Polymorphic Endpoint Types for Copyless Message Passing

Viviana Bono   Luca Padovani

Dipartimento di Informatica, Università di Torino, Italy

We present PolySing♯, a calculus that models process interaction based on copyless message passing, in the style of Singularity OS. We equip the calculus with a type system that accommodates polymorphic endpoint types, which are a variant of polymorphic session types, and we show that well-typed processes are free from faults, leaks, and communication errors. The type system is essentially linear, although linearity alone may leave room for scenarios where well-typed processes leak memory. We identify a condition on endpoint types that prevents these leaks from occurring.

1 Introduction

Singularity OS is the prototype of a dependable operating system where processes share the same address space and interact with each other solely by message passing over asynchronous channels. The overhead of communication-based interactions is tamed by copyless message passing: only *pointers* to messages are physically transferred from one process to another. Static analysis of Singularity processes guarantees *process isolation*, namely that every process can only access memory it owns exclusively.

In [2] we presented CoreSing♯, a formalization of the core features of Sing♯ – the language used for the implementation of Singularity OS – along with a type system ensuring that well-typed processes are free from communication errors, memory faults, and memory leaks. At first sight it might seem that these properties can be trivially enforced through a linear type system based on session types [10, 11]. However, in [2] we remarked how linearity alone can be too restrictive in some contexts and too permissive in others. To illustrate why linearity can be too restrictive, consider the code fragment

```
expose (a) { send(b, arg, *a); *a := new T(); }
```

which dereferences the pointer `a` and sends `*a` on the endpoint `b`. Linearity is violated right after the `send(arg, b, *a)` command, since `*a` is owned both by the sender (indirectly, through `a`) as well as by the receiver. The construct **expose** is used by the Sing♯ compiler to keep track of memory ownership. In particular, Sing♯ allows pointer dereferentiation only within **expose** blocks. The semantics of an **expose**(a) block is to temporarily transfer the ownership of `*a` from `a` to the process exposing the pointer. If the process still owns `*a` at the end of the **expose** block, the construct is well typed. In [2] we showed that all we need to capture the static semantics of **expose**(a) blocks is to distinguish cells with type $*t$ (whose content, of type t, is owned by the cell) from cells with type $*\bullet$ (whose content is owned directly by the process). At the beginning of the **expose** block, `a` is accessed and its type turns from $*t$ to $*\bullet$; within the block it is possible to (linearly) use `*a`; at the end of the block, `*a` is assigned with the pointer to a newly allocated object that the process owns, thus turning `a`'s type from $*\bullet$ back to some $*s$.

An example where linearity can be too permissive is given by the code fragment

$$(e, f) := \textbf{open}(); \textbf{send}(e, arg, f); \textbf{close}(e); \qquad (1)$$

which creates the two endpoints `e` and `f` of a channel, sends `f` as the argument of an `arg`-tagged message on `e`, and closes `e`. This code uses `e` and `f` linearly and is well typed by associating `e` and `f` with suitable

endpoint types T and S, where $T = !\texttt{arg}(S).\textbf{end}$ and $S = \mu\alpha.?\texttt{arg}(\alpha).\textbf{end}$. Observe that the code fragment (1) uses e in accordance with type T and that T and S are *dual endpoint types* (they describe complementary actions). Yet, the code generates a memory leak: after the **close** instruction, no reference to f is available, therefore the arg-tagged message will never be received and f will never be deallocated. In [2] we have shown that the leak produced by this code originates from the recursive type S and can be avoided by imposing a simple restriction on endpoint types. The idea is to define a notion of *weight* for endpoint types which roughly gives the "depth" of the message queues in the endpoints having those types and to restrict endpoint types to those having finite weight. Then, one can show that the code (1) is well typed only if endpoint types with infinite weight are allowed.

In this work we present PolySing♯, a variant of CoreSing♯ where we add *bounded polymorphism* to endpoint types, along the lines of [8], while preserving all the properties mentioned earlier. For instance, the polymorphic endpoint type $!\texttt{m}\langle\alpha\rangle(\alpha).?\texttt{m}(\alpha).\textbf{end}$ denotes an endpoint on which it is possible to send an m-tagged message with an argument of any type, and then receive another m-tagged message whose argument has the same type as that of the first message. It is possible to specify bounds for type variables, like in $!\texttt{m}\langle\alpha \leqslant t\rangle(\alpha).?\texttt{m}(\alpha).\textbf{end}$, to denote that the type variable α ranges over any *subtype* of t, and to recover unconstrained polymorphism by devising a type Top that is supertype of any other type.

Now, it may come as a surprise that, when polymorphic endpoint types are allowed, the code fragment (1) can be declared well typed without resorting to recursive types by taking $T = !\texttt{arg}\langle\alpha\rangle(\alpha).\textbf{end}$ and $S = ?\texttt{arg}\langle\alpha\rangle(\alpha).\textbf{end}$. Since the type T of e is polymorphic, e accepts a message with an argument of *any* type and, in particular, a message with argument f. Fortunately, a smooth extension of the finite-weight restriction we have introduced in the monomorphic case rules out this problematic example. The idea is to estimate the weight of type variables by looking at their bound. In T and S above, the type variable α has no bound (or, to be precise, is bounded by the top type Top) and therefore is given infinite weight. When α occurs in a constraint $\alpha \leqslant t$, we estimate the weight of α to be the same as the weight of t. The estimation relies on a fundamental relationship between weights and subtyping, whereby the weight of s is always smaller than or equal to the weight of t if $s \leqslant t$. We show that forbidding the output of messages whose argument has a type with infinite weight allows us to prove the absence of leaks, together with the other desired properties. Our work is the first to formalize polymorphic endpoint types, which are effectively described as a feature of Singularity contracts [13] even though, to the best of our knowledge, they never made it to the prototype implementation of Singularity. Also, the availability of polymorphic endpoint types allows us to encode linear mutable cells. This renders the $*t$ and $*\bullet$ types superfluous and makes our model even more essential with respect to the one presented in [2].

The rest of the paper is organized thus: Section 2 presents an example to introduce all the fundamental concepts (endpoint types, subtyping, bounded polymorphism) of this work; in Section 3 we define the syntax and operational semantics of PolySing♯ and we formalize the notion of well-behaved process as a process where faults, leaks, and communication errors do not occur; Section 4 defines the type system for PolySing♯ and presents a soundness result (well-typed processes are well behaved). We discuss related work in Section 5 and we draw some conclusions in Section 6. Proofs and additional technical material can be found in the long version, which is available at the authors' home pages.

2   An example

It has already been observed in the recent literature that session types can conveniently describe the interface of distributed objects. For instance, the session type

$$\texttt{SellerT} = \mu\alpha.(!\texttt{offer}(\texttt{nat}).?\texttt{response}(\texttt{nat}).\alpha \oplus !\texttt{buy}(\texttt{string}).\textbf{end} \oplus !\texttt{leave}().\textbf{end})$$

may be used to describe (part of) the behavior of a Seller object as it can be used by Buyer. Buyer may make an offer regarding an item he or she is interested to buy by sending a `offer` message and receiving a counteroffer from Seller. After this exchange of messages, the protocol repeats itself. Alternatively, Buyer may buy the item by sending a message that contains the delivery address, or it may `leave` the virtual shop. Buyer and Seller are connected by a pair (c,d) of related endpoints: c : `SellerT` is given to Buyer which can use it according to the protocol `SellerT`; d : $\overline{\texttt{SellerT}}$ is used by Seller and its type $\overline{\texttt{SellerT}}$ is the dual of `SellerT`, where inputs have been replaced by outputs and vice versa. This guarantees that Buyer and Seller interact without errors.

Suppose now that we want to implement a Broker to which Buyer can delegate the bargaining protocol with Seller. We may describe Broker in some pseudo-language, as follows:

```
1    BROKER(b : BrokerT) {
2      p₀ := receive(b, price);
3      x := receive(b, seller);
4      p := p₀;
5      while (better_deal(p₀, p)) {
6        send(x, offer, compute_new_offer(p₀, p));
7        p := receive(x, response);
8      }
9      send(b, price, p);
10     send(b, seller, x);
11   }
```

Buyer and Broker interact by means of another pair (a,b) of related endpoints, a : `BrokerT` in use by Buyer (not shown here) and b : $\overline{\texttt{BrokerT}}$ in use by Broker. Broker accepts an initial `price` p_0 and the endpoint of Seller from Buyer (lines 2–3). Then, it engages the bargaining protocol with Seller (lines 4–8) until the best deal p is achieved. Finally, it returns p and the Seller's endpoint to Buyer (lines 9–10) so that Buyer can conclude the interaction with Seller appropriately.

Buyer interacts with Broker on the endpoint a, whose type can be described by:

$$\texttt{BrokerT} = \texttt{!price(nat).!seller(SellerT).?price(nat).?seller(SellerT).\textbf{end}}$$

Observe that `BrokerT` might be considered too precise, since Broker uses only a strict subset of the functionalities supported by Seller and described in `SellerT`. To maximize reusability, it could be more appropriate to replace the type `SellerT` in `BrokerT` with

$$\texttt{BargainT} = \mu\alpha.\texttt{!offer(nat).?response(nat).}\alpha$$

which specifies the minimum set of functionalities of Seller that Broker actually uses. Buyer can still delegate an endpoint of type `SellerT` to Broker since `SellerT` is a subtype of `BargainT`, which we express as the relation $\texttt{SellerT} \leqslant \texttt{BargainT}$. This is consistent with the notion of subtyping in object-oriented languages: if we think of the message tags in `SellerT` as of the methods provided by Seller, then `BargainT` is one possible *interface* that Seller implements and in fact is the least interface needed by Broker. The problem, in this case, is that when an endpoint is delegated, Buyer can no longer use it (endpoints are linear resources). It is true that Broker eventually returns Seller's endpoint to Buyer, but by that time its type has been widened to `BargainT` and Buyer can no longer access the functionalities in `SellerT` that have been hidden in `BargainT`. This lost information can be recovered using bounded polymorphism and by giving a more precise type to a:

$$\texttt{BrokerT} = \texttt{!price(nat).!seller}\langle\alpha \leqslant \texttt{BargainT}\rangle(\alpha).\texttt{?price(nat).?seller}(\alpha).\textbf{end}$$

Table 1: Syntax of PolySing♯ processes and of heap objects.

P	::=		**Process**	μ	::=		**Heap**
		0	(idle)			$\emptyset$	(empty)
	|	X	(variable)		|	$a \mapsto [a,q]$	(endpoint)
	|	**close**(u)	(close channel)		|	μ,μ	(composition)
	|	**open**$(a,a).P$	(open channel)				
	|	$u!\mathrm{m}(u).P$	(send)	q	::=		**Queue**
	|	$\sum_{i \in I} u?\mathrm{m}_i(x_i).P_i$	(receive)			ε	(empty)
	|	$P \oplus P$	(choice)		|	$\mathrm{m}(a)$	(message)
	|	$P \mid P$	(composition)		|	$q :: q$	(composition)
	|	**rec** $X.P$	(recursion)				

In this case, the protocol still allows Buyer to send any endpoint corresponding to Seller whose type conforms to (is a subtype of) `BargainT`, but it also specifies that the endpoint eventually returned to Buyer has the same type as the one sent earlier. In general, once an endpoint has been delegated, the delegator loses access to the endpoint as well as to any information about its type. This loss of information, which alone justifies the interest for bounded polymorphism in sequential programming, is likely to occur much more frequently in our context where many resources are linear.

3   Language syntax and semantics

We fix some notation: we use P, Q, ... to range over processes and a, b, ... to range over *heap pointers* (or simply *pointers*) taken from some infinite set `Pointers`; we use x, y, ... to range over *variables* taken from some infinite set `Variables` disjoint from `Pointers` and we let u, v, ... range over *names*, which are elements of `Pointers` $\cup$ `Variables`; finally, we let X range over *process variables*.

The language of processes, defined by the grammar in Table 1, essentially is a monadic pi-calculus equipped with tag-based message dispatching and primitives for handling endpoints. The process **0** is idle and performs no action. Terms **rec** $X.P$ and X are used for building recursive processes, as usual. The process $u!\mathrm{m}(v).P$ sends a message $\mathrm{m}(v)$ on the endpoint u and continues as P. A *message* is made of a *tag* m along with it *parameter* v. The term $\sum_{i \in I} u?\mathrm{m}_i(x_i).P_i$ denotes a process that waits for a message from the endpoint u. The tag m_i of the received message determines the continuation P_i where the variable x_i is instantiated with the parameter of the message. We assume that in every such term the m_i's are pairwise distinct. Sometimes we will write $u?\mathrm{m}_1(x_1).P_1 + \cdots + u?\mathrm{m}_n(x_n).P_n$ in place of $\sum_{i=1}^{n} u?\mathrm{m}_i(x_i).P_i$. The term **open**$(a,b).P$ denotes a process creating a *channel*, represented as two peer endpoints a and b. The process **close**(u) closes the endpoint located at u. The processes $P \oplus Q$ and $P \mid Q$ are standard and respectively denote the non-deterministic choice and the parallel composition of P and Q. The sets of free and bound names of every process P, respectively denoted by $\mathrm{fn}(P)$ and $\mathrm{bn}(P)$, are standard: the construct **open**$(a,b).P$ binds both a and b in P, while $\sum_{i \in I} u?\mathrm{m}_i(x_i).P_i$ binds x_i in P_i for each $i \in I$. The construct **rec** X is the only binder for process variables. We adopt the Barendregt convention for both variables and process variables.

Example 3.1 (linear mutable cell). *The following process models a linear mutable cell located at c:*

$$\mathrm{CELL}(c) = \mathbf{rec}\ X.(c?\mathtt{set}(x).c!\mathtt{get}(x).X + c?\mathtt{free}().\mathbf{close}(c))$$

Table 2: Reduction of systems.

(R-OPEN)	(R-REC)
$(\mu; \textbf{open}(a,b).P) \rightarrow (\mu, a \mapsto [b, \varepsilon], b \mapsto [a, \varepsilon]; P)$	$(\mu; \textbf{rec } X.P) \rightarrow (\mu; P\{\textbf{rec } X.P/X\})$

(R-CHOICE)   (R-SEND)

$(\mu; P \oplus Q) \rightarrow (\mu; P)$ $(\mu, a \mapsto [b,q], b \mapsto [a,q']; a!\mathtt{m}(c).P) \rightarrow (\mu, a \mapsto [b,q], b \mapsto [a, q' :: \mathtt{m}(c)]; P)$

(R-RECEIVE)

$$\dfrac{k \in I}{(\mu, a \mapsto [b, \mathtt{m}_k(c) :: q]; \sum_{i \in I} a?\mathtt{m}_i(x_i).P_i) \rightarrow (\mu, a \mapsto [b,q]; P_k\{c/x_k\})}$$

(R-PAR)

$$\dfrac{(\mu; P) \rightarrow (\mu'; P')}{(\mu; P \mid Q) \rightarrow (\mu'; P' \mid Q)}$$

The user of the cell interacts with it on the peer endpoint of c. Initially, the cell is empty and offers to its user the possibility of setting (with a set*-tagged message) the content of the cell with a pointer x, or deallocating (with a* free*-tagged message) the cell. In the first case, the cell transits into a new state where the only possible operation is retrieving (with a* get*-tagged message) the content of the cell. At that point, the cell returns to its original state. The cell is* linear *in the sense that it allows setting its content only if the previous content has been retrieved. This cell implementation resembles that of a 1-place buffer, but we retain the name "cell" for continuity with our previous work [2].* ∎

Heaps, ranged over by μ, ..., are finite maps from pointers to heap objects represented as terms defined according to the syntax in Table 1: the heap $\emptyset$ is empty; the heap $a \mapsto [b,q]$ is made of an endpoint located at a which is a structure referring to the peer endpoint b and containing a *queue q* of messages waiting to be read from a. Heap compositions μ, μ' are defined only when the domains of the heaps being composed, which we denote by $\mathsf{dom}(\mu)$ and $\mathsf{dom}(\mu')$, are disjoint. We assume that heaps are equal up to commutativity and associativity of composition and that $\emptyset$ is neutral for composition. *Queues*, ranged over by q, ..., are finite ordered sequences of messages $\mathtt{m}_1(c_1) :: \cdots :: \mathtt{m}_n(c_n)$. We build queues from the empty queue ε and concatenation of messages by means of ::. We assume that queues are equal up to associativity of :: and that ε is neutral for ::.

We define the operational semantics of processes as the combination of a structural congruence relation and a reduction relation. Structural congruence, denoted by $\equiv$, is the least congruence relation that includes alpha conversion on bound names and the usual laws

$$P \mid \mathbf{0} \equiv P \qquad P \mid Q \equiv Q \mid P \qquad P \mid (Q \mid R) \equiv (P \mid Q) \mid R$$

PolySing♯ processes communicate with each other by means of the heap. Therefore, the reduction relation defines the transitions of *systems* instead of processes, where a system is a pair $(\mu; P)$ of a heap μ and a process P. The reduction relation $\rightarrow$, inductively defined in Table 2, is described in the following paragraph. (R-OPEN) creates a new channel consisting of two fresh endpoints which refer to each other and have an empty queue. (R-CHOICE) (and its symmetric, omitted) states that a process $P \oplus Q$ may autonomously reduce to either P or Q leaving the heap unchanged. (R-SEND) describes the output of a message $\mathtt{m}(c)$ on the endpoint a. The message is enqueued at the right end of a's peer endpoint queue. (R-RECEIVE) describes the input of a message from the endpoint a. The message at the left end of a's queue is removed from the queue, its tag is used for selecting some branch $k \in I$, and its parameter instantiates the variable x_k. (R-PAR) (and its symmetric, omitted) expresses reductions under parallel composition. The heap is treated globally, even when it is only a sub-process to reduce. (R-REC) is the usual unfolding of recursive processes. Observe that the Barendregt convention makes sure that in the

unfolding $P\{\textbf{rec } X.P/X\}$ of a recursive process no name occurring free in P is accidentally captured, because $\textsf{fn}(P) \cap \textsf{bn}(P) = \emptyset$. Not shown in Table 2 is the usual rule (R-STRUCT) describing reductions modulo structural congruence, which plays an essential role in ensuring that $\textbf{open}(a,b).P$ is never stuck, because a and b can always be alpha converted to some pointers not occurring in $\textsf{dom}(\mu)$. There is no reduction for $\textbf{close}(a)$ processes. In principle, $\textbf{close}(a)$ should deallocate the endpoint located at a by removing its association from the heap. However, since peer endpoints mutually refer to each other, removing one endpoint could leave a dangling reference from the corresponding peer. Also, it is convenient to treat $\textbf{close}(a)$ processes as persistent because, in this way, we keep track of the pointers that have been properly deallocated. We will see that this information is crucial in the definition of well-behaved processes (Definition 3.2). A process willing to deallocate a pointer a and to continue as P afterwards can be modelled as $\textbf{close}(a) \mid P$. In the following we write $\Rightarrow$ for the reflexive, transitive closure of $\to$ and we write $(\mu; P) \nrightarrow$ if there exist no μ' and P' such that $(\mu; P) \to (\mu'; P')$.

In this work we characterize well-behaved systems as those that are free from faults, leaks, and communication errors: a *fault* is an attempt to use a pointer not corresponding to an allocated object or to use a pointer in some way which is not allowed by the object it refers to; a *leak* is a region of the heap that some process allocates and that becomes unreachable because no reference to it is directly or indirectly available to the processes in the system; a *communication error* occurs if some process receives a message of unexpected type. We conclude this section formalizing these properties. To do so, 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 $\textsf{fn}(P)$ (we can think of the pointers in $\textsf{fn}(P)$ as of the local variables of the process stored in its stack); from these pointers, the process may reach other heap objects by reading messages from other endpoints it can reach.

Definition 3.1 (reachable pointers). *We say that c is* reachable *from a in μ, notation $c \prec_\mu a$, if $a \mapsto [b, q :: \texttt{m}(c) :: q'] \in \mu$. We write $c \prec_\mu^* a$ for the reflexive, transitive closure of $\prec_\mu$. Let $\textsf{reach}(A, \mu) = \{c \mid \exists a \in A : c \prec_\mu^* a\}$.*

We now define well-behaved systems formally.

Definition 3.2 (well-behaved process). *We say that P is* well behaved *if $(\emptyset; P) \Rightarrow (\mu; Q)$ implies:*

1. *$\textsf{dom}(\mu) = \textsf{reach}(\textsf{fn}(Q), \mu)$;*

2. *$Q \equiv P_1 \mid P_2$ implies $\textsf{reach}(\textsf{fn}(P_1), \mu) \cap \textsf{reach}(\textsf{fn}(P_2), \mu) = \emptyset$;*

3. *$Q \equiv P_1 \mid P_2$ and $(\mu; P_1) \nrightarrow$ where P_1 does not have unguarded parallel compositions imply either $P_1 = \boldsymbol{0}$ or $P_1 = \textbf{close}(a)$ or $P_1 = \sum_{i \in I} a?\texttt{m}_i(x_i).P_i$ and $a \mapsto [b, \varepsilon] \in \mu$.*

In words, a process P is well behaved if every residual of P reachable from a configuration where the heap is empty satisfies a number of conditions. Conditions (1) and (2) guarantee the absence of faults and leaks. Indeed, condition (1) states that every pointer to the heap is reachable by one process, and that every reachable pointer corresponds to an object allocated on the heap. Condition (2) states that processes are isolated, namely that the sets of reachable pointers corresponding to different processes are disjoint. Since processes of the form $\textbf{close}(a)$ are persistent, this also guarantees the absence of faults where a process tries to use an endpoint that has already been deallocated, or where the same endpoint is deallocated twice. Condition (3) guarantees the absence of communication errors, namely that if $(\mu; Q)$ is stuck (no reduction is possible), then it is because every non-terminated process in Q is waiting for a message on an endpoint having an empty queue. This configuration corresponds to a genuine deadlock where every process in some set is waiting for a message that is to be sent by another process in the same set. We only consider initial configurations with an empty heap for two reasons: first, we take the point of view that initially there are no allocated objects; second, since we will need a well-typed predicate for

Table 3: Syntax of types.

T	$::=$		**Endpoint Type**	t	$::=$		**Type**
		end	(termination)			Top	(top type)
	$\mid$	α	(variable)		$\mid$	T	(endpoint type)
	$\mid$	$!\{m_i\langle\alpha_i \leqslant t_i\rangle(s_i).T_i\}_{i\in I}$	(internal choice)				
	$\mid$	$?\{m_i\langle\alpha_i \leqslant t_i\rangle(s_i).T_i\}_{i\in I}$	(external choice)				
	$\mid$	$\mu\alpha.T$	(recursive type)				

heaps and we do not want to verify heap well-typedness at runtime, we will make sure that the empty heap is trivially well typed.

4   Type system

We introduce some notation for the type system: we assume an infinite set of *type variables* ranged over by α, β, ... Types are ranged over by t, s, ... while endpoint types are ranged over by T, S, ... The syntax of types and endpoint types is defined in Table 3. 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, and deallocate the endpoint. The endpoint type **end** denotes an endpoint on which no further input/output operation is possible and that can be deallocated. An endpoint type $!\{m_i\langle\alpha_i \leqslant t_i\rangle(s_i).T_i\}_{i\in I}$ denotes an endpoint on which a process may send any message m_i with $i \in I$. The message carries an argument of type s_i and the type variable α_i can be instantiated with any subtype of the bound t_i (subtyping will be defined shortly). Depending on the tag m_i of the message, the endpoint can be used thereafter according to the endpoint type T_i. In a dual manner, an endpoint type $?\{m_i\langle\alpha_i \leqslant t_i\rangle(s_i).T_i\}_{i\in I}$ denotes and endpoint from which a process must be ready to receive any message m_i with $i \in I$. Again, s_i denotes the type of the message's argument, while t_i is the bound for the type variable α_i. Depending on the tag m_i of the received message, the endpoint is to be used according to T_i. Terms α and $\mu\alpha.T$ can be used to specify recursive behaviors, as usual. Types are either endpoint types or the top type Top, which is supertype of any other (endpoint) type.

   Here are some handy conventions regarding types and endpoint types:

- we sometimes use an infix notation for internal and external choices and write $!m_1\langle\alpha_1 \leqslant t_1\rangle(s_1).T_1 \oplus \cdots \oplus !m_n\langle\alpha_n \leqslant t_n\rangle(s_n).T_n$ instead of $!\{m_i\langle\alpha_i \leqslant t_i\rangle(s_i).T_i\}_{i\in\{1,\dots,n\}}$ and $?m_1\langle\alpha_1 \leqslant t_1\rangle(s_1).T_1 + \cdots + ?m_n\langle\alpha_n \leqslant t_n\rangle(s_n).T_n$ instead of $?\{m_i\langle\alpha_i \leqslant t_i\rangle(s_i).T_i\}_{i\in\{1,\dots,n\}}$;

- we omit the bound when it is Top and write, for example, $!m\langle\alpha\rangle(s).T$ in place of $!m\langle\alpha \leqslant \text{Top}\rangle(s).T$;

- we omit the bound specification $\langle\cdot\rangle$ altogether when useless (if the type variable occurs nowhere else) and write, for example, $!m(s).T$;

- we omit the type of the argument when irrelevant.

   We have standard notions of free and bound type variables for (endpoint) types. The only binders are μ and the bound constraints for messages. In particular, $\mu\alpha.T$ binds α in T and $\dagger m\langle\alpha \leqslant t\rangle(s).T$ where $\dagger \in \{!,?\}$ binds α in s and in T but not in t. We adopt the Barendregt convention for type variables. In what follows we will consider endpoint types modulo renaming of bound variables and folding/unfolding of recursions, that is $\mu\alpha.T = T\{\mu\alpha.T/\alpha\}$ where $T\{\mu\alpha.T/\alpha\}$ is the endpoint type obtained from T by replacing each free occurrence of α with $\mu\alpha.T$.

The duality between inputs and outputs induces dual bounding modalities. In particular, a process using an endpoint with type $!\mathtt{m}\langle\alpha\leqslant t\rangle(s).T$ may choose a particular $t'\leqslant t$ to instantiate α and use $T\{t'/\alpha\}$ accordingly. In other words, the variable α is *universally quantified* over all the subtypes of t. Dually, a process using an endpoint with type $?\mathtt{m}\langle\alpha\leqslant t\rangle(s).T$ does not know the exact type $t'\leqslant t$ with which α is instantiated, since this type is chosen by the sender. In other words, the type variables of an external choice are *existentially quantified* over all the subtypes of the corresponding bounds.

Not every term generated by the grammar in Table 3 makes sense. Type variables bound by a recursion μ must be guarded by a prefix (therefore a non-contractive endpoint type such as $\mu\alpha.\alpha$ is forbidden) and type variables bound in a constraint as in $!\mathtt{m}\langle\alpha\leqslant t\rangle(s).T$ can only occur in s and within the prefixes of T. We formalize this last requirement as a well-formedness condition for types denoted by a judgment $\mathscr{I},\mathscr{O}\vdash t$ where $\mathscr{I}$ is a set of so-called *inner variables* (those that can occur only within prefixes) and $\mathscr{O}$ is a set of so-called *outer variables* (those that can occur everywhere):

$$\mathscr{I},\mathscr{O}\vdash\textbf{end}\qquad\mathscr{I},\mathscr{O}\vdash\mathsf{Top}\qquad\frac{\alpha\in\mathscr{O}\setminus\mathscr{I}}{\mathscr{I},\mathscr{O}\vdash\alpha}\qquad\frac{\mathscr{I},\mathscr{O}\cup\{\alpha\}\vdash T}{\mathscr{I},\mathscr{O}\vdash\mu\alpha.T}$$

$$\frac{\dagger\in\{!,?\}\qquad\emptyset,\mathscr{I}\cup\mathscr{O}\vdash t_i{}^{(i\in I)}\qquad\emptyset,\mathscr{I}\cup\mathscr{O}\cup\{\alpha_i\}\vdash s_i{}^{(i\in I)}\qquad\mathscr{I}\cup\{\alpha_i\},\mathscr{O}\vdash T_i{}^{(i\in I)}}{\mathscr{I},\mathscr{O}\vdash\dagger\{\mathtt{m}_i\langle\alpha_i\leqslant t_i\rangle(s_i).T_i\}_{i\in I}}$$

We say that t is well formed if $\emptyset,\emptyset\vdash t$ is derivable. Observe that well-formed (endpoint) types are closed. Well formedness restricts the expressiveness of types, in particular types such as $!\mathtt{m}\langle\alpha\leqslant t\rangle(s).\alpha$ and $?\mathtt{m}\langle\alpha\leqslant t\rangle(s).\alpha$ are forbidden because ill formed. We claim that ill-formed types have negligible practical utility: a process that instantiates α with $S\leqslant T$ in $!\mathtt{m}\langle\alpha\leqslant T\rangle(s).\alpha$ precisely knows its behavior after the output operation; dually, a process that receives a message from an endpoint with type $?\mathtt{m}\langle\alpha\leqslant T\rangle(s).\alpha$ cannot do any better than assuming that α has been instantiated with T.

Duality expresses the fact that two processes accessing peer endpoints interact without errors if they behave in complementary ways: if one of the two processes sends a message, the other process waits for a message; if one process waits for a message, the other process sends; if one process has finished using an endpoint, the other process has finished too.

Definition 4.1 (duality). *We say that $\mathscr{D}$ is a* duality relation *if $(T,S)\in\mathscr{D}$ implies either*

- $T=S=\textbf{end}$*, or*
- $T=?\{\mathtt{m}_i\langle\alpha_i\leqslant t_i\rangle(s_i).T_i\}_{i\in I}$ *and* $S=!\{\mathtt{m}_i\langle\alpha_i\leqslant t_i\rangle(s_i).S_i\}_{i\in I}$ *and* $(T_i,S_i)\in\mathscr{D}$ *for every $i\in I$,*
- $T=!\{\mathtt{m}_i\langle\alpha_i\leqslant t_i\rangle(s_i).T_i\}_{i\in I}$ *and* $S=?\{\mathtt{m}_i\langle\alpha_i\leqslant t_i\rangle(s_i).S_i\}_{i\in I}$ *and* $(T_i,S_i)\in\mathscr{D}$ *for every $i\in I$.*

We say that T and S are dual *if $(T,S)\in\mathscr{D}$ for some duality relation $\mathscr{D}$.*

It is easy to see that if T and S_1 are dual and T and S_2 are dual, then $S_1=S_2$. In other words, the duality relation induces a function $\overline{\cdot}$ such that T and $\overline{T}$ are dual for every T.

An important property of well-formed endpoint types is that duality does not affect their inner variables. Therefore, duality and the instantiation of inner variables commute, in the following sense:

Proposition 4.1. *Let $\{\alpha\},\emptyset\vdash T$. Then $\overline{T\{s/\alpha\}}=\overline{T}\{s/\alpha\}$.*

We now define subtyping. Because of bound constraints on type variables, subtyping is relative to an environment $\Delta=\alpha_1\leqslant t_1,\dots,\alpha_n\leqslant t_n$ which is an ordered sequence of constraints such that each α_i may only occur in the t_j's with $j>i$. We write $\mathsf{dom}(\Delta)$ for the domain of Δ and $\Delta(\alpha_i)$ to denote the bound t_i associated with the rightmost occurrence of $\alpha_i\in\mathsf{dom}(\Delta)$; we use $\bullet$ to denote the empty bound environment. Subtyping is fairly standard, therefore we provide only a coinductive characterization (an equivalent deduction system restricted to finite endpoint types can be found in [8]).

Definition 4.2 (subtyping). *We say that $\mathscr{S}$ is a* coinductive subtyping *if $(\Delta, t, s) \in \mathscr{S}$ implies either:*

1. *$t = s$, or*

2. *$s = \mathsf{Top}$, or*

3. *$t = \alpha \in \mathsf{dom}(\Delta)$ and $(\Delta, \Delta(\alpha), s) \in \mathscr{S}$, or*

4. *$t = ?\{m_i \langle \alpha_i \leqslant t_i \rangle (s_i).T_i\}_{i \in I}$ and $s = ?\{m_i \langle \alpha_i \leqslant t_i \rangle (s_i').S_i\}_{i \in J}$ with $I \subseteq J$ and $((\Delta, \alpha_i \leqslant t_i), s_i, s_i') \in \mathscr{S}$ and $((\Delta, \alpha_i \leqslant t_i), T_i, S_i) \in \mathscr{S}$ for every $i \in I$, or*

5. *$t = !\{m_i \langle \alpha_i \leqslant t_i \rangle (s_i).T_i\}_{i \in I}$ and $s = !\{m_i \langle \alpha_i \leqslant t_i \rangle (s_i').S_i\}_{i \in J}$ with $J \subseteq I$ and $((\Delta, \alpha_i \leqslant t_i), s_i', s_i) \in \mathscr{S}$ and $((\Delta, \alpha_i \leqslant t_i), T_i, S_i) \in \mathscr{S}$ for every $i \in I$.*

We write $\Delta \vdash t \leqslant s$ if $(\Delta, t, s) \in \mathscr{S}$ for some coinductive subtyping $\mathscr{S}$ and $t \leqslant s$ if $\bullet \vdash t \leqslant s$.

Item (1) states that subtyping is reflexive; item (2) states that Top is indeed the top type; items (4) and (5) are the usual covariant and contravariant rules for inputs and outputs respectively. Observe that subtyping is always covariant with respect to the continuations and that we require the bounds on type variables of related endpoint types be the same. This corresponds to the so-called "Kernel" variant of bounded polymorphism as opposed to the "Full" one. We adopt the Kernel variant for simplicity, since it is orthogonal to the rest of the theory. Also, the Full variant is known to be undecidable [8]. Finally, item (3) allows one to deduce $\Delta \vdash \alpha \leqslant s$ if $\Delta \vdash \Delta(\alpha) \leqslant s$ holds.

The reader may easily verify that subtyping is transitive (it is enough to show that $\{(\Delta, t_1, t_2) \mid \exists s : \Delta \vdash t_1 \leqslant s \ \& \ \Delta \vdash s \leqslant t_2\}$ is a coinductive subtyping). The following property is standard and shows that duality is contravariant with respect to subtyping.

Proposition 4.2. *$T \leqslant S$ if and only if $\overline{S} \leqslant \overline{T}$.*

Well-formedness of endpoint types is essential for Proposition 4.2 to hold in our setting. Consider, for example, the endpoint types $T = !m\langle \alpha \leqslant ?m().\mathbf{end} \rangle().\alpha$ and $S = !m\langle \alpha \leqslant ?m().\mathbf{end} \rangle().?m().\mathbf{end}$ where T is ill formed. Then $T \leqslant S$ would hold but $?m\langle \alpha \leqslant ?m().\mathbf{end} \rangle().!m().\mathbf{end} = \overline{S} \leqslant \overline{T} = ?m\langle \alpha \leqslant ?m().\mathbf{end} \rangle().\alpha$ would not because $\alpha \leqslant ?m().\mathbf{end} \vdash !m().\mathbf{end} \not\leqslant \alpha$. An alternative theory where well-formedness is not necessary for proving Proposition 4.2 is given in [8] and consists in distinguishing dualized type variables $\overline{\alpha}$ from type variables and by having type constraints of the form $t_1 \leqslant \alpha \leqslant t_2$ with both lower and upper bounds, so that the bounds of the corresponding dualized type variable are known and given by $\overline{t_2} \leqslant \overline{\alpha} \leqslant \overline{t_1}$.

Typing the heap. The heap plays a primary role in our setting because inter-process communication utterly relies on heap-allocated structures; also, most properties of well-behaved processes are direct consequences of related properties of the heap. Therefore, just as we will check well typedness of a process P with respect to some environment that associates the pointers occurring in P with the corresponding types, we will 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. The mere fact that we have this notion does not mean that we need to type-check the heap at runtime. Well typedness of the heap will be a consequence of well typedness of processes, and the empty heap will be trivially well typed. We will express well-typedness of a heap μ with respect to a pair $\Gamma_0; \Gamma$ of environments where Γ represents the type of the *roots* of μ (the pointers that are not referenced by any other structure allocated in the heap) and Γ_0 describes the type of the pointers to allocated structures that are directly or indirectly reachable from one of the roots of the heap.

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 of the communication model that we have adopted, which is asynchronous. Since messages can accumulate in the queue of an endpoint before they are received, the type of the endpoint as perceived by the process using it and the actual type of the endpoint as assumed by the process using its peer can be misaligned. On the one hand, we want to enforce the invariant that the endpoint types of peer endpoints are (and remain) dual, modulo the subtyping relation. On the other hand, this can only happen when the two endpoints have empty queues. In general, we need to compute the actual endpoint type of an endpoint taking into account its type as perceived by the process using it *and* the messages in its queue. This is accomplished by the $\mathsf{tail}(\cdot,\cdot)$ function below, which takes an endpoint type T and a sequence $\mathsf{m}_1(s_1)\cdots\mathsf{m}_n(s_n)$ of specifications corresponding the messages enqueued into the endpoint with type T and computes the residual endpoint type that assumes that all those messages have been received:

$$\mathsf{tail}(T,\varepsilon) = T \qquad \frac{k \in I \qquad t \leqslant t_k \qquad s \leqslant s_k\{t/\alpha_k\} \qquad \mathsf{tail}(T_k\{t/\alpha_k\},\mathsf{m}_1(s_1')\cdots\mathsf{m}_n(s_n')) = S}{\mathsf{tail}(?\{\mathsf{m}_i\langle\alpha_i \leqslant t_i\rangle(s_i).T_i\}_{i\in I},\mathsf{m}_k(s)\mathsf{m}_1(s_1')\cdots\mathsf{m}_n(s_n')) = S}$$

From a technical point of view tail is a relation, since there can be several possible choices for instantiating the type variables in the endpoint type being processed. For example, we have $\mathsf{tail}(?\mathsf{m}\langle\alpha \leqslant t\rangle(\alpha).?\mathsf{m}(\alpha).\mathbf{end},\mathsf{m}(s)) = ?\mathsf{m}(t').\mathbf{end}$ for every $s \leqslant t' \leqslant t$. Nonetheless we will sometimes use tail as a function and write $\mathsf{tail}(T,\mathsf{m}_1(s_1)\cdots\mathsf{m}_m(s_n))$ in place of some S such that $\mathsf{tail}(T,\mathsf{m}_1(s_1)\cdots\mathsf{m}_m(s_n)) = S$. For instance, the notation $\mathsf{tail}(T,\mathsf{m}_1(s_1)\cdots\mathsf{m}_m(s_n)) \leqslant S'$ means that $S \leqslant S'$ for some S such that $\mathsf{tail}(T,\mathsf{m}_1(s_1)\cdots\mathsf{m}_m(s_n)) = S$.

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)**.** *We write $\Gamma_0;\Gamma \vdash \mu$ if:*

1. *for every $a \mapsto [b,q] \in \mu$ we have $b \mapsto [a,q'] \in \mu$ and either $q = \varepsilon$ or $q' = \varepsilon$;*

2. *for every $a \mapsto [b,\mathsf{m}_1(c_1) :: \cdots :: \mathsf{m}_n(c_n)] \in \mu$ and $b \mapsto [a,\varepsilon]$ we have $\overline{\mathsf{tail}(T,\mathsf{m}_1(s_1)\cdots\mathsf{m}_m(s_n))} \leqslant S$ where $a : T \in \Gamma$ and $b : S \in \Gamma$ and $c_i : s_i \in \Gamma$ for $i \in \{1,\ldots,n\}$;*

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

4. $\mathsf{reach}(\{a\},\mu) \cap \mathsf{reach}(\{b\},\mu) = \emptyset$ *for every $a,b \in \mathsf{dom}(\Gamma)$ with $a \neq b$.*

Condition (1) requires that in a well-typed heap every endpoint comes along with its peer and that at least one of the queues of peer endpoints be empty. This invariant is ensured by duality, since a well-typed process does not send messages on an endpoint until it has read all the pending messages from the corresponding queue. Condition (2) requires that the endpoint types of peer endpoints are dual, modulo subtyping. More precisely, for every endpoint a whose peer b has an empty queue there exists a residual $\mathsf{tail}(T,\mathsf{m}_1(s_1)\cdots\mathsf{m}_n(s_n))$ of its type T whose dual is a subtype of the peer's type S. 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 $b \in \mathsf{dom}(\Gamma)$. Since the roots will be distributed linearly to the processes of the system, this guarantees the absence of leaks, namely of allocated objects which are no longer reachable. Finally, condition (4) requires the uniqueness of the root for every allocated object. This guarantees process isolation, namely the fact that every allocated object belongs to one and only one process.

Typing processes. We want to define a type system such that well-typed processes are well behaved and, in particular, such that well-typed processes do not leak memory. As we have anticipated in the

introduction, the critical situation that we must avoid is the possibility that an endpoint is enqueued into its own queue, since this would cause the creation of a circular structure not owned by any process.

The intuition behind our solution is to use some property of types to detect — and avoid — the configurations in which there exists some potential to create such circular structures. This property, which we dub *weight* of a type, gives an upper bound to the length of chains of pointers linking endpoints.

Definition 4.4 (weight). *We say that $\mathcal{W}$ is a coinductive weight bound if $(\Delta, t, n) \in \mathcal{W}$ implies either:*

- $t = \textbf{end}$, *or*

- $t = !\{m_i\langle \alpha_i \leqslant t_i\rangle(s_i).T_i\}_{i \in I}$, *or*

- $t = \alpha \in \text{dom}(\Delta)$ *and* $(\Delta, \Delta(\alpha), n) \in \mathcal{W}$, *or*

- $t = ?\{m_i\langle \alpha_i \leqslant t_i\rangle(s_i).T_i\}_{i \in I}$ *and* $n > 0$ *and* $((\Delta, \alpha_i \leqslant t_i), s_i, n-1) \in \mathcal{W}$ *and* $((\Delta, \alpha_i \leqslant t_i), T_i, n) \in \mathcal{W}$
  for every $i \in I$.

We write $\Delta \vdash t \downarrow n$ if $(\Delta, t, n) \in \mathcal{W}$ for some coinductive weight bound $\mathcal{W}$. The weight *of a type t with respect to Δ, denoted by $\|t\|_\Delta$, is defined by $\|t\|_\Delta = \min\{n \in \mathbb{N} \mid \Delta \vdash t \downarrow n\}$ where we let $\min \emptyset = \infty$. We omit the environment Δ when it is empty and simply write $\|t\|$ instead of $\|t\|_\bullet$. 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 $\infty \leq \infty$.*

Like other relations involving types, the relation $\Delta \vdash t \downarrow n$ is parametric on an environment Δ specifying the upper bound of type variables that may occur in t and expresses the fact that n is an upper bound for the weight of t. The weight of t is then defined as the least of its upper bounds, or ∞ if there is no such upper bound. A few weights are straightforward to compute, for example we have $\|\textbf{end}\| = \|!\{m_i\langle \alpha_i \leqslant t_i\rangle(s_i).T_i\}_{i \in I}\| = 0$ and $\|\textsf{Top}\| = \infty$. Endpoints with type **end** and those in a send state have a null weight because their corresponding queues are empty and therefore the chains of pointers originating from them has zero length. In the case of $\textsf{Top}$, it does not have a finite weight since $\textsf{Top}$ is the type of *any* endpoint, and in particular of any endpoint with an arbitrary weight. Only endpoints in a receive state do have a strictly positive weight. For instance we have $\|?m(\textbf{end}).\textbf{end}\| = 1$ and $\|?m(?m(\textbf{end}).\textbf{end}).\textbf{end}\| = 2$, while $\|\mu\alpha.?m(\alpha).T\|_\Delta = \|?m(\textsf{Top}).T\|_\Delta = \infty$. The weight of type variables occurring in a constraint $\alpha \leqslant t$ is given by the weight of t. In particular, $\|\alpha\|_{\alpha \leqslant t} = \|t\|$ and $\|?m\langle \alpha \leqslant t\rangle(\alpha).\textbf{end}\| = 1 + \|t\|$ (this latter equality holds if $\|t\| < \infty$). In a sense, the (type) bound t for α determines also a (weight) bound $\|t\|$ for α. Since the actual type with which α will be instantiated is not known, this approximation works well only if the relation between the weights is coherent with subtyping. This fundamental property does indeed hold, as stated in the following proposition.

Proposition 4.3. $t \leqslant s$ *implies* $\|t\| \leq \|s\|$.

The typing rules for processes are inductively defined in Table 4. Judgments have the form $\Sigma; \Delta; \Gamma \vdash P$ and state that process P is well typed under the specified environments. The additional environment Σ is a map from process variables to pairs $(\Delta; \Gamma)$ and is used for typing recursive processes. It plays a role in two rules only, (T-VAR) and (T-REC), which are standard except for the unusual premise $\text{dom}(\Gamma) = \text{fn}(P)$ in rule (T-REC) that enforces a weak form of contractivity on recursive processes. It states that **rec** $X.P$ is well typed under Γ only if P actually uses the names in $\text{dom}(\Gamma)$. Normally, divergent processes such as **rec** $X.X$ can be typed in every environment. If this were the case, the process **open**$(a,b).$**rec** $X.X$, which leaks a and b, would be well typed. The idle process is well typed in the empty environment $\emptyset$. Since we will impose a correspondence between the free names of a process and the roots of the heap, this rule states that the terminated process has no leaks. Rule (T-CLOSE) states that a process **close**(u) is well typed provided that u is the only name owned by the process and that it corresponds to an endpoint with type **end**, on which no further interaction is possible. Rule (T-OPEN) types the creation of a new

Table 4: Typing rules for processes.

$$
\text{(T-IDLE)} \qquad \text{(T-CLOSE)} \qquad \frac{\text{(T-REC)}}{\Sigma;\Delta;\Gamma \vdash \textbf{rec } X.P} \quad \frac{\Sigma,\{X \mapsto (\Delta;\Gamma)\};\Delta;\Gamma \vdash P \qquad \text{dom}(\Gamma) = \text{fn}(P)}{\Sigma;\Delta;\Gamma \vdash \textbf{rec } X.P}
$$

$$
\Sigma;\Delta;\emptyset \vdash \textbf{0} \qquad \Sigma;\Delta;u:\textbf{end} \vdash \textbf{close}(u)
$$

$$
\frac{\text{(T-OPEN)}}{\Sigma;\Delta;\Gamma,a:T,b:\overline{T} \vdash P} \qquad \frac{\text{(T-SEND)}}{\Delta \vdash t' \leqslant t \qquad \|s\{t'/\alpha\}\|_\Delta < \infty \qquad \Sigma;\Delta;\Gamma,u:S\{t'/\alpha\} \vdash P}
$$

$$
\frac{}{\Sigma;\Delta;\Gamma \vdash \textbf{open}(a,b).P} \qquad \frac{}{\Sigma;\Delta;\Gamma,u:\,!m\langle\alpha \leqslant t\rangle(s).S,v:s\{t'/\alpha\} \vdash u!m(v).P}
$$

$$
\frac{\text{(T-VAR)}}{\Sigma,\{X \mapsto (\Delta;\Gamma)\};\Delta;\Gamma \vdash X} \qquad \frac{\text{(T-RECEIVE)}}{\Sigma;\Delta,\alpha_i \leqslant t_i;\Gamma,u:T_i,x_i:s_i \vdash P_i \;\;^{(i \in I)}}{\Sigma;\Delta;\Gamma,u:\,?\{m_i\langle\alpha_i \leqslant t_i\rangle(s_i).T_i\}_{i \in I} \vdash \sum_{i \in I} u?m_i(x_i).P_i}
$$

$$
\frac{\text{(T-CHOICE)}}{\Sigma;\Delta;\Gamma \vdash P \qquad \Sigma;\Delta;\Gamma \vdash Q} \qquad \frac{\text{(T-PAR)}}{\Sigma;\Delta;\Gamma_1 \vdash P \qquad \Sigma;\Delta;\Gamma_2 \vdash Q} \qquad \frac{\text{(T-SUB)}}{\Sigma;\Delta;\Gamma,u:s \vdash P \qquad \Delta \vdash t \leqslant s}
$$

$$
\frac{}{\Sigma;\Delta;\Gamma \vdash P \oplus Q} \qquad \frac{}{\Sigma;\Delta;\Gamma_1,\Gamma_2 \vdash P\,|\,Q} \qquad \frac{}{\Sigma;\Delta;\Gamma,u:t \vdash P}
$$

channel, which is visible in the continuation process as two peer endpoints typed by dual endpoint types. Rules (T-CHOICE) and (T-PAR) are standard. In the latter, the type environment is split into disjoint environments to type the processes being composed. Together with heap well-typedness, this ensures process isolation. Rule (T-SEND) states that a process $u!m(v).P$ is well typed if u is associated with an endpoint type $!m\langle\alpha \leqslant t\rangle(s).S$ that permits the output of m-tagged messages. The rule guesses the type parameter t' with which the type variable α is instantiated (an explicitly typed process might explicitly provide t'). The type of the argument v must match the expected type in the endpoint type where α has been instantiated with t' and the continuation P must be well typed in a context where the message argument has disappeared and the endpoint u is typed according to a properly instantiated S. This means that P can rely on the knowledge of t', namely α is universally quantified over all the subtypes of t. The condition $\|s\{t'/\alpha\}\|_\Delta < \infty$ imposes that v's type must have a finite weight. Since the peer of u must be able to accept a message with an argument of type $s\{t'/\alpha\}$, its weight will be strictly larger than that of $s\{t'/\alpha\}$, or it will be infinite. In both cases, we are sure that the argument v being sent is not the peer of u. Rule (T-RECEIVE) deals with inputs: a process waiting for a message from an endpoint $u : ?\{m_i\langle\alpha_i \leqslant t_i\rangle(s_i).T_i\}_{i \in I}$ is well typed if it can deal with any m_i-tagged message. The continuation process may use the endpoint u according to the endpoint type T_i and can access the message argument x_i. The environment Γ is enriched with the assumption $\alpha_i \leqslant t_i$ denoting the fact that P_i does not know the exact type with which α_i has been instantiated, but only its upper bound, namely α_i is existentially quantified over all the subtypes of t_i. Finally, rule (T-SUB) is a subsumption rule for assumptions: if a process P is well typed with respect to a context $\Gamma,u:s$, it remains well typed if the type associated with u is more precise than (is a subtype of) s.

Systems $(\mu;P)$ are well typed if so are their components:

Definition 4.5 (well-typed system). *We write* $\Gamma_0;\Gamma \vdash (\mu;P)$ *if* $\Gamma_0;\Gamma \vdash \mu$ *and* $\Gamma \vdash P$.

We conclude with two standard results about our framework: well-typedness is preserved by reduction, and well-typed process are well behaved. Subject reduction is slightly non-standard, in the sense that types in the environment may change as the process reduces. This is common in session type theories, since session types are behavioral types.

Theorem 4.1 (subject reduction). *Let $\Gamma_0; \Gamma \vdash (\mu; P)$ and $(\mu; P) \to (\mu'; P')$. Then $\Gamma_0'; \Gamma' \vdash (\mu'; P')$ for some Γ_0' and Γ'.*

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

Example 4.1. *Consider the endpoint type*

$$\text{CellT} = \mu\alpha.(!\text{set}\langle\beta\rangle(\beta).?\text{get}(\beta).\alpha \oplus !\text{free}().\textbf{end})$$

modeling the interface of a linear mutable cell. Then it is easy to verify that

$$c : \overline{\text{CellT}} \vdash \text{CELL}(c)$$

is derivable, where CELL *is the process presented in Example 3.1. Therefore,* CELL *is a correct implementation of a linear mutable cell.*

Since CellT *begins with an internal choice we have* $\|\text{CellT}\| = 0$. *This means that it is always safe to send an empty cell as the argument of a message since the second premise of rule* (T-SEND) *will always be satisfied. On the contrary, we have* $\|?\text{get}(\beta).\text{CellT}\|_{\beta \leqslant \text{Top}} = \infty$, *therefore it seems like initialized cells can never be sent as messages. However, if sender and initializer are the same process, there might be just enough information to deduce that the process is safe. For example, the judgment*

$$a : t, b : !\text{send}(?\text{get}(t).\text{CellT}).\textbf{end}, c : \text{CellT} \vdash c!\text{set}(a).b!\text{send}(c).\textbf{close}(b)$$

is derivable if $\|t\| < \infty$. *In this case, the sub-process* $b!\text{send}(c).\textbf{close}(b)$ *is type checked in an environment where the (residual) endpoint type of c has been instantiated to* $?\text{get}(t).\text{CellT}$ *whose weight is* $\|?\text{get}(t).\text{CellT}\| = \|t\| + 1 < \infty$. ∎

Example 4.2. *Suppose we want to implement a forwarder process that receives two endpoints with dual types and forwards the stream of messages coming from the first endpoint to the second one. We might implement the process thus:*

$$\text{FWD}(a) = a?\text{src}(x).a?\text{dest}(y).\textbf{rec}\, X.(x?\text{m}(z).y!\text{m}(z).X + x?\text{eos}().y!\text{eos}().(\textbf{close}(x)|\textbf{close}(y)|\textbf{close}(a)))$$

However, the judgment

$$a : ?\text{src}\langle\alpha\rangle(\overline{\text{Stream}}).?\text{dest}(\text{Stream}).\textbf{end} \vdash \text{FWD}(a)$$

where $\text{Stream} = \mu\beta.(!\text{m}(\alpha).\beta \oplus !\text{eos}().\textbf{end})$ *is not derivable because there is no upper bound to the weight of α and* $\text{FWD}(a)$ *attempts at sending z where* $z : \alpha$. *Had* $\text{FWD}(a)$ *been typable, it would be possible to create a leak with the process*

$$\textbf{open}(a,b).(\text{FWD}(a)|\textbf{open}(c,d).\textbf{open}(e,f).b!\text{src}(d).b!\text{dest}(e).c!\text{m}(f).c!\text{eos}().(\textbf{close}(b)|\textbf{close}(c))$$

which has the effect to enqueue f into its own queue. The process FWD *becomes typable as soon as α in a's type is given a bound with a finite weight.* ∎

5   Related work

Copyless message passing is one of the key features adopted by the Singularity OS [12] to compensate the overhead of communication-based interactions between isolated processes. Communication safety and

deadlock freedom can be ensured by checking processes against *channel contracts* that are *deterministic*, *autonomous*, and *synchronizing* [15]. As argued in [7] and shown in [2], the first two conditions make it possible to split contracts into pairs of dual endpoint types, and to implement the static analysis along the lines of well known session type theories [10, 11]. In [2] it was also observed that these conditions are insufficient for preventing memory leaks and it was shown how to address the issue by imposing a "finite-weight" restriction to endpoint types. In the present paper, we further generalize our solution by allowing infinite-weight endpoint types in general, although only endpoints with finite-weight can actually be sent as messages (see the extra premise of rule (T-SEND)). As a matter of fact, [7] already noted that the implementation of ownership transfer posed some consistency issues if endpoints not in a *send-state* were allowed to be sent as messages, but no relation with memory leaks was observed. Since our "finite-weight" condition is a generalization of the *send-state* condition (*send-state* endpoints always have null weight), our work provides a formal proof that the *send-state* condition is sufficient also for avoiding memory leaks. Other works [7, 9] introduce apparently similar, "finite-weight" restrictions on session types to make sure that message queues of the corresponding channels are bounded. Our weights are unrelated to the size of queues and concern the length of chains of pointers involving queues.

Polymorphic contracts and endpoint types of the Singularity OS [13] have never been formalized before nor do they appear to be supported by the Singularity RDK. Our polymorphic endpoint types are closely related to the session types in [8], which was the first work to introduce bounded polymorphism for session types. There are a few technical differences between our polymorphic endpoint types and the session types in [8]: we unify external and internal choices respectively with input and output operations, so as to model Singularity contracts more closely; we admit recursive endpoint types, while the support for recursion was only informally sketched in [8]. This led us to define subtyping coinductively, rather than by means of an inductive deduction system. Finally, we dropped lower type bounds and preferred to work with a restricted language of well-formed endpoint types which we claim to be appropriate in practice. Another work where session types are enriched with bounded polymorphism is [6], but in that case polymorphism is restricted to data, while in [8] and in the present paper type variables range over behaviors as well. Interestingly, all the critical endpoint types in [2] that violate the "finite-weight" restriction are recursive. This is no longer the case when (bounded) polymorphism is added and in fact there are finite endpoint types that can cause memory leaks if sent as messages.

A radically different approach for the static analysis of Singularity processes is given by [17, 18], where the authors develop a proof system based on a variant of *separation logic* [14]. However, leaks in [17] manifest themselves only when both endpoints of any channel have been closed. In particular, it is possible to prove that the code fragment (1) is correct, although it does indeed leak some memory. This problem has been subsequently recognized and solved in [16]. Roughly, the solution consists in forbidding the output of a message unless it is possible to prove (in the logic) that the queue that is going to host the message is reachable from the content of the message itself. In principle this condition is optimal, in the sense that it should permit every safe output. However, it relies on the knowledge of the identity of endpoints, that is a very precise information that is not always available. For this reason, [16] also proposes an approximation of this condition, consisting in tagging endpoints of a channel with distinct *roles* (basically, what are called *importing* and *exporting* ends in Singularity). Then, an endpoint can be safely sent as a message only if its role matches the one of the endpoint on which it is sent. This solution is incomparable to the one we advocate – restricting the output to endpoints with finite-weight type – suggesting that it may be possible to work out a combination of the two.

There exist a few works on session types [1, 4] that guarantee a global progress property for well-typed systems where the basic idea is to impose an order on channels to prevent circular dependencies that could lead to a deadlock. Not surprisingly, the critical processes (such as (1)) that we rule out

thanks to the finite-weight restriction on the type of messages are ill typed in these works. It turns out that a faithful encoding of (1) into the models proposed in these works is impossible, because the **open**$(\cdot,\cdot)$ primitive we adopt creates *both* endpoints of a channel within the same process, while the session initiation primitives in [1, 4] associate the fresh endpoints of a newly opened session to different processes running in parallel. This invariant – that the same process cannot own more than one endpoint of the same channel – is preserved in well-typed processes because of a severe restriction: whenever an endpoint c is received, the continuation process cannot use any endpoint other than c and the one from which c was received.

6   Conclusion and future work

In [2] we have formalized CoreSing♯, a core language of software isolated processes that communicate through copyless message passing. Well-typed processes are shown to be free from faults, leaks, and communication errors. In the present paper we have extended the type system of CoreSing♯ with bounded polymorphism, on the lines on what has been done for session types in [8]. Bounded polymorphism increases the expressiveness of types and improves modularity and reusability of components. In our setting, where resources – and endpoints in particular – are *linear*, we claim that bounded polymorphism is even more useful in order to avoid the loss of type information that occurs when endpoints are delegated and therefore exit the scope of the sender process (Section 2). We have shown that the *finite-weight* restriction we introduced in [2] scales smoothly to the richer type language, despite the fact that polymorphism augments the critical situations in which a leak may occur (recursive types are no longer necessary to find apparently well-typed processes that leak memory, as shown in Section 1). This is mostly due to a nice property of weights (Proposition 4.3). Unlike [2], we have omitted cells and open cell types, basically because they can be encoded thanks to the increased expressiveness given by (bounded) polymorphism (Examples 3.1 and 4.1).

With respect to Sing♯, our model still differs in a number of ways: first of all, PolySing♯ processes are terms of a process algebra, while Sing♯ is an imperative programming language similar to C♯. As a consequence, there is no direct mapping of Sing♯ programs onto PolySing♯ processes, even though we claim that our calculus captures all of the peculiar features of Sing♯, namely the explicit memory management (with respect to the exchange heap), the controlled ownership of memory allocated in the exchange heap, and the contract-based communication primitives. Second, in [2] and in the present paper we do not deal with mutable record structures nor with mutable arrays but only with mutable cells. The extension to the general framework is straightforward and does not pose new technical problems. Finally, the fact that we do not take into account non-linear values is indeed a limitation of the model presented here, but we plan to extend it in this direction as described next.

We envision two developments for this work. The first one is to enrich the type system with non-linear types, those denoting resources (such as permanent services) that can be shared among processes. Even though it is plausible to think that the technical details are relatively easy to work out (non-linear types are supported by [8] and in most other session type theories) we see this enhancement as a necessary step for PolySing♯ to be a useful model of Singularity processes. The second development, which looks much more challenging, is the definition of an algorithm for deciding subtyping (Definition 4.2). As observed in [8], bounded polymorphic session types share many properties with the type language in the system $F_{<:}$ [3]. Therefore, while it is reasonable to expect that properties and algorithms for extensions of $F_{<:}$ with recursive types [5] carry over to our setting, the exact details may vary.

References

[1] Lorenzo Bettini, Mario Coppo, Loris D'Antoni, Marco De Luca, Mariangiola Dezani-Ciancaglini & Nobuko Yoshida (2008): *Global Progress in Dynamically Interleaved Multiparty Sessions*. In: *CONCUR'08*, LNCS 5201, Springer, pp. 418–433, doi:10.1007/978-3-540-85361-9_33.

[2] Viviana Bono, Chiara Messa & Luca Padovani (2011): *Typing Copyless Message Passing*. In: *ESOP'11*, LNCS 6602, Springer, pp. 57–76, doi:10.1007/978-3-642-19718-5_4.

[3] Luca Cardelli, Simone Martini, John C. Mitchell & Andre Scedrov (1994): *An Extension of System F with Subtyping*. Information and Computation 109(1/2), pp. 4–56, doi:10.1006/inco.1994.1013.

[4] Giuseppe Castagna, Mariangiola Dezani-Ciancaglini, Elena Giachino & Luca Padovani (2009): *Foundations of Session Types*. In: *PPDP'09*, ACM, pp. 219–230, doi:10.1145/1599410.1599437.

[5] Dario Colazzo & Giorgio Ghelli (2005): *Subtyping, recursion, and parametric polymorphism in kernel Fun*. Information and Computation 198(2), pp. 71–147, doi:10.1016/j.ic.2004.11.003.

[6] Mariangiola Dezani-Ciancaglini, Sophia Drossopoulou, Elena Giachino & Nobuko Yoshida (2007): *Bounded Session Types for Object-Oriented Languages*. In: *FMCO'06*, LNCS 4709, Springer, pp. 207–245, doi:10.1007/978-3-540-74792-5_10.

[7] Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen Hunt, James R. Larus & Steven Levi (2006): *Language support for fast and reliable message-based communication in Singularity OS*. In: *EuroSys'06*, ACM, pp. 177–190, doi:10.1145/1217935.1217953.

[8] Simon Gay (2008): *Bounded polymorphism in session types*. Mathematical Structures in Computer Science 18(5), pp. 895–930, doi:10.1017/S0960129508006944.

[9] Simon Gay & Vasco T. Vasconcelos (2010): *Linear type theory for asynchronous session types*. Journal of Functional Programming 20(01), pp. 19–50, doi:10.1017/S0956796809990268.

[10] Kohei Honda (1993): *Types for dyadic interaction*. In: *CONCUR'93*, LNCS 715, Springer, pp. 509–523, doi:10.1007/3-540-57208-2_35.

[11] Kohei Honda, Vasco T. Vasconcelos & Makoto Kubo (1998): *Language primitives and type disciplines for structured communication-based programming*. In: *ESOP'98*, LNCS 1381, Springer, pp. 122–138, doi:10.1007/BFb0053567.

[12] 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 & Brian Zill (2005): *An Overview of the Singularity Project*. Technical Report MSR-TR-2005-135, Microsoft Research.

[13] Microsoft (2004): *Singularity design note 5: Channel contracts*. Technical Report, Microsoft Research. Available at http://www.codeplex.com/singularity.

[14] Peter W. O'Hearn, John C. Reynolds & Hongseok Yang (2001): *Local Reasoning about Programs that Alter Data Structures*. In: *CSL'01*, LNCS 2142, Springer, pp. 1–19, doi:10.1007/3-540-44802-0_1.

[15] Zachary Stengel & Tevfik Bultan (2009): *Analyzing singularity channel contracts*. In: *ISSTA'09*, ACM, pp. 13–24, doi:10.1145/1572272.1572275.

[16] Jules Villard (2011): *Heaps and Hops*. Ph.D. thesis, Laboratoire Spécification et Vérification, ENS Cachan, France.

[17] Jules Villard, Étienne Lozes & Cristiano Calcagno (2009): *Proving Copyless Message Passing*. In: *APLAS'09*, LNCS 5904, Springer, pp. 194–209, doi:10.1007/978-3-642-10672-9_15.

[18] Jules Villard, Étienne Lozes & Cristiano Calcagno (2010): *Tracking Heaps That Hop with Heap-Hop*. In: *TACAS'10*, LNCS 6015, Springer, pp. 275–279, doi:10.1007/978-3-642-12002-2_23.