac{\Gamma_0;\Gamma_R,\Gamma \Vdash \mu \qquad \Gamma \vdash_n P}{\Gamma_0;\Gamma_R;\Gamma \vdash_n \mu \,⨟\, P}$$

(T-RUNNING PARALLEL)
$$\frac{\Gamma_0;\Gamma_R,\Gamma_2;\Gamma_1 \vdash_n \mu \,⨟\, P \qquad \Gamma_0;\Gamma_R,\Gamma_1;\Gamma_2 \vdash_n \mu \,⨟\, Q}{\Gamma_0;\Gamma_R;\Gamma_1,\Gamma_2 \vdash_n \mu \,⨟\, P \mid Q}$$

(T-RUNNING TRANSACTION)
$$\frac{\begin{array}{c}\mu\text{-}\mathsf{balanced}(\{a_i : S_i\}_{i\in I}) \qquad \mu\text{-}\mathsf{balanced}(B) \qquad \mathsf{local}(\Gamma_2) \\ \{a_i\}_{i\in I} \cup B = \mu\text{-}\mathsf{reach}(\{a_i\}_{i\in I} \cup \mathsf{dom}(\Gamma_2)) \\ \Gamma_0;\Gamma_R;[\Gamma_1],\{a_i : T_i\}_{i\in I},\Gamma_2 \vdash_{n+1} \mu \,⨟\, P \qquad \Gamma_1,\{a_i : S_i\}_{i\in I} \vdash_n Q\end{array}}{\Gamma_0;\Gamma_R;\Gamma_1,\{a_i : \{S_i\}T_i\}_{i\in I},\Gamma_2 \vdash_n \mu \,⨟\, \langle\{a_i\}_{i\in I}, B, \{Q\}P\rangle}$$

Table 7. Typing rules for configurations.

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. 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, 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.

4.5 Typing Configurations

Table 7 defines typing rules for configurations $\mu \,⨟\, P$ as an extension of the typing rules for processes. Judgments have the form

$$\Gamma_0;\Gamma_R;\Gamma \vdash_n \mu \,⨟\, P$$

and state that the configuration $\mu \,⨟\, P$ is well typed at nesting level n with respect to 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 analogous 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 $\langle\{a_i\}_{i\in I},B,\{Q\}P\rangle$, 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 4.4 (balanced context)**.** We say that Γ is *balanced* in μ, written $\mu\text{-balanced}(\Gamma)$, 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 in three parts Γ_1, $\{a_i : \{S_i\}T_i\}_{i\in I}$, 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 type of these endpoints are sealed in the judgment corresponding to the typing of P. The environment $\{a_i : \{S_i\}T_i\}_{i\in I}$ 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 two premises $\mu\text{-balanced}(\{a_i : S_i\}_{i\in I})$ and $\mu\text{-balanced}(B)$ 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_i 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 recovered 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 $\mathsf{local}(\Gamma_2)$ identifies the Γ_2 partition of the context corresponding to the endpoints that have been created inside the transaction. The premise $\{a_i\}_{i\in I} \cup B = \mu\text{-reach}(\{a_i\}_{i\in I} \cup \mathsf{dom}(\Gamma_2))$ 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 nesting level is increased by one 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 7 are sufficient to cover all possible forms of runtime configurations.

4.6 Type Soundness

We conclude this section with 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* $\Gamma_0;\Gamma_R;\Gamma \vdash_n \mu \,⨟\, P$ *and* $\mu \,⨟\, P \to \mu' \,⨟\, P'$. *Then* $\Gamma_0';\Gamma_R;\Gamma' \vdash_n \mu' \,⨟\, P'$ *for some* Γ_0' *and* Γ'.

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). The details are omitted here, but can be found in Appendix B.

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

5. Related work

This work follows the type-based formalization of Singularity OS detailed in [1]. To simplify the formal development of the present paper we dropped polymorphism and non-linear types from the type system in [1]. 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 in [13, 14], 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 [3] and [2]. In [3], 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; [2] 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 [3] and [2] it is possible that messages already present in channel queues at the time an exception occurs are discarded. In our context, this would easily lead to undesired 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. Neither [3] nor [2] consider session delegation, namely the communication of channels. Also, in [2] the type system forces inner `try` blocks to use a subset of the channels involved in outer blocks. We relax this restriction and allow locally created channels to be involved in inner transactions. The most notable difference between [2] and the present work regards the semantics of exceptions in nested transactions: in [2], 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 [2], 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 lots of references is provided in [6]. In our context, the component Q of a process $\langle A,B,\{Q\}P\rangle$ is analogous to a *compensation handler*. The main difference between our handlers and 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 in [7]. 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 [2]. The semantics of [7] uses a clever combination of small- and big-step reduction rules and is even more abstract than ours, but we find it more appropriate in a functional setting since non-terminating functions have smaller practical interest than non-terminating processes.

The authors of [4] 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. The authors of [15–17] 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 [6]. 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 for modeling Singularity processes that can throw exceptions and have studied a type system guaranteeing some safety properties, in particular that well-typed processes do no leak memory even in presence of (caught) exceptions. This property has fundamental importance in systems relying on copyless message passing, where the sharing of data and explicit memory allocation require controlled policies on the ownership of heap-allocated objects.

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. 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.

Future work. The one major theoretical aspect we are investigating is how to relax the type system and allow a wider set of messages to be exchanged within transactions. Currently, only local endpoints (those that are unsealed and have null rank) 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). We claim that endpoints with a sealed type are also safe to be sent as messages, although the proof of this fact seems to require a non-trivial modification of rule (T-RUNNING TRANSACTION) which is already quite elaborate in the present state. There are two main reasons why we think this extension is interesting: first of all, because it would grant transactions the ability to change the ownership of *existing* heap-allocated objects, in addition to that of new ones as is currently the case; second, because endpoints with a sealed type can be safely sent regardless of the weight of their type. In other words, transactions provide an effective mechanism to safely circumvent the finite-weight restriction imposed by the typing rule (T-SEND).

On the practical side, we plan to work on prototype implementations of the exception handling mechanism in a few different programming languages so as to explore its practical costs.

Acknowledgments

This work has been supported by Cost Action IC0901, the Serbian Ministry of Education and Science (projects ON174026 and III44006). We are grateful to Sara Capecchi and Elena Giachino who were keen on discussing some aspects of the semantics of distributed exception handling. Mariangiola Dezani and the PPDP reviewers have provided useful feedback for improving both presentation and content of the paper.

References

[1] Viviana Bono and Luca Padovani. Typing Copyless Message Passing. *Logical Methods in Computer Science*, 8:1–50, 2012.

[2] Sara Capecchi, Elena Giachino, and Nobuko Yoshida. Global escape in multiparty sessions. In *Proceedings of FSTTCS'10*, pages 338–351, 2010.

[3] Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured interactional exceptions in session types. In *Proceedings of CONCUR'08*, LNCS 5201, pages 402–417. Springer, 2008.

[4] Kevin Donnelly and Matthew Fluet. Transactional events. In *Proceedings of ICFP'06*, pages 124–135. ACM, 2006.

[5] Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen Hunt, James R. Larus, and Steven Levi. Language Support for Fast and Reliable Message-based Communication in Singularity OS. In *Proceedings of EuroSys'06*, pages 177–190. ACM, 2006.

[6] Carla Ferreira, Ivan Lanese, Antonio Ravara, Hugo Torres Vieira, and Gianluigi Zavattaro. Advanced mechanisms for service combination and transactions. In *Rigorous Software Engineering for Service-Oriented Systems*, LNCS 6582, pages 302–325. Springer, 2011.

[7] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. In *Proceedings of PPoPP'05*, pages 48–60. ACM, 2005.

[8] Kohei Honda. Types for Dyadic Interaction. In *Proceedings of CONCUR'93*, LNCS 715, pages 509–523. Springer, 1993.

[9] Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language Primitives and Type Disciplines for Structured Communication-based Programming. In *Proceedings of ESOP'98*, LNCS 1381, pages 122–138. Springer, 1998.

[10] Galen Hunt, James Larus, Martín Abadi, Mark Aiken, Paul Barham, Manuel Fähndrich, Chris Hawblitzel, Orion Hodson, Steven Levi, Nick Murphy, Bjarne Steensgaard, David Tarditi, Ted Wobber, and Brian Zill. An Overview of the Singularity Project. Technical Report MSR-TR-2005-135, Microsoft Research, 2005.

[11] Galen C. Hunt and James R. Larus. Singularity: Rethinking the Software Stack. *SIGOPS Operating Systems Review*, 41:37–49, 2007.

[12] Zachary Stengel and Tevfik Bultan. Analyzing Singularity Channel Contracts. In *Proceedings of ISSTA'09*, pages 13–24. ACM, 2009.

[13] Jules Villard, Étienne Lozes, and Cristiano Calcagno. Proving Copyless Message Passing. In *Proceedings of APLAS'09*, LNCS 5904, pages 194–209. Springer, 2009.

[14] Jules Villard, Étienne Lozes, and Cristiano Calcagno. Tracking Heaps That Hop with Heap-Hop. In *Proceedings of TACAS'10*, LNCS 6015, pages 275–279. Springer, 2010.

[15] Westley Weimer. Exception-handling bugs in java and a language extension to avoid them. In *Advanced Topics in Exception Handling Techniques*, LNCS 4119, pages 22–41. Springer, 2006.

[16] Westley Weimer and George C. Necula. Finding and preventing runtime error handling mistakes. In *Proceedings of OOPSLA'04*, pages 419–431. ACM, 2004.

[17] Westley Weimer and George C. Necula. Exceptional situations and program reliability. *ACM Trans. Program. Lang. Syst.*, 30(2):8:1–8:51, 2008.

Table 8. Free and bound names of processes.

$$\begin{aligned}
\mathsf{fn}(\mathtt{done}) = \mathsf{fn}(\mathtt{throw}) &= \emptyset\\
\mathsf{fn}(\mathtt{open}(a,b).P) &= \mathsf{fn}(P)\setminus\{a,b\}\\
\mathsf{fn}(\mathtt{close}(u).P) &= \{u\}\cup\mathsf{fn}(P)\\
\mathsf{fn}(u!\mathtt{m}(v).P) &= \{u,v\}\cup\mathsf{fn}(P)\\
\mathsf{fn}(\textstyle\sum_{i\in I} u?\mathtt{m}_i(x_i).P_i) &= \{u\}\cup\textstyle\bigcup_{i\in I}(\mathsf{fn}(P_i)\setminus\{x_i\})\\
\mathsf{fn}(P\oplus Q) = \mathsf{fn}(P\,|\,Q) &= \mathsf{fn}(P)\cup\mathsf{fn}(Q)\\
\mathsf{fn}(\mathtt{try}(U)\ \{Q\}P) &= U\cup\mathsf{fn}(P)\cup\mathsf{fn}(Q)\\
\mathsf{fn}(\langle A,B,\{Q\}P\rangle) &= A\cup B\cup\mathsf{fn}(P)\cup\mathsf{fn}(Q)\\
\mathsf{fn}(\mathtt{commit}(U).P) &= U\cup\mathsf{fn}(P)\\
\mathsf{fn}(X\langle\tilde{u}\rangle) &= \tilde{u}
\end{aligned}$$

$$\begin{aligned}
\mathsf{bn}(\mathtt{done}) = \mathsf{bn}(\mathtt{throw}) = \mathsf{bn}(X\langle\tilde{u}\rangle) &= \emptyset\\
\mathsf{bn}(\mathtt{open}(a,b).P) &= \{a,b\}\cup\mathsf{bn}(P)\\
\mathsf{bn}(\mathtt{close}(u).P) = \mathsf{bn}(u!\mathtt{m}(v).P) &= \mathsf{bn}(P)\\
\mathsf{bn}(\textstyle\sum_{i\in I} u?\mathtt{m}_i(x_i).P_i) &= \textstyle\bigcup_{i\in I}(\{x_i\}\cup\mathsf{bn}(P_i))\\
\mathsf{bn}(P\oplus Q) = \mathsf{bn}(P\,|\,Q) &= \mathsf{bn}(P)\cup\mathsf{bn}(Q)\\
\mathsf{bn}(\mathtt{try}(U)\ \{Q\}P) &= \mathsf{bn}(P)\cup\mathsf{bn}(Q)\cup\mathsf{bn}(R)\\
\mathsf{bn}(\langle A,B,\{Q\}P\rangle) &= \mathsf{bn}(P)\cup\mathsf{bn}(Q)\cup\mathsf{bn}(R)\\
\mathsf{bn}(\mathtt{commit}(U).P) &= \mathsf{bn}(P)
\end{aligned}$$

A. Supplementary definitions

Table 8 reports the formal definition of free and bound names for processes.

B. Proofs

Proposition B.1 (Proposition 4.1)**.** *Let $\vdash S : r$. Then $\Theta, \alpha : r \vdash T : r'$ implies $\Theta \vdash T\{S/\alpha\} : r'$.*

Proof. A simple induction on the derivation of $\Theta, \alpha : r \vdash T : r'$. □

Proposition B.2 (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 duality 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). □

B.1 Subject reduction

The two following lemmas are standard and prove that typing is preserved by structural congruence and by substitutions.

Lemma B.1. *Let $\Gamma \vdash_n P$ and $P \equiv Q$. Then $\Gamma \vdash_n Q$.*

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

Lemma B.2 (substitution)**.** *If $\Gamma, u : T \vdash_n P$ and $v \notin \mathsf{dom}(\Gamma) \cup \mathsf{bn}(P)$, then $\Gamma, v : S \vdash_n P\{v/u\}$.*

Proof. For notational simplicity we prove the result when $u = x$ and $v = a$. We proceed by induction on the derivation of $\Gamma, x : T \vdash_n P$ and by cases on the last rule applied. We only prove a few cases.

- (T-INACTION) This case is impossible.
- (T-CLOSE) In this case:
  - $P = \mathtt{close}(u).P'$;
  - $\Gamma, x : T = \Gamma', u : \mathtt{end}$;
  - $\Gamma' \vdash_n P'$.

  If $x \in \mathsf{dom}(\Gamma')$ then $\Gamma' = \Gamma'', x : T$ and $\Gamma = \Gamma'', x : T, u : \mathtt{end}$. By induction hypothesis we have $\Gamma'', a : T \vdash_n P'\{a/x\}$. Since we know that $a \neq u$, from rule (T-CLOSE) we obtain $\Gamma, a : T \vdash_n \mathtt{close}(u).P'\{a/x\}$.
  If $x = u$ then $\Gamma = \Gamma'$, $T = \mathtt{end}$ and $P' = P'\{a/x\}$. From rule (T-CLOSE) we obtain $\Gamma, a : \mathtt{end} \vdash_n \mathtt{close}(a).P'\{a/x\}$ which is what we needed to prove.
- (T-OPEN) In this case:
  - $P = \mathtt{open}(c,d).P'$

  Since we know $x \notin \{c,d\}$ the proof is concluded by a straightforward induction.
- (T-SEND) In this case:
  - $P = u!\mathtt{m}_k(v).P'$;

- $\Gamma, x : T = \Gamma', u : \{!\mathtt{m}_i(S_i).T_i\}_{i\in I}, v : S_k$;
- $k \in I$;
- $\|S_k\| < \infty$;
- $\Gamma', u : T_k \vdash_n P'$.

If $x \in \mathsf{dom}(\Gamma')$ then $\Gamma' = \Gamma'', x : T$ and $\Gamma = \Gamma'', u : \{!\mathtt{m}_i(S_i).T_i\}_{i\in I}, v : S_k$. From $\Gamma', u : T_k \vdash_n P'$ by induction hypothesis we obtain $\Gamma'', a : T, u : T_k \vdash_n P'\{a/x\}$. Since $a \notin \mathsf{dom}(\Gamma)$ we know that $a \notin \{u, v\}$ and from rule (T-SEND) we can conclude $\Gamma, a : T \vdash_n u!\mathtt{m}_k(v).P'\{a/x\}$.
If $x = u$ then $\Gamma = \Gamma', v : S_k$ and $T = \{!\mathtt{m}_i(S_i).T_i\}_{i\in I}$. From $\Gamma', u : T_k \vdash_n P'$ by induction hypothesis we obtain $\Gamma', a : T_k \vdash_n P'\{a/x\}$ and the from rule (T-SEND) we get $\Gamma', a : \{!\mathtt{m}_i(S_i).T_i\}_{i\in I}, v : S_k \vdash_n a!\mathtt{m}_k(v).P'\{a/x\}$.
If $x = v$ then $x \notin \mathsf{dom}(\Gamma') \cup \{u\}$ and $P' = P'\{a/x\}$. We conclude with an application of rule (T-SEND). □

The next lemma connects well-typed configurations to well-typed heaps and shows the irrelevance of the middle component Γ_R in typing processes.

Lemma B.3. *Let $\Gamma_0; \Gamma_R; \Gamma \vdash_n \mu \fatsemi P$. Then:*

(1) $\Gamma_0; \Gamma_R, \Gamma \Vdash \mu$;
(2) $\Gamma'_0; \Gamma'_R, \Gamma \Vdash \mu'$ implies $\Gamma'_0; \Gamma'_R; \Gamma \vdash_n \mu' \fatsemi P$.

Proof. By induction on the derivation of $\Gamma_0; \Gamma_R; \Gamma \vdash_n \mu \fatsemi P$ and by cases on the last rule applied. The only interesting case is the case of rule (T-RUNNING TRANSACTION), where we have:

- $P = \langle \{a_i\}_{i\in I}, B, \{R\}Q\rangle$;
- $\Gamma = \Gamma_1, \{a_i : \{S_i\}T_i\}_{i\in I}, \Gamma_2$;
- μ-$\mathsf{balanced}(\{a_i : S_i\}_{i\in I})$;
- μ-$\mathsf{balanced}(B)$;
- $\mathsf{local}(\Gamma_2)$;
- $\{a_i\}_{i\in I} \cup B = \mu$-$\mathsf{reach}(\{a_i\}_{i\in I} \cup \mathsf{dom}(\Gamma_2))$;
- $\Gamma_0; \Gamma_R; [\Gamma_1], \{a_i : T_i\}_{i\in I}, \Gamma_2 \vdash_{n+1} \mu \fatsemi Q$;
- $\Gamma_1, \{a_i : S_i\}_{i\in I} \vdash_n R$.

Regarding (1), from $[\Gamma_1], \{a_i : T_i\}_{i\in I}, \Gamma_2 \vdash_{n+1} \mu \fatsemi Q$ by induction hypothesis we obtain $[\Gamma_1], \{a_i : T_i\}_{i\in I}, \Gamma_2 \Vdash \mu$ and then from the definition of function tail and μ-$\mathsf{balanced}(\{a_i : S_i\}_{i\in I})$ we deduce $\Gamma_0; \Gamma_R, \Gamma \Vdash \mu$.

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 $\Gamma_0; \Gamma_R; [\Gamma_1], \{a_i : T_i\}_{i\in I}, \Gamma_2 \vdash_{n+1} \mu \fatsemi Q$ by induction hypothesis we obtain $\Gamma'_0; \Gamma'_R; [\Gamma_1], \{a_i : T_i\}_{i\in I}, \Gamma_2 \vdash_{n+1} \mu \fatsemi Q$. We conclude with an application of rule (T-RUNNING TRANSACTION). □

The following lemma provides a connection between the typing rule (T-PARALLEL) for processes and rule (T-RUNNING PARALLEL) for configurations. It is used for simplifying some cases in the proof of subject reduction (Lemma B.5).

Lemma B.4. *If $\Gamma_0; \Gamma_R; \Gamma \vdash_n \mu \fatsemi P_1 \mid P_2$ is derivable using rule* (T-RUNNING PROCESS) *it is also derivable using* (T-RUNNING PARALLEL).

Proof. From $\Gamma_0; \Gamma_R; \Gamma \vdash_n \mu \fatsemi P_1 \mid P_2$ and rule (T-RUNNING PROCESS) we obtain:

- (H.1) $\Gamma \vdash_n P_1 \mid P_2$;
- (H.2) $\Gamma_0; \Gamma_R, \Gamma \Vdash \mu$.

From (H.1) and rule (T-PARALLEL) we obtain:

- (T.1) $\Gamma = \Gamma_1, \Gamma_2$;
- (P.i) $\Gamma_i \vdash_n P_i$ for $i \in \{1, 2\}$.

From (H.2), (T.1), (P.i) and rule (T-RUNNING PROCESS) we obtain $\Gamma_0; \Gamma_R, \Gamma_{3-i}; \Gamma_i \vdash_n \mu \fatsemi P_i$ 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 B.3. *If $\mu \fatsemi P \to \mu' \fatsemi P'$ then μ'-$\mathsf{balanced}(\mathsf{dom}(\mu') \setminus \mathsf{dom}(\mu))$.*

Proof. Simple induction on the reduction that occurs. □

The next lemma is in fact the full version of Theorem 4.1 proving that reductions preserve well typedness and showing the relationship between the contexts used for typing the two configurations involved.

Lemma B.5 (Theorem 4.1)**.** *Let $\Gamma_0; \Gamma_R; [\Gamma_S], \Gamma \vdash_n \mu \fatsemi P$ where $\mathsf{unsealed}(\Gamma)$ and $\mu \fatsemi P \to \mu' \fatsemi P'$. Then there exist Γ'_0 and Γ' such that:*

(1) $\Gamma'_0; \Gamma_R; [\Gamma_S], \Gamma' \vdash_n \mu' \fatsemi P'$, 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 μ-$\mathsf{balanced}(\mu$-$\mathsf{reach}(\mathsf{dom}(\Gamma_I, \Gamma)))$ 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.

- (R-OPEN) In this case:
  - $P = \texttt{open}(a,b).P'$;
  - $\mu' = \mu, a \mapsto [b,\varepsilon], b \mapsto [a,\varepsilon]$.

  From rule (T-RUNNING PROCESS) we obtain:
  - (H.1) $[\Gamma_S], \Gamma \vdash_n \texttt{open}(a,b).P'$;
  - (H.2) $\Gamma_0; \Gamma_R, [\Gamma_S], \Gamma \Vdash \mu$.

  From (H.2) and rule (T-OPEN) we obtain:
  - $\vdash T : 0$;
  - (C.1) $[\Gamma_S], \Gamma, a : T, b : \overline{T} \vdash_n P'$.

  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:
  - $P = \texttt{close}(a).P'$;
  - $\mu = \mu', a \mapsto [b, \mathfrak{Q}]$.

  From rule (T-RUNNING PROCESS) we obtain:
  - (H.1) $[\Gamma_S], \Gamma \vdash_n \texttt{close}(a).P'$;
  - (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) $\Gamma' \vdash_n P'$.

  Let $\Gamma_0' = \Gamma_0$. We only have to show that (C.2) $\Gamma_0'; \Gamma_R, [\Gamma_S], \Gamma' \Vdash \mu'$. We prove the items of Definition 4.3 in order.
  1. Straightforward from hypothesis (H.2).
  2. Straightforward from hypothesis (H.2).
  3. We only need to prove $\mathsf{dom}(\Gamma_0', \Gamma_R, [\Gamma_S], \Gamma') = \mu'\text{-}\mathsf{reach}(\mathsf{dom}(\Gamma_R', [\Gamma_S], \Gamma'))$ since $\mathsf{dom}(\mu') = \mathsf{dom}(\Gamma_0', \Gamma_R, [\Gamma_S], \Gamma')$ is obvious. 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
  $$\mu'\text{-}\mathsf{reach}(\mathsf{dom}(\Gamma_R, [\Gamma_S], \Gamma')) = \mu\text{-}\mathsf{reach}(\mathsf{dom}(\Gamma_R, [\Gamma_S], \Gamma)) \setminus \{a\}.$$
  From the hypothesis (H.2) we obtain
  $$\mu\text{-}\mathsf{reach}(\mathsf{dom}(\Gamma_R, [\Gamma_S], \Gamma)) \setminus \{a\} = \mathsf{dom}(\Gamma_0, \Gamma_R, [\Gamma_S], \Gamma) \setminus \{a\}$$
  which equals to $\mathsf{dom}(\Gamma_0', \Gamma_R, \Gamma')$.
  4. Straightforward from hypothesis (H.2).

  From (C.1), (C.2) and (T-RUNNING PROCESS) we conclude (1). Item (2) holds trivially. 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-CHOICE) Trivial.
- (R-PARALLEL) In this case:
  - $P = P_1 \mid P_2$;
  - $\mu \fatsemi P_1 \to \mu' \fatsemi P_1'$;
  - $P' = P_1' \mid P_2$.

  According to Lemma B.4 we can assume that $\Gamma_0; \Gamma_R, [\Gamma_S], \Gamma \vdash_n \mu \fatsemi P$ was derived by rule (T-RUNNING PARALLEL). Then:
  - $\Gamma = \Gamma_1, \Gamma_2$ and $\Gamma_S = \Gamma_{S1}, \Gamma_{S2}$;
  - (P.1) $\Gamma_0; \Gamma_R, [\Gamma_{S2}], \Gamma_2; [\Gamma_{S1}], \Gamma_1 \vdash_n \mu \fatsemi P_1$;
  - (P.2) $\Gamma_0; \Gamma_R, [\Gamma_{S1}], \Gamma_1; [\Gamma_{S2}], \Gamma_2 \vdash_n \mu \fatsemi P_2$.

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

  (1') $\Gamma_0'; \Gamma_R, [\Gamma_{S2}], \Gamma_2; [\Gamma_{S1}], \Gamma_1' \vdash_n \mu' \fatsemi P_1'$, 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 B.3(1) we obtain (N.1) $\Gamma_0'; \Gamma_R, [\Gamma_S], \Gamma' \Vdash \mu'$. From (P.2), (N.1) and Lemma B.3(2) we deduce (P.2') $\Gamma_0'; \Gamma_R, [\Gamma_{S1}], \Gamma_1'; [\Gamma_{S2}], \Gamma_2 \vdash_n \mu' \fatsemi P_2$.
  From (1'), (P.2') and rule (T-RUNNING PARALLEL) we deduce (1).
  Regarding (2), notice that $\mathsf{unsealed}(\Gamma')$.
  Regarding (3), for every $\Gamma_J \subseteq \Gamma_R, [\Gamma_S]$ we have $\Gamma_I = \Gamma_J, \Gamma_2$ and we know $\Gamma_I \subseteq \Gamma_R, [\Gamma_S], \Gamma_2$, so we can conclude from (3') that if Γ_I is such that $\mu\text{-}\mathsf{balanced}(\mu\text{-}\mathsf{reach}(\mathsf{dom}(\Gamma_I, \Gamma_1)))$ we have
  $$\mu\text{-}\mathsf{reach}(\mathsf{dom}(\Gamma_R, \Gamma_S, \Gamma_2) \setminus \mathsf{dom}(\Gamma_I)) = \mu'\text{-}\mathsf{reach}(\mathsf{dom}(\Gamma_R, \Gamma_S, \Gamma_2) \setminus \mathsf{dom}(\Gamma_I))$$
  which is exactly (3).

- (R-SEND) In this case:
  - $P = a!\mathtt{m}(c).P'$;
  - $\mu = \mu'', a \mapsto [b, \mathfrak{Q}], b \mapsto [a, \mathfrak{Q}']$;
  - $\mu' = \mu'', a \mapsto [b, \mathfrak{Q}], b \mapsto [a, \mathfrak{Q}' :: \mathtt{m}(c)]$.

  From the hypothesis and rule (T-RUNNING PROCESS) we obtain:
  - (H.1) $[\Gamma_S], \Gamma \vdash_n a!\mathtt{m}(c).P'$;
  - (H.2) $\Gamma_0; \Gamma_R, [\Gamma_S], \Gamma \Vdash \mu'', a \mapsto [b, \mathfrak{Q}], b \mapsto [a, \mathfrak{Q}']$.

  From the hypothesis (H.1) and rule (T-SEND) for some $k \in I$ we obtain:
  - (L.1) $\Gamma = \Gamma'', a : \{!\mathtt{m}_i(S_i).T_i\}_{i \in I}, c : S_k$;
  - $\mathtt{m} = \mathtt{m}_k$;
  - $\|S_k\| < \infty$;
  - (C.1) $[\Gamma_S], \Gamma'', a : T_k \vdash_n P'$.

  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 4.3 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 occurs 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 $0 \le i \le p$ 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 we conclude
  $$\overline{T_k} = \mathsf{tail}(T_b, \mathtt{m}_1(S'_1) \cdots \mathtt{m}_p(S'_p)\mathtt{m}(S_k)).$$
  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
  $$\|S_k\| < \|\mathsf{tail}(T_b, \mathtt{m}_1(S'_1) \cdots \mathtt{m}_p(S'_p))\| \le \|T_b\| \le \|T_0\|$$
  therefore $c \ne b_0$. We conclude $b_0 \in \mathsf{dom}(\Gamma_R, \Gamma_S, \Gamma')$.
  4. Immediate from hypothesis (H.2).

  Item (2) holds trivially.
  Regarding (3), let $\Gamma_I \subseteq \Gamma_R, [\Gamma_S]$ such that μ-$\mathsf{balanced}(\mu$-$\mathsf{reach}(\mathsf{dom}(\Gamma_I, \Gamma)))$. From $a \in \mathsf{dom}(\Gamma)$ we deduce that $b \in \mu$-$\mathsf{reach}(\mathsf{dom}(\Gamma_I, \Gamma))$ and $c \preccurlyeq_{\mu'} b$, therefore μ-$\mathsf{reach}(\mathsf{dom}(\Gamma_R, \Gamma_S) \setminus \mathsf{dom}(\Gamma_I)) = \mu'$-$\mathsf{reach}(\mathsf{dom}(\Gamma_R, \Gamma_S) \setminus \mathsf{dom}(\Gamma_I))$.
- (R-RECEIVE) In this case:
  - $P = \sum_{i \in I} a?\mathtt{m}_i(x_i).P_i$;
  - $\mu = \mu'', a \mapsto [b, \mathtt{m}(c) :: \mathfrak{Q}]$ where $\mathfrak{Q} = \mathtt{m}_1(c_1) :: \cdots :: \mathtt{m}_p(c_p)$;
  - $\mathtt{m} = \mathtt{m}_k$ for some $k \in I$;
  - $P' = P_k\{c/x_k\}$;
  - $\mu' = \mu'', a \mapsto [b, \mathfrak{Q}]$.

  From the hypothesis and rule (T-RUNNING PROCESS) we obtain:
  - (H.1) $[\Gamma_S], \Gamma \vdash_n \sum_{i \in I} a?\mathtt{m}_i(x_i).P_i$;
  - (H.2) $\Gamma_0; \Gamma_R, [\Gamma_S], \Gamma \Vdash \mu'', a \mapsto [b, \mathtt{m}(c) :: \mathfrak{Q}]$ where $\mathfrak{Q} = \mathtt{m}_1(c_1) :: \cdots :: \mathtt{m}_p(c_p)$.

  From the hypothesis (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) $\Gamma'', a : T_k, x_k : S_k \vdash_n P_k$

  If we take $\Gamma_0 = \Gamma'_0, c : S_k$ and $\Gamma' = \Gamma'', a : T_k, c : S_k$ then from (N.1) and Lemma B.2 we conclude (C.1) $\Gamma' \vdash_n P_k\{c/x_k\}$.
  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 4.3 in order.
  1. If $a = b$ there is nothing to prove. Suppose $a \ne b$. 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 4.3 is satisfied.
  2. Suppose $a \ne b$ for otherwise there is nothing to prove. From hypothesis (H.2) we deduce $\Gamma_0, \Gamma_R, [\Gamma_S], \Gamma \vdash b : T_b$ and
  $$\begin{aligned}\overline{T_b} &= \mathsf{tail}(\{?\mathtt{m}_i(S_i).T_i\}_{i \in J}, \mathtt{m}(S_k)\mathtt{m}_1(S'_1) \cdots \mathtt{m}_p(S'_p)) \\ &= \mathsf{tail}(T_k, \mathtt{m}(S_k)\mathtt{m}_1(S'_1) \cdots \mathtt{m}_p(s'_p))\end{aligned}$$
  where $\Gamma_0, \Gamma_R, [\Gamma_S], \Gamma \vdash c_i : S'_i$ for $1 \le i \le p$.
  3. Straightforward by definition of Γ'_0 and Γ'.
  4. Immediate from hypothesis (H.2).

  From (C.1), (C.2) and rule (T-RUNNING PROCESS) we conclude $\Gamma'_0; \Gamma_R, [\Gamma_S], \Gamma' \vdash_n \mu' \mathbin{⨟} P'$.
  Regarding (2), from (H.2) and condition (2) of Definition 4.3 we know that $\mathsf{rank}(S_k) = 0$.
  Regrading (3), from $c \in \mathsf{dom}(\Gamma')$ we deduce μ-$\mathsf{reach}(\mathsf{dom}(\Gamma_R, \Gamma_S))) = \mu'$-$\mathsf{reach}(\mathsf{dom}(\Gamma_R, \Gamma_S))$.
- (R-START TRANSACTION) In this case:
  - $P = \prod_{i \in I} \mathtt{try}(A_i)\ \{Q_i\} P_i$;
  - $P' = \langle \bigcup_{i \in I} A_i, \emptyset, \{\prod_{i \in I} Q_i\} \prod_{i \in I} P_i \rangle$;

  - $\mu' = \mu$;
  - μ-balanced$(\bigcup_{i\in I} A_i)$.

  According to Lemma B.4 we can assume that $\Gamma_0;\Gamma_R,[\Gamma_S],\Gamma \vdash_n \mu \mathbin{;} P$ 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) $\Gamma_0;\Gamma_R,\bigsqcup_{j\in I\setminus\{i\}}[\Gamma_{Sj}],\bigsqcup_{j\in I\setminus\{i\}}\Gamma_j;[\Gamma_{Si}],\Gamma_i \vdash_n \mu \mathbin{;} \texttt{try}(A_i)\ \{Q_i\}P_i$ for every $i \in I$.

  From (L.1), (P.i) and rule (T-RUNNING PROCESS) we obtain:
  - (H.1) $[\Gamma_{Si}],\Gamma_i \vdash_n \texttt{try}(A_i)\ \{Q_i\}P_i$ where unsealed(Γ_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) $\Gamma_i = \Gamma'_i,\{a:\{S_a\}[\![T_a\}_{a\in A_i}$;
  - $[[\Gamma_{Si}],\Gamma'_i],\{a:T_a\}_{a\in A_i} \vdash_{n+1} P_i$;
  - $[\Gamma_{Si}],\Gamma'_i,\{a:S_a\}_{a\in A_i} \vdash_n Q_i$;

  By rule (T-PARALLEL) we obtain:
  - (P.1) $[[\Gamma_S],\bigsqcup_{i\in I}\Gamma'_i],\{a:T_a\}_{i\in I,a\in A_i} \vdash_{n+1} \prod_{i\in I} P_i$;
  - (P.2) $[\Gamma_S],\bigsqcup_{i\in I}\Gamma'_i,\{a:S_a\}_{i\in I,a\in A_i} \vdash_n \prod_{i\in I} Q_i$.

  It is easy to see that the queue associated with a is empty for every $i\in I$ and $a \in A_i$ since otherwise would contradict (L.2). Hence, from (H.2) and (L.2) and the fact that μ-balanced$(\bigcup_{i\in I}A_i)$ we have μ-balanced$(\{a:S_a\}_{i\in I,a\in A_i})$ and $\bigcup_{i\in I}A_i = \mu$-reach$(\bigcup_{i\in I}A_i)$. Also, from (H.2) we have
  - (H.1') $\Gamma_0;\Gamma_R,[[\Gamma_S],\bigsqcup_{i\in I}\Gamma'_i],\{a:T_a\}_{i\in I,a\in A_i} \Vdash \mu$;

  From (H.1'), (P.1) and rule (T-RUNNING PROCESS) we obtain
  - (T.1) $[[\Gamma_S],\bigsqcup_{i\in I}\Gamma'_i],\{a:T_a\}_{i\in I,a\in A_i} \vdash_{n+1} \prod_{i\in I} P_i$.

  Let $\Gamma'_0 = \Gamma_0$ and $\Gamma' = \bigsqcup_{i\in I}\Gamma'_i,\{a:\{T\}_a\}_{i\in I,a\in A_i}$. From rule (T-RUNNING TRANSACTION), (T.1), (P.2) and facts proven above we obtain (1).
  We conclude by observing that item (2) is trivial and (3) holds since $\mu' = \mu$.
- (R-END TRANSACTION) In this case:
  - $P = \langle A,B,\{Q\}\prod_{i\in I}\texttt{commit}(A_i).P_i\rangle$;
  - $P' = \prod_{i\in I} P_i$;
  - $\mu' = \mu$.

  From rule (T-RUNNING TRANSACTION) we obtain:
  - $\Gamma = \Gamma_1,\{a:\{S_a\}T_a\}_{a\in A},\Gamma_2$;
  - (T.1) $\Gamma_0;\Gamma_R;[[\Gamma_S],\Gamma_1],\{a:T_a\}_{a\in A},\Gamma_2 \vdash_{n+1} \prod_{i\in I}\texttt{commit}(A_i).P_i$;
  - μ-balanced$(\{a:S_a\}_{a\in A})$;
  - μ-balanced(B);
  - local(Γ_2);
  - $A\cup B = \mu$-reach$(A\cup \mathsf{dom}(\Gamma_2))$.

  Then, from (T.1) and rules (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 $\{a:T_a\} = \bigsqcup_{i\in I}\{a:T_a\}_{a\in B_i}$ and $\Gamma_2 = \bigsqcup_{i\in I}\Gamma_{2i}$ where rank$(\Gamma_{2i}) = 0$ for every $i\in I$;
  - $[[\Gamma_{Si}],\Gamma_{1i}],\{a:T_a\}_{a\in B_i},\Gamma_{2i} \vdash_{n+1} \texttt{commit}(A_i).P_i$ for every $i \in I$.

  From rule (T-COMMIT) we deduce:
  - $B_i = A_i$;
  - $T_a =]\!]T'_a$ for every $i\in I$ and $a \in A_i$;
  - $[\Gamma_{Si}],\Gamma_{1i},\{a:T'_a\}_{a\in A_i},\Gamma_{2i} \vdash_n P_i$ for every $i\in I$.

  From rule (T-PARALLEL) we deduce (C.1) $[\Gamma_S],\Gamma_1,\{a:T'_a\}_{i\in I,a\in A_i},\Gamma_2 \vdash_n P'$.
  Let $\Gamma'_0 = \Gamma_0$ and $\Gamma' = \Gamma_1,\{a:T'_a\}_{i\in I,a\in A_i},\Gamma_2$. From $\Gamma_0;\Gamma_R;[\Gamma_S],\Gamma \vdash_n \mu \mathbin{;} P$ and Lemma B.3(1) we obtain (H.1) $\Gamma_0;\Gamma_R,[\Gamma_S],\Gamma \Vdash \mu$. From the fact that the queues associated with pointers a are empty, since $\{a:\{S_a\}]\!]T'_a\}$, and (H.1) we deduce (C.2) $\Gamma_0;\Gamma_R,[\Gamma_S],\Gamma' \Vdash \mu'$.
  From (C.1), (C.2) and rule (T-RUNNING PROCESS) we deduce (1).
  We observe that (2) holds since rank$(\{S_a\}T_a) = $ rank(T'_a) for $a\in A$ and (3) holds trivially.
- (R-RUN TRANSACTION) In this case:
  - $P = \langle A,B,\{R\}Q\rangle$;
  - $P' = \langle A,B',\{R\}Q'\rangle$ where $B' = (B\cup(\mathsf{dom}(\mu')\setminus\mathsf{dom}(\mu)))\setminus(\mathsf{dom}(\mu)\setminus\mathsf{dom}(\mu'))$;
  - $\mu \mathbin{;} Q \to \mu' \mathbin{;} Q'$;

  Let $A = \bigcup_{i\in I}\{a_i\}$. From rule (T-RUNNING TRANSACTION) we deduce:
  - $\Gamma = \Gamma_1,\{a_i:\{S_i\}T_i\}_{i\in I},\Gamma_2$;
  - (T.1) $\Gamma_0;\Gamma_R;[[\Gamma_S],\Gamma_1],\{a_i:T_i\}_{i\in I},\Gamma_2 \vdash_{n+1} \mu \mathbin{;} Q$;
  - (T.2) $\Gamma_1,\{a_i:S_i\}_{i\in I} \vdash_n R$;
  - (T.3) μ-balanced$(\{a_i:S_i\}_{i\in I})$;
  - (T.4) μ-balanced(B);
  - (T.5) local(Γ_2);
  - (T.6) $\{a_i\}_{i\in I}\cup B = \mu$-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 unsealed(Γ) by induction hypothesis we obtain that there exist Γ'_0 and Γ'_3 such that:

(1') $\Gamma'_0; \Gamma_R; [[\Gamma_S], \Gamma_1], \Gamma'_3 \vdash_{n+1} \mu' \,⨾\, Q'$, 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 $i \in I$ we know that all the a_i's are still in the environment for Q' and we have $\Gamma'_3 = \{a : T'_i\}_{i \in I}, \Gamma'_2$.
Let $\Gamma' = \Gamma_1, \{a_i : \{S_i\}T'_i\}_{i \in I}, \Gamma'_2$.
Regarding (1), from (T.4) and Proposition B.3 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') $\{a_i\}_{i \in I} \cup B' = \mu\text{-}\mathsf{reach}(\{a_i\}_{i \in I} \cup \mathsf{dom}(\Gamma'_2))$ we will use following two sequences of equalities:

$$
\begin{aligned}
(*)\ \mathsf{dom}(\mu') &= (\mathsf{dom}(\mu) \cup (B' \setminus B)) \setminus (B \setminus B') && \text{from definition of } B' \\
&= (\mu\text{-}\mathsf{reach}(\mathsf{dom}(\Gamma_R, [\Gamma_S], \Gamma_1, \Gamma_2) \cup \{a_i\}_{i \in I}) \cup (B' \setminus B)) \setminus (B \setminus B') && \text{from item (3) of Definition 4.3} \\
&= (\mu\text{-}\mathsf{reach}(\mathsf{dom}(\Gamma_R, [\Gamma_S], \Gamma_1)) \uplus \mu\text{-}\mathsf{reach}(\mathsf{dom}(\Gamma_2) \cup \{a_i\}_{i \in I}) \cup (B' \setminus B)) \setminus (B \setminus B') && \text{from item (4) of Definition 4.3} \\
&= (\mu\text{-}\mathsf{reach}(\mathsf{dom}(\Gamma_R, [\Gamma_S], \Gamma_1)) \cup \{a_i\}_{i \in I} \cup B \cup (B' \setminus B)) \setminus (B \setminus B') && \text{from (T.6)} \\
&= \mu\text{-}\mathsf{reach}(\mathsf{dom}(\Gamma_R, [\Gamma_S], \Gamma_1)) \cup \{a_i\}_{i \in I} \cup B'
\end{aligned}
$$

From (1') and Lemma B.3(1) we obtain $\Gamma'_0; \Gamma_R; [[\Gamma_S], \Gamma_1], \Gamma'_3 \Vdash \mu'$, so we have:

$$
\begin{aligned}
(**)\ \mathsf{dom}(\mu') &= \mu'\text{-}\mathsf{reach}(\mathsf{dom}(\Gamma_R, [\Gamma_S], \Gamma_1, \Gamma'_2) \cup \{a_i\}_{i \in I}) && \text{from item (3) of Definition 4.3} \\
&= \mu'\text{-}\mathsf{reach}(\mathsf{dom}(\Gamma_R, [\Gamma_S], \Gamma_1)) \uplus \mu'\text{-}\mathsf{reach}(\mathsf{dom}(\Gamma'_2) \cup \{a_i\}_{i \in I}) && \text{from item (4) of Definition 4.3.}
\end{aligned}
$$

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, if we take $\Gamma_I = \emptyset$ 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 (T.1), (T.2), (T.3), (T.4'), (T.5') and (T.6').
From (2') and rule (WF-RUN) of Definition 5 we obtain (2).
Regarding (3), for every $\Gamma_J \subseteq \Gamma_R, [\Gamma_S]$ we have $\Gamma_I = \Gamma_J, [\Gamma_1]$ and we know $\Gamma_I \subseteq \Gamma_R, [[\Gamma_S], \Gamma_1]$, so we can conclude from (3') that if Γ_I is such that $\mu\text{-}\mathsf{balanced}(\mu\text{-}\mathsf{reach}(\mathsf{dom}(\Gamma_I, \Gamma_2) \cup \{a_i\}_{i \in I}))$ we have

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

which is exactly (3).

- (R-ABORT TRANSACTION) In this case:
  - $P = \langle \{a_i\}_{i \in I}, \mathsf{dom}(\mu_2), \{P'\}(\texttt{throw} \mid P'') \rangle$;
  - $\mu = \mu_1, \{a_i \mapsto [b_i, \mathfrak{Q}_i]\}_{i \in I}, \mu_2$;
  - $\mu' = \mu_1, \{a_i \mapsto [b_i, \varepsilon]\}_{i \in I}$;

  From rule (T-RUNNING TRANSACTION) we deduce:
  - (L.1) $\Gamma = \Gamma_1, \{a_i : \{S_i\}T_i\}_{i \in I}, \Gamma_2$;
  - (C.1) $[\Gamma_S], \Gamma_1, \{a_i : S_i\}_{i \in I} \vdash P'$.
  - (T.1) $\mu\text{-}\mathsf{balanced}(\{a_i : S_i\}_{i \in I})$;
  - (T.2) $\mathsf{local}(\Gamma_2)$;
  - (T.3) $\{a_i\}_{i \in I} \cup \mathsf{dom}(\mu_2) = \mu\text{-}\mathsf{reach}(\{a_i\}_{i \in I} \cup \mathsf{dom}(\Gamma_2))$;

  Let $\Gamma'_0 = \Gamma_0 \setminus \mathsf{dom}(\mu_2)$ and $\Gamma' = \Gamma_1, \{a_i : S_i\}_{i \in I}$. We only have to show that (C.2) $\Gamma'_0; \Gamma_R, [\Gamma_S], \Gamma' \Vdash \mu'$ and we prove the items of Definition 4.3. 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. Regarding item (3), we have to show that $\mathsf{dom}(\mu') = \mathsf{dom}(\Gamma'_0, \Gamma_R, [\Gamma_S], \Gamma') = \mu'\text{-}\mathsf{reach}(\mathsf{dom}(\Gamma_R, [\Gamma_S], \Gamma'))$. The first equality is easy. Regarding the second equality, we derive:

$$
\begin{aligned}
\mathsf{dom}(\mu) &= \mu\text{-}\mathsf{reach}(\{a_i\}_{i \in I} \uplus \mathsf{dom}(\Gamma_R, \Gamma_1, \Gamma_2)) && \text{from item (3) of Definition 4.3} \\
&= \mu\text{-}\mathsf{reach}(\mathsf{dom}(\Gamma_R, \Gamma_1)) \uplus \mu\text{-}\mathsf{reach}(\{a_i\}_{i \in I} \cup \mathsf{dom}(\Gamma_2)) && \text{from item (4) of Definition 4.3} \\
&= \mu\text{-}\mathsf{reach}(\mathsf{dom}(\Gamma_R, \Gamma_1)) \uplus \{a_i\}_{i \in I} \uplus \mathsf{dom}(\mu_2) && \text{from (T.3)} \\
&= \mathsf{dom}(\mu_1) \uplus \{a_i\}_{i \in I} \uplus \mathsf{dom}(\mu_2) && \text{by definition of } \mu
\end{aligned}
$$

  where we write $\uplus$ for disjoint union. From the last equality we deduce
  - (*) $\mathsf{dom}(\mu_1) = \mu\text{-}\mathsf{reach}(\mathsf{dom}(\Gamma_R, [\Gamma_S], \Gamma_1)) = \mu_1\text{-}\mathsf{reach}(\mathsf{dom}(\Gamma_R, [\Gamma_S], \Gamma_1))$

  and now we conclude

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

  From (C.1), (C.2) and rule (T-RUNNING PROCESS) we obtain (1).
  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.
  (3) holds trivially.
- (R-INVOKE) Trivial.
- (R-STRUCT) Follows from Lemma B.1 and induction. □

B.2 Type soundness

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

Proposition B.4. *If* $\Gamma \vdash_n P$*, then* $\mathsf{fn}(P) \subseteq \mathsf{dom}(\Gamma)$.

Proof. A simple induction on the derivation of $\Gamma \vdash_n P$. □

Lemma B.6. *If* $\Gamma \vdash_0 P$ *and* $\mathsf{rank}(\Gamma) = 0$*, then* $\mathsf{fn}(P) = \mathsf{dom}(\Gamma)$.

Proof. By induction on the derivation of $\Gamma \vdash_0 P$ and by cases on the last rule applied.

- (T-INACTION) Then $P = \mathtt{done}$. From the hypotheses $\Gamma \vdash_0 \mathtt{done}$ we conclude that $\Gamma = \emptyset$.
- (T-THROW) This case is impossible since `throw` processes are well typed only at nesting greater than 0.
- (T-INVOKE) Trivial.
- (T-OPEN) Then $P = \mathtt{open}(a,b).P'$ and $\Gamma, a : T, b : \overline{T} \vdash_0 P'$. By induction hypothesis we obtain $\mathsf{fn}(P') = \mathsf{dom}(\Gamma, a : T, b : \overline{T})$ and then we conclude $\mathsf{fn}(P) = \mathsf{fn}(P') \setminus \{a,b\} = \mathsf{dom}(\Gamma, a : T, b : \overline{T}) \setminus \{a,b\} = \mathsf{dom}(\Gamma)$.
- (T-CLOSE) Then $P = \mathtt{close}(a).P'$, $\Gamma = \Gamma', u : \mathtt{end}$ and $\Gamma' \vdash_0 P'$. By induction hypothesis we obtain $\mathsf{fn}(P') = \mathsf{dom}(\Gamma')$ and then we conclude $\mathsf{fn}(P) = \{u\} \cup \mathsf{fn}(P') = \{u\} \cup \mathsf{dom}(\Gamma') = \mathsf{dom}(\Gamma)$.
- (T-SEND) Then $P = u!\mathtt{m}_k(v).P'$, $\Gamma = \Gamma', u : \{!\mathtt{m}_i(S_i).T_i\}_{i\in I}, v : S_k$ and $\Gamma', u : T_k \vdash_0 P'$ for $k \in I$. By induction hypothesis we obtain $\mathsf{fn}(P') = \mathsf{dom}(\Gamma', u : T_k)$ and then we conclude $\mathsf{fn}(P) = \{u,v\} \cup \mathsf{fn}(P') = \{u,v\} \cup \mathsf{dom}(\Gamma', u : T_k) = \mathsf{dom}(\Gamma)$.
- (T-RECEIVE) Then $P = \sum_{i\in I} u?\mathtt{m}_i(x_i).P_i$, $\Gamma = \Gamma', u : \{?\mathtt{m}_i(S_i).T_i\}_{i\in I}$ and for all $i \in I$ we have $\Gamma', u : T_i, x_i : S_i \vdash_0 P_i$. By induction hypothesis for all $i \in I$ we obtain $\mathsf{fn}(P_i) = \mathsf{dom}(\Gamma', u : T_i, x_i : S_i)$ and then we conclude $\mathsf{fn}(P) = \{u\} \cup \bigcup_{i\in I}(\mathsf{fn}(P_i) \setminus \{x_i\}) = \{u,\} \cup \bigcup_{i\in I}(\mathsf{dom}(\Gamma', u : T_i, x_i : S_i)) \setminus \{x_i\}) = \mathsf{dom}(\Gamma)$.
- (T-CHOICE) Then $P = P_1 \oplus P_2$ and $\Gamma \vdash_0 P_i$ for $i \in \{1,2\}$. By induction hypothesis we obtain $\mathsf{fn}(P_i) = \mathsf{dom}(\Gamma)$ for $i \in \{1,2\}$ and then we conclude $\mathsf{fn}(P) = \mathsf{fn}(P_1) \cup \mathsf{fn}(P_2) = \mathsf{dom}(\Gamma)$.
- (T-PARALLEL) Then $P = P_1 \mid P_2$, $\Gamma = \Gamma_1, \Gamma_2$ and $\Gamma_i \vdash_0 P_i$ for $i \in \{1,2\}$. By induction hypothesis we obtain $\mathsf{fn}(P_i) = \mathsf{dom}(\Gamma_i)$ for $i \in \{1,2\}$ and then we conclude $\mathsf{fn}(P) = \mathsf{fn}(P_1) \cup \mathsf{fn}(P_2) = \mathsf{dom}(\Gamma_1) \cup \mathsf{dom}(\Gamma_2) = \mathsf{dom}(\Gamma)$.
- (T-TRY) Then $P = \mathtt{try}(\{u_i\}_{i\in I})\ \{R\}Q$, $\Gamma = \Gamma', \{u_i : \{S_i\}[\![T_i]\!]_{i\in I}$ and $[\Gamma'], \{u_i : T_i\}_{i\in I} \vdash_1 Q$ and $\Gamma, \{u_i : S_i\}_{i\in I} \vdash_0 R$. By Proposition B.4 applied to Q we deduce $\mathsf{fn}(Q) \subseteq \mathsf{dom}(\Gamma') \cup \{u_i\}_{i\in I}$. By induction hypothesis on R we deduce $\mathsf{fn}(R) = \mathsf{dom}(\Gamma') \cup \{u_i\}_{i\in I}$. We conclude by noting that $\mathsf{fn}(P) = \{u_i\}_{i\in I} \cup \mathsf{fn}(Q) \cup \mathsf{fn}(R) = \mathsf{dom}(\Gamma') \cup \{u_i\}_{i\in I} = \mathsf{dom}(\Gamma)$.
- (T-COMMIT) This case is impossible since commit processes are well typed only at nesting greater than 0. □

Proposition B.5. *If* $\Gamma_0; \Gamma_R; \Gamma \vdash_n \mu \mathbin{⨾} P$*, then* $\mu\text{-}\mathsf{reach}(\mathsf{fn}(P)) \subseteq \mu\text{-}\mathsf{reach}(\mathsf{dom}(\Gamma))$.

Proof. Straightforward consequence of Proposition B.4 and the fact that μ-reach is a closure. □

Lemma B.7. *If* $\Gamma_0; \Gamma_R; \Gamma \vdash_0 \mu \mathbin{⨾} P$ *and* $\mathsf{rank}(\Gamma) = 0$*, then* $\mu\text{-}\mathsf{reach}(\mathsf{fn}(P)) = \mu\text{-}\mathsf{reach}(\mathsf{dom}(\Gamma))$.

Proof. By induction on the derivation of $\Gamma_0; \Gamma_R; \Gamma \vdash_0 \mu \mathbin{⨾} P$ and by cases on the last rule applied.

- (T-RUNNING PROCESS) Then $\Gamma \vdash_0 P$. From Lemma B.6 we deduce $\mathsf{fn}(P) = \mathsf{dom}(\Gamma)$, from which we conclude immediately $\mu\text{-}\mathsf{reach}(\mathsf{fn}(P)) = \mu\text{-}\mathsf{reach}(\mathsf{dom}(\Gamma))$.
- (T-RUNNING PARALLEL) Then $\Gamma = \Gamma_1, \Gamma_2$ and $P = P_1 \mid P_2$ and $\Gamma_0; \Gamma_R, \Gamma_{3-i}; \Gamma_i \vdash_0 P_i$ for $i \in \{1,2\}$. From the hypothesis $\mathsf{rank}(\Gamma) = 0$ we deduce $\mathsf{rank}(\Gamma_i) = 0$ for $i \in \{1,2\}$. By induction hypothesis we deduce $\mu\text{-}\mathsf{reach}(\mathsf{fn}(P_i)) = \mu\text{-}\mathsf{reach}(\mathsf{dom}(\Gamma_i))$. From the fact that the heap is well typed (Lemma B.3) and item (4) of Definition 4.3, we conclude $\mu\text{-}\mathsf{reach}(\mathsf{fn}(P)) = \mu\text{-}\mathsf{reach}(\mathsf{fn}(P_1) \cup \mathsf{fn}(P_2)) = \mu\text{-}\mathsf{reach}(\mathsf{fn}(P_1)) \cup \mu\text{-}\mathsf{reach}(\mathsf{fn}(P_2)) = \mu\text{-}\mathsf{reach}(\mathsf{dom}(\Gamma_1)) \cup \mu\text{-}\mathsf{reach}(\mathsf{dom}(\Gamma_2)) = \mu\text{-}\mathsf{reach}(\mathsf{dom}(\Gamma_1) \cup \mathsf{dom}(\Gamma_2)) = \mu\text{-}\mathsf{reach}(\mathsf{dom}(\Gamma))$.
- (T-RUNNING TRANSACTION) Then:
  - $P = \langle \{a_i\}_{i\in I}, B, \{R\}Q \rangle$;
  - $\Gamma = \Gamma_1, \{a_i : \{S_i\}T_i\}_{i\in I}, \Gamma_2$;
  - $\{a_i\}_{i\in I} \cup B = \mu\text{-}\mathsf{reach}(\{a_i\}_{i\in I} \cup \mathsf{dom}(\Gamma_2))$;
  - $\Gamma_0; \Gamma_R; [\Gamma_1], \{a_i : T_i\}_{i\in I}, \Gamma_2 \vdash_1 \mu \mathbin{⨾} Q$;
  - $\Gamma_1, \{a_i : S_i\}_{i\in I} \vdash_0 R$.

  From the hypothesis $\mathsf{rank}(\Gamma) = 0$ we deduce $\mathsf{rank}(\Gamma_1, \{a_i : S_i\}_{i\in I}) = 0$. From Proposition B.5 and Lemma B.6 we obtain:
  - $\mu\text{-}\mathsf{reach}(\mathsf{fn}(Q)) \subseteq \mu\text{-}\mathsf{reach}(\{a_i\}_{i\in I} \cup \mathsf{dom}([\Gamma_1], \Gamma_2))$;
  - $\mathsf{fn}(R) = \{a_i\}_{i\in I} \cup \mathsf{dom}(\Gamma_1)$.

  We conclude $\mu\text{-}\mathsf{reach}(\mathsf{fn}(P)) = \mu\text{-}\mathsf{reach}(\{a_i\}_{i\in I} \cup B \cup \mathsf{fn}(Q) \cup \mathsf{fn}(R)) = \mu\text{-}\mathsf{reach}(\mathsf{dom}(\Gamma_1) \cup \mathsf{dom}(\Gamma_2) \cup \{a_i\}_{i\in I}) = \mu\text{-}\mathsf{reach}(\mathsf{dom}(\Gamma))$. □

We conclude with the soundness proof of the type system.

Theorem B.1 (Theorem 4.2)**.** *Let* $\vdash_0 P$*. Then* P *is well behaved.*

Proof. From the hypothesis we deduce that $\emptyset; \emptyset; \emptyset \vdash_0 \emptyset \mathbin{⨾} P$. Consider a derivation $\emptyset \mathbin{⨾} P \Rightarrow \mu \mathbin{⨾} Q$. From Lemma B.5 we deduce that there exist Γ_0 and Γ such that $\Gamma_0; \emptyset; \Gamma \vdash_0 \mu \mathbin{⨾} Q$ and $\mathsf{rank}(\Gamma) = 0$ and, from Lemma B.3, we obtain $\Gamma_0; \Gamma \Vdash \mu$.

Regarding condition (1) of Definition 3.5, then from Definition 4.3 we know $\mathsf{dom}(\mu) = \mu\text{-}\mathsf{reach}(\mathsf{dom}(\Gamma))$. By Lemma B.7 we conclude $\mu\text{-}\mathsf{reach}(\mathsf{dom}(\Gamma)) = \mu\text{-}\mathsf{reach}(\mathsf{fn}(Q))$.

Regarding condition (2) of Definition 3.5, suppose that $Q \equiv Q_1 \mid Q_2$ and $\mu \mathbin{⨾} Q_1 \not\rightarrow$ and $Q_1 \not\equiv \mathtt{throw} \mid Q_1'$ (the last hypothesis being granted by the fact that Q is well typed at nesting 0). We prove $\mu \mathbin{⨾} Q_1 \downarrow$ by induction on Q_1.

- ($Q_1 = \mathtt{done}$) We conclude with an application of rule (ST-IDLE).

- ($Q_1 = \texttt{open}(a,b).R$) Without loss of generality we may assume $a,b \notin \mathsf{dom}(\mu)$ and now $\mu \mathbin{⨾} Q_1 \rightarrow$ which contradicts the hypothesis, therefore this case is impossible.
- ($Q_1 = \texttt{close}(a).R$) From Proposition B.4 and Definition 4.3 we deduce $a \in \mathsf{dom}(\Gamma) \subseteq \mathsf{dom}(\mu)$, and now $\mu \mathbin{⨾} Q_1 \rightarrow$ which contradicts the hypothesis, therefore this case is impossible.
- ($Q_1 = R_1 \oplus R_2$) This case is impossible because $\mu \mathbin{⨾} R_1 \oplus R_2$ 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 $a \in \mathsf{dom}(\mu)$. From item (2) of Definition 4.3 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 $\mu \mathbin{⨾} Q_1 \rightarrow$ 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 $\mu \mathbin{⨾} \sum_{i\in I} a?\mathtt{m}_i(x_i).R_i$ does not reduce is when $\mathfrak{Q} = \varepsilon$, therefore we conclude $\mu \mathbin{⨾} Q_1 \downarrow$ by an application of rule (ST-INPUT).
- ($Q_1 = R_1 \mid R_2$) From the hypothesis $\mu \mathbin{⨾} Q_1 \not\rightarrow$ we deduce $\mu \mathbin{⨾} R_i \not\rightarrow$ for $i \in \{1,2\}$. From the hypothesis $Q_1 \not\equiv \texttt{throw} \mid Q_1'$ we deduce $R_i \not\equiv \texttt{throw} \mid R_i'$ for $i \in \{1,2\}$. By induction hypothesis we obtain $\mu \mathbin{⨾} R_i \downarrow$ for $i \in \{1,2\}$. We conclude with an application of rule (ST-PARALLEL).
- ($Q_1 = \texttt{try}(A)\ \{R_2\}R_1$) From the hypothesis $\mu \mathbin{⨾} Q_1 \not\rightarrow$ we deduce $\neg\mu\text{-}\mathsf{balanced}(A)$. We conclude with an application of rule (ST-TRY).
- ($Q_1 = \texttt{throw}$) 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 $X(\tilde{u}) \overset{\text{def}}{=} R$ is a definition and $\tilde{a}$ and $\tilde{u}$ have the same length. Then $\mu \mathbin{⨾} X\langle\tilde{a}\rangle \rightarrow$ which contradicts the hypothesis, therefore this case is impossible.
- ($Q_1 = \mu \mathbin{⨾} \langle A, B, \{R_2\}R_1\rangle$) From the hypothesis $\mu \mathbin{⨾} Q_1 \not\rightarrow$ we deduce $\mu \mathbin{⨾} R_1 \not\rightarrow$ and $R_1 \not\equiv \prod_{i\in I} \texttt{commit}(A_i).R_i$ and $R_1 \not\equiv \texttt{throw} \mid R_1'$ since these are the cases when $\mu \mathbin{⨾} Q_1$ does reduce. By induction hypothesis we deduce $\mu \mathbin{⨾} R_1 \downarrow$ and therefore we conclude with an application of rule (ST-RUNNING TRANSACTION). □