Typing Copyless Message Passing

Viviana Bono, Chiara Messa, and Luca Padovani

Dipartimento di Informatica, Università di Torino, Italy

Abstract. We present a calculus that models a form of process interaction based on copyless message passing, in the style of Singularity OS. The calculus is equipped with a type system ensuring that well-typed processes are free from faults, leaks, and communication errors. The type system is essentially linear, but we show that linearity alone is inadequate. On the one hand, it is too strict when dealing with heap-allocated objects; on the other hand, it leaves room for scenarios where well-typed processes leak significant amounts of memory. We address these problems using dedicated types for keeping track of dereferenced pointers and by basing the type system upon an original variant of session types.

1 Introduction

Singularity OS is the prototype of a dependable operating system where software-isolated processes (SIPs) run in the same address space. Process interaction occurs solely through the exchange of messages over asynchronous, FIFO channels consisting of a pair of related endpoints (the channel *peers*), such that any message sent over one of the endpoints is received on the other endpoint. The communication overhead is tamed by copyless message passing: only *pointers* to messages are physically transferred from one process to another. Static analysis guarantees *process isolation*, namely that every process can only access memory it owns exclusively.

In this paper, we present CoreSing#, a calculus that captures the essential features of Singularity. We equip the calculus with a type system ensuring that well-typed processes are free from communication errors, memory faults, and memory leaks. We avoid communication errors by associating each endpoint with an endpoint type specifying the sequence of input/output actions allowed on that endpoint. Endpoint types are a variant of session types [7,8,13] tailored to the Singularity communication model. For example, the recursive endpoint type

$$T = \mathsf{rec}\, X.!\{\mathtt{arg}.?\{\mathtt{ack}.X\}, \mathtt{quit}.\mathsf{end}\}$$

indicates that a process can send an arbitrary number of `arg`-tagged messages and can close the endpoint only after it has sent a `quit`-tagged message. After every `arg`-tagged message, the process must be ready to receive an `ack`-tagged message coming from the process using the peer endpoint. Communication errors are avoided by imposing that peer endpoints are typed by *dual* session types. For example, the dual of T is $\overline{T} = \mathsf{rec}\, X.?\{\mathtt{arg}.!\{\mathtt{ack}.X\}, \mathtt{quit}.\mathsf{end}\}$, which specifies complementary actions with respect to those occurring in T.

To avoid memory faults and memory leaks, our type system relies essentially on the *linear usage* of pointers, although linearity alone proves to be inadequate. As a matter of fact, linearity is not enough to guarantee the absence of leaks. For instance the function

```
void foo() { (e, f) = open(); send(arg, f, e); close(f); }
```

creates the two endpoints `e` and `f` of a channel, sends `e` as the argument of an `arg`-tagged message on `f`, and closes `f`. This function uses `e` and `f` linearly and it is typable by associating `e` and `f` with suitable endpoint types. Yet, `foo` leaks some memory: after the **close** instruction, no reference to `e` is left to the process, therefore the `arg`-tagged message will never be deallocated. This memory leak can be avoided by imposing a simple restriction on the endpoint types. The idea is to define a notion of *weight* for endpoint types which roughly gives the "size" 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 `foo` is typable only if endpoint types with infinite weight are allowed.

From another point of view, linearity is too restrictive and must be relaxed, since *using* a pointer does not necessarily mean *consuming* the resource it points to. For example, the same endpoint can be used multiple times, as specified by its endpoint type. We must also deal with cases where a process owns a cell (that is, a one-field structure) but not its content. This happens after executing the instruction

```
send(arg, e, *a);
```

which sends the content of `a` in an `arg`-tagged message on endpoint `e`. The process performing the **send** no longer owns (the memory located at) `*a` after this instruction, but it still owns (the memory located at) `a`. We address this issue by distinguishing *cell types* $*t$ from *exposed cell types* $*\bullet$. The former ones denote memory cells that a process owns together with their content (of type t); the latter ones denote memory cells that a process owns, but whose content is owned by a different process. In the code fragment above, `a` has type $*t$ for some t *before* the **send**, and type $*\bullet$ *after* the **send**. This prevents `a` from being dereferenced multiple times, until the process regains ownership of its content, typically by assigning some value it owns into `*a`.

CoreSing# can be seen as a formalization of the language Sing#, which is an extension of C# specifically tailored to the implementation of Singularity processes. In particular, we provide a purely type-based framework to explain *channel contracts* and the **expose** construct in Sing# programs.

In Sing#, a channel contract describes an interaction pattern between the users of the two endpoints. For instance, the Sing# contract

```
contract C {
  initial state Transfer { !arg → WaitAck; !quit → End }
  state WaitAck { ?ack → Transfer }
  final state End { } }
```

describes a pattern where the process using the so-called *exporting endpoint* behaves as specified by the endpoint type T defined above, while the process using the *importing endpoint* behaves as specified by $\overline{T}$. There are clear analogies between contracts and endpoint types: the contract describes an interaction between two processes in terms of states and transitions, with a bias towards one of the two processes; the endpoint type describes the behavior of a single process involved in the interaction.

Regarding the **expose** construct, it is used by the Sing# compiler to keep track of memory ownership. In particular, Sing# allows pointer dereferentiation only within **expose** blocks. To illustrate the point, consider the following code fragments:

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

The effect of **expose**(a) is to 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 the fragment on the left hand side, *a is owned by the process before and after the **send**, even though the type associated with a might have changed since the **send** can possibly change the state of *a's contract. In the fragment on the right hand side, the process loses ownership of *a after the **send**, but it assigns the pointer to a newly allocated object into *a. The same assignment in the left fragment would cause a memory leak, while omitting the assignment in the right fragment would prevent the **expose** block from being well typed. Distinguishing between cells with type $*t$ and cells with type $*\bullet$ is all we need to capture the static semantics of **expose**(a) blocks in Sing#. At the beginning of the 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 a (possibly different) pointer that the process owns, thus turning a's type from $*\bullet$ back to some $*s$.

Related work. Session types have been introduced as a structuring mechanism in distributed systems, where processes engage into a conversation by first establishing a *private session* and then carrying on the conversation within the protected scope of the session. The session type prescribes, for each process involved in the session, the sequence and type of messages the process is allowed to send or expected to receive at each given time. Endpoint types are dyadic, "finite-weight" session types describing the exchange of heap pointers. There exist session type theories guaranteeing not only the absence of communication errors, but also the *global progress* of systems. While definitely interesting, we do not care about global progress in this work, since it can be achieved by means of orthogonal mechanisms, as shown for example in [3,1].

Aspects of the Singularity OS and of Sing# have already been formalized and studied elsewhere. In particular, [14] and [16] introduce general frameworks for reasoning on (pseudo-)linear access to shared memory. Regarding channel contracts, [12] shows that they are implementable without deadlocks if they are *deterministic* and *autonomous*. The first condition requires that there cannot be two transitions that differ only for the target state. The autonomous condition requires that every two transitions departing from the same state are either two sends or two receives. These conditions make it possible to split contracts into pairs of dual session types, and to fit existing session type theories in our setting in such a natural way. To keep session types simple, however, we do not model Singularity contracts where a final state has outgoing transitions. More expressive session types [2] could be adopted, but then we should presumably impose also the *synchronizing* restriction on channel contracts, as investigated in [14]. In [4] it was already observed that, to prevent inconsistencies related to the ownership of messages, special care is required when sending endpoints. There, the authors show that sending endpoints in a *send-state* condition is safe. Our investigation provides further motivations to restrict the endpoints that can be sent as messages. Also, our "finite-weight" condition generalizes the *send-state* condition in [4]. Other works [4,6] introduce apparently similar, "finite-weight" restrictions on session types to make sure that message queues of the corresponding channels are bounded. In our case, the weight concerns a different measure and the restriction prevents the creation of cycles involving channel queues in the heap (see Section 3 for details).

Our work shares many aspects with [14,15], where the authors develop a proof system for a significant fragment of Sing#. The proof system is based on a variant of *separation logic* [9,11] and permits the derivation of Hoare triples of the form $\{A\}\ p\ \{B\}$ where p is a program and A and B are logical formulas describing the state of the heap before and after the execution of p. A judgment $\{\mathsf{emp}\}\ p\ \{\mathsf{emp}\}$ indicates that if p is executed in the empty heap (the pre-condition emp), then it leaks no memory (the post-condition emp). However, leaks in [14] manifest themselves only when both endpoints of any channel used in p have been closed. In particular, it is possible to derive the judgment $\{\mathsf{emp}\}$ `foo()` $\{\mathsf{emp}\}$ for the function `foo` we have shown above, although it is intuitively clear that `foo` does indeed leak some memory. We tighten the definition of "leak" and we are able to declare `foo` ill typed. Endpoint types allow us to easily detect communication errors, while this aspect is still to be investigated in the setting of [14].

Structure of the paper. In Section 2 we present the syntax and semantics of CoreSing# and we formalize faults, leaks, and communication errors. Section 3 presents the type system for CoreSing# and the soundness results. In Section 4 we briefly sketch some extensions of the type system which are supported by Sing#. We conclude in Section 5.

2 Syntax and Semantics of CoreSing#

Let us 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*. In the following, we will write $\tilde{a}$, $\tilde{x}$, $\tilde{u}$, ... for denoting sequences of pointers, variables, and names, respectively. We will sometimes treat $\tilde{u}$ as the set of names occurring in the sequence $\tilde{u}$.

The language of processes, defined by the grammar in Table 1, essentially is a pi-calculus equipped with tag-based message dispatching and primitives for handling heap-allocated objects (cells and endpoints). To keep the model manageable and the presentation within the page limits we only consider cells, but multi-field structures can be accommodated with reasonable effort. The process $\mathbf{0}$ is idle and performs no action. Terms **rec** $X.P$ and X are used for building recursive processes, as usual. The process $u!\mathtt{m}\langle\tilde{u}\rangle.P$ sends a message $\mathtt{m}\langle\tilde{u}\rangle$ on the endpoint u and continues as P. A *message* is made of a *tag* m along with its *parameters* $\tilde{u}$. The process $\sum_{i\in I} u?\mathtt{m}_i(\tilde{x}_i).P_i$ waits for a message from the endpoint u. The tag $\mathtt{m}_i$ of the received message determines the continuation P_i where the variables x_i are instantiated with the parameters of the message. The process **cell**$(a,u).P$ creates a new cell located at a, initializes it with u, and continues as P. Similarly, **open**$(a,b).P$ creates a *channel*, represented as a pair of endpoints a and b. We will say that b is the peer endpoint of a and vice-versa. The process **free**(u) declares that the object located at u is no longer in use. The process **expose**$(u,x).P$ reads the content of the cell located at u and binds it to x in the continuation P, while **unexpose**$(u,v).P$

assigns v to the cell located at u. These primitives are named **expose** and **unexpose** because they are strictly related with the **expose** construct in Singularity. In Section 3 we will see that they also represent complementary scoping primitives. 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.

Table 1. Syntax of CoreSing# processes and of heap objects

$P ::=$		**Process**	$\mu ::=$		**Heap**
	$\mathbf{0}$	(idle)		$\emptyset$	(empty)
$\mid$	X	(variable)	$\mid$	$a \mapsto a$	(cell)
$\mid$	$\mathbf{free}(u)$	(garbage)	$\mid$	$a \mapsto [a,q]$	(endpoint)
$\mid$	$\mathbf{open}(a,a).P$	(open channel)	$\mid$	μ,μ	(composition)
$\mid$	$\mathbf{cell}(a,u).P$	(create cell)			
$\mid$	$\mathbf{expose}(u,x).P$	(expose cell)	$q ::=$		**Queue**
$\mid$	$\mathbf{unexpose}(u,u).P$	(unexpose cell)		ε	(empty)
$\mid$	$u!\mathtt{m}\langle\tilde{u}\rangle.P$	(send)	$\mid$	$\mathtt{m}\langle\tilde{a}\rangle$	(message)
$\mid$	$\sum_{i\in I} u?\mathtt{m}_i(\tilde{x}_i).P_i$	(receive)	$\mid$	$q :: q$	(composition)
$\mid$	$P \oplus P$	(choice)			
$\mid$	$P \mid P$	(composition)			
$\mid$	$\mathbf{rec}\ X.P$	(recursion)			

The sets of free and bound names of every process P, respectively denoted by $\mathrm{fn}(P)$ and $\mathrm{bn}(P)$, are almost standard. Beware that the process $\mathbf{cell}(a,u).P$ binds a but not u, which is thus free. The construct **rec** X is the only binder for process variables. We restrict recursive processes so that in every term **rec** $X.P$ we have $\mathrm{fn}(P) \cap \mathrm{bn}(P) = \emptyset$. This makes sure that in the unfolding $P\{\mathbf{rec}\ X.P/X\}$ of a recursive process no name occurring free in P is accidentally captured.

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 $a \mapsto b$ is made of a cell located at a that contains b; 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 $\mathrm{dom}(\mu)$ and $\mathrm{dom}(\mu')$, 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 $\mathtt{m}_1\langle\tilde{c}_1\rangle :: \cdots :: \mathtt{m}_n\langle\tilde{c}_n\rangle$. 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, which equates processes we never want to distinguish, and a reduction relation. Structural congruence, denoted by $\equiv$, is the least congruence relation that includes alpha conversion on bound names, commutativity and associativity of $|$, and the law $P \mid \mathbf{0} \equiv P$.

CoreSing# processes never interact directly with each other: every action performed by a process has some effect (or depends) on the heap, and processes communicate with

Table 2. Reduction of systems

(R-CELL)
$$(\mu;\textsf{cell}(a,b).P) \to (\mu, a \mapsto b; P)$$

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

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

(R-EXPOSE)
$$(\mu, a \mapsto b; \textsf{expose}(a,x).P) \to (\mu, a \mapsto b; P\{b/x\})$$

(R-REC)
$$(\mu; \textsf{rec}\, X.P) \to (\mu; P\{\textsf{rec}\, X.P/X\})$$

(R-UNEXPOSE)
$$(\mu, a \mapsto b; \textsf{unexpose}(a,c).P) \to (\mu, a \mapsto c; P)$$

(R-SEND)
$$(\mu, a \mapsto [b,q], b \mapsto [a,q']; a!\mathtt{m}\langle\tilde{c}\rangle.P) \to (\mu, a \mapsto [b,q], b \mapsto [a, q' :: \mathtt{m}\langle\tilde{c}\rangle]; P)$$

(R-RECEIVE)
$$\frac{k \in I}{(\mu, a \mapsto [b, \mathtt{m}_k\langle\tilde{c}\rangle :: q]; \sum_{i\in I} a?\mathtt{m}_i(\tilde{x}_i).P_i) \to (\mu, a \mapsto [b,q]; P_k\{\tilde{c}/\tilde{x}_k\})}$$

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

(R-PAR)
$$\frac{(\mu;P) \to (\mu';P')}{(\mu; P \mid Q) \to (\mu'; P' \mid Q)}$$

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 $\to$, inductively defined in Table 2, is described in the following paragraph. (R-CELL) and (R-OPEN) respectively create a new cell and a new channel. The channel consists of two fresh endpoints which refer to each other and have an empty queue. The cell is properly initialized with the pointer specified in the process. Both reductions are possible provided that the newly introduced pointers do not already occur in $\mathrm{dom}(\mu)$, for otherwise the heap in the resulting system would be undefined. (R-CHOICE) (and its symmetric, omitted) is a standard choice rule, saying that a process $P \oplus Q$ may autonomously reduce to either P or Q leaving the heap unchanged. For the sake of simplicity, we do not specify the test performed by the process, since it is irrelevant for the investigation we are pursuing. (R-EXPOSE) reads the content of a cell and binds it to a variable in the continuation process. As usual, $P\{b/x\}$ denotes the capture-avoiding substitution of every free occurrence of x in P with b. (R-UNEXPOSE) describes the assignment of a pointer c to the content of a cell pointed to by a whose previous content was b. The pointers b and c may be equal, in which case the reduction is a no-op. (R-SEND) describes the sending of a message $\mathtt{m}\langle\tilde{c}\rangle$ on the endpoint a. The message is enqueued at the end of a's peer endpoint queue. (R-RECEIVE) describes a process waiting for a message from the endpoint a. The message at the front of a's queue is removed from the queue, its tag is used for selecting some branch $k \in I$, and its parameters instantiate the variables $\tilde{x}_k$. If the queue is not empty and the first message in the queue does not match any of the tags $\{\mathtt{m}_i \mid i \in I\}$, then no reduction occurs and the process is stuck. The rule implicitly assumes that the sequence $\tilde{c}$ of parameters in the message and the sequence $\tilde{x}_k$ of variables in the process have equal length. This will be enforced by the type system in Section 3. (R-STRUCT) describes reductions modulo structural congruence. It plays an essential role in ensuring that $\textsf{cell}(a,b).P$ and $\textsf{open}(a,b).P$ are never stuck, because a and b can always be alpha converted to some pointers not occurring in $\mathrm{dom}(\mu)$. Finally, (R-PAR) expresses reductions under parallel composition. Observe that the heap is treated globally, even when it is only a sub-process to reduce. There is no reduction for $\textsf{free}(a)$ processes: it is technically convenient to treat them as persistent processes so that we can easily track which pointers have been properly deallocated. A process willing to deallocate a pointer a and to continue as P afterwards can be modelled as $\textsf{free}(a) \mid P$. In the following we write $\Rightarrow$ for the reflexive, transitive closure of $\rightarrow$, and we write $(\mu;P) \nrightarrow$ if there exist no μ' and P' such that $(\mu;P) \rightarrow (\mu';P')$.

In this work we focus on three properties of systems: we wish every system to be fault free, where 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; we wish every system to be leak free, where 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; finally, we wish every system to enjoy (a limited form of) progress, meaning that no process in the system should get stuck while reading messages from a non-empty queue. We conclude this section by making these properties precise, using CoreSing# as the reference calculus.

Before we move on, we need to formalize 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 $\mathrm{fn}(P)$ (we can think of the pointers in $\mathrm{fn}(P)$ as of the local variables of the process stored in its stack); from these pointers, the process may reach other heap objects by exposing cells and by reading messages from the queue of the endpoints it can reach.

Definition 2.1 (Reachable pointers). *Let $A \subseteq \texttt{Pointers}$. We write $\mathrm{reach}(A,\mu)$ for the least set A' that includes A and such that:*

- *$a \in A'$ and $a \mapsto b \in \mu$ implies $b \in A'$;*
- *$a \in A'$ and $a \mapsto [b, q :: \mathtt{m}\langle\tilde{c}\rangle :: q'] \in \mu$ implies $\tilde{c} \subseteq A'$.*

Observe that the peer of an endpoint located at a may not be reachable, because the calculus has no primitive operator to access the b component in the heap $a \mapsto [b,q]$. We now define well-behaved systems formally.

Definition 2.2 (Well-behaved process). *We say that the process P is* well behaved *if $(\emptyset;P) \Rightarrow (\mu;Q)$ implies: (1) $\mathrm{fn}(Q) \subseteq \mathrm{dom}(\mu)$; (2) $\mathrm{dom}(\mu) \subseteq \mathrm{reach}(\mathrm{fn}(Q),\mu)$; (3) $Q \equiv P_1 \mid P_2$ implies $\mathrm{fn}(P_1) \cap \mathrm{fn}(P_2) = \emptyset$; (4) $Q \equiv P_1 \mid P_2$ and $(\mu;P_1) \nrightarrow$ where P_1 does not have unguarded parallel compositions imply either $P_1 = \mathbf{0}$ or $P_1 = \textsf{free}(a)$ or $P_1 = \sum_{i\in I} a?\mathtt{m}_i(x_i).P_i$ where the queue in the endpoint located at a is empty.*

Let us comment on the conditions (1–4) in Definition 2.2 and see how they capture the desirable properties discussed earlier. *Absence of faults* is formalized as the combination of conditions (1), (3), and (4). Indeed, (1) ensures that any pointer that is directly reachable refers to an allocated object. Since Definition 2.2 quantifies over *every* possible reduction of P, condition (1) must hold for every pointer that is indirectly reachable. Condition (3) implies that no well-behaving process can access a deallocated object or can deallocate the same object twice, because in CoreSing# we keep track of deallocated objects by means of persistent $\textbf{free}(u)$ processes. For example, the process $\textbf{expose}(a,x).\textbf{expose}(a,y).(\textbf{free}(a) \mid \textbf{free}(x) \mid \textbf{free}(y))$ violates condition (3) because it deallocates the content of the cell located at a twice. Finally, condition (4) ensures that a stable, atomic process cannot be an attempt to (**un**)**expose** an endpoint or to send or receive a message using a pointer that refers to a cell. *Absence of leaks* is formalized as condition (2), requiring that the set of pointers reachable from Q must include the whole domain of the heap. The processes $\textbf{open}(a,b).\mathbf{0}$ and $\textbf{expose}(a,x).\textbf{free}(x)$ are simple examples of violation of condition (2). *Progress* is formalized as condition (4), implying that if $(\mu;Q)$ is a stuck configuration, then 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. The same condition requires that a deallocated endpoint must have an empty queue.

3 Type System

Types. Types describe the nature of objects allocated in the heap. According to Table 1, the heap contains *cells* and *endpoints*, therefore we want to discriminate these entities using different types. Since endpoints can be used for complex interactions involving messages with different tags, the type of endpoints will be a structured term describing a protocol, in the same spirit of *session types*. Also, we will discriminate between cells that have been exposed from those that have not. For the sake of minimality we only focus on types for linear entities. Non-linear entities, which are essential in practice, can be added by means of orthogonal extensions of the type system, as shown e.g. in [13].

Table 3. Syntax of types

$t ::=$		**Type**	$T ::=$			**Endpoint Type**
	$*t$	(cell type)			**end**	(termination)
	$\mid\ *\bullet$	(exposed cell type)		$\mid$	X	(variable)
	$\mid\ T$	(endpoint type)		$\mid$	$!\{\mathtt{m}_i.T_i\}_{i\in I}$	(internal choice)
				$\mid$	$?\{\mathtt{m}_i.T_i\}_{i\in I}$	(external choice)
				$\mid$	$\textbf{rec}\, X.T$	(recursive type)

Types, ranged over by t, s, $\dots$, and endpoint types, ranged over by T, S, $\dots$, are defined in Table 3. The *cell type* $*t$ describes a cell whose content has type t; the *exposed cell type* $*\bullet$ also describes a cell, but it says nothing about the content of the cell and therefore makes the cell visible but inaccessible. Endpoint types describe the sequence of actions that processes can perform on a given endpoint. The endpoint type **end** describes an endpoint on which no further action, save for deallocation, is possible. The internal choice $!\{\mathtt{m}_i.T_i\}_{i\in I}$ describes an endpoint on which a process can send any message with tag $\mathtt{m}_i$ for $i \in I$. Depending on the tag $\mathtt{m}_i$ of the message, the process must then use the endpoint according to the endpoint type T_i. The external choice $?\{\mathtt{m}_i.T_i\}_{i\in I}$ describes an endpoint from which a process is supposed to receive a message which is guaranteed to have a tag $\mathtt{m}_i$ for some $i \in I$. After the message is received, the process must use the endpoint according to the endpoint type T_i. In both choices we assume that $I \neq \emptyset$ and that $i, j \in I$ with $i \neq j$ implies $\mathtt{m}_i \neq \mathtt{m}_j$ (this corresponds to the deterministic condition for Sing# contracts). When I is a singleton, we will usually write $!\mathtt{m}.T$ instead of $!\{\mathtt{m}.T\}$, and similarly for external choices. We use endpoint variables and terms of the form $\textbf{rec}\, X.T$ for describing recursive protocols, as usual. For instance, $\textbf{rec}\, X.!\mathtt{m}.X$ describes an endpoint on which a process must send an infinite number of m-tagged messages. We forbid non-contractive endpoint types such as $\textbf{rec}\, X.X$ requiring that every recursion variable is guarded by an internal or external choice. We assume that a recursive endpoint type $\textbf{rec}\, X.T$ and its unfolding $T\{\textbf{rec}\, X.T/X\}$ are equal.

We assume a global environment Σ associating message tags with sequences of types. A judgment of the form $\Sigma \vdash \mathtt{m} : \langle t_1, \dots, t_n \rangle$ indicates that messages tagged by m have n parameters with type $t_1, \dots, t_n$, in this order. This obliges a sender of a m-tagged message to provide n parameters with these types and guarantees a receiver of a m-tagged message that the n parameters have these types.

A crucial notion regarding endpoint types is that of *duality*. Intuitively, two peer endpoints should be typed by dual endpoint types to guarantee that the communications on them occur without errors. For example, the endpoint types $!\{\mathtt{m}_i.\textbf{end}\}_{i\in I}$ and $?\{\mathtt{m}_i.\textbf{end}\}_{i\in I}$ are dual: every message sent by the sender process is accepted by the receiver. More generally, the dual of an endpoint type T, denoted by $\overline{T}$, is obtained by swapping internal and external choices, while **end** is invariant. Formally:

$$\overline{\textbf{end}} = \textbf{end} \qquad \overline{X} = X \qquad \overline{\textbf{rec}\, X.T} = \textbf{rec}\, X.\overline{T}$$
$$\overline{!\{\mathtt{m}_i.T_i\}_{i\in I}} = ?\{\mathtt{m}_i.\overline{T}_i\}_{i\in I} \qquad \overline{?\{\mathtt{m}_i.T_i\}_{i\in I}} = !\{\mathtt{m}_i.\overline{T}_i\}_{i\in I}$$

Typing the Heap. The heap plays a primary role, not just because it enables the interaction between processes, but also because 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 context 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 context. 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.

The context Δ we use for typing heaps (and processes) does not specify an association for every location of the heap, as this would be redundant. For example, the association $a : *t$ tells us not only that a is a pointer to a cell, but also that the content of the cell, which is another pointer, has itself type t. Therefore, the association $b : t$ is redundant, assuming that b is the content of the cell located at a, insofar the context contains the association $a : *t$. The context we use for typing a heap μ is defined on a proper subset of $\mathrm{dom}(\mu)$ which happens to be the set of pointers that the processes of the system have direct access to. The reader can think of $\mathrm{dom}(\Delta)$ as playing a similar role as the *roots* of a garbage collector. The composition of two contexts Δ_1 and Δ_2, denoted by Δ_1, Δ_2, is defined only when $\mathrm{dom}(\Delta_1) \cap \mathrm{dom}(\Delta_2) = \emptyset$.

Informally, well-typedness of a heap with respect to some context Δ entails the following properties:

(Consistency) Every allocated object is consistent with the corresponding type in the context. Hence, associations $a : *t$ and $a : *\bullet$ in the context correspond to cells $a \mapsto b$ in the heap; analogously, associations $a : T$ in the context correspond to endpoints $a \mapsto [b, q]$ in the heap such that each message in q has parameters that are consistent with the their type, as determined by the tag of the message; also, the tag is among the ones occurring in T (we will make this more precise shortly).

(Reachability) Every allocated object is reachable from exactly one of the pointers in $\mathrm{dom}(\Delta)$. This has two implications: first, there is no leak in a well-typed heap provided that every pointer in $\mathrm{dom}(\Delta)$ occurs in some process; second, a well-typed heap has no overlapping of the objects that can be reached from different roots, therefore each allocated object is "owned" by exactly one root. Since the roots will be distributed linearly among the processes in the system (different processes cannot share the same root) this immediately guarantees that different processes have access to disjoint regions of the heap.

(Duality) Peer endpoints have dual endpoint types. This invariant is needed for ensuring that communications occurring on peer endpoints are error free.

We split heap type checking in two parts: first we define a well-formedness relation $\Delta \Vdash \mu$, meaning that μ satisfies the consistency and reachability conditions we have informally described above. Then, we say that μ is well typed with respect to Δ when $\Delta \Vdash \mu$ holds and the duality condition is also satisfied. We proceed this way because consistency and reachability are more conveniently defined in terms of a deduction system, while duality is a global property that involves pairs of peer endpoints, which may be (and usually are) located in disjoint regions of the heap.

Table 4. Well-formedness rules for the heap

(T-Mem Empty)
$$\emptyset \Vdash \emptyset$$

(T-Mem Open)
$$a : *\bullet \Vdash a \mapsto b$$

(T-Mem Cell)
$$\frac{b : t \Vdash \mu}{a : *t \Vdash \mu, a \mapsto b}$$

(T-Mem Split)
$$\frac{\Delta_1 \Vdash \mu_1 \qquad \Delta_2 \Vdash \mu_2}{\Delta_1, \Delta_2 \Vdash \mu_1, \mu_2}$$

(T-Mem Endpoint)
$$\frac{\Sigma \vdash \mathtt{m}_i : \langle \tilde{t}_i \rangle \ {}^{(i=1..n)} \qquad \{\tilde{c}_i : \tilde{t}_i\}_{i=1..n} \Vdash \mu \qquad \mathrm{tail}(T, \mathtt{m}_1 \cdots \mathtt{m}_n) \text{ is defined}}{a : T \Vdash \mu, a \mapsto [b, \mathtt{m}_1\langle \tilde{c}_1 \rangle :: \cdots :: \mathtt{m}_n\langle \tilde{c}_n \rangle]}$$

The relation $\Vdash$ is defined in Table 4. In the derivation of a judgment $\Delta \Vdash \mu$ both the context Δ and the heap μ are treated linearly, implying that μ should contain all and only the objects that are reachable from the roots in Δ. Axiom (T-Mem Empty) states that the empty heap is well formed only with respect to the empty context. Axiom (T-Mem Open) states that the heap $a \mapsto b$ is well formed with respect to the context $a : *\bullet$ regardless of what b is associated with. Since the type system for processes will prevent access to b, the heap region rooted at b is not accessible if a is typed by $*\bullet$. Rule (T-Mem Cell) states that the heap containing $a \mapsto b$ is well formed with respect to the context $a : *t$ if it is well formed with respect to the context $b : t$.

Rule (T-MEM SPLIT) allows to check well formedness of a compound heap μ_1, μ_2 by means of a compound context Δ_1, Δ_2. Rule (T-MEM ENDPOINT) states that the heap $a \mapsto [b, q]$ is well formed with respect to the context $a : T$, provided that the heap rooted at the parameters of the messages in q is well formed with respect to the context determined by the tags of the messages, and that T does indeed specify that the messages in q are expected to be received *in the order in which they occur in* q. The last check is enforced by means of a partial function $\text{tail}(T, \mathtt{m}_1 \cdots \mathtt{m}_n)$, which is defined by induction on n as follows:

$$\text{tail}(T, \varepsilon) = T \qquad \frac{k \in I \qquad \text{tail}(T_k, \tilde{\mathtt{m}}) = S}{\text{tail}(?\{\mathtt{m}_i.T_i\}_{i \in I}, \mathtt{m}_k \tilde{\mathtt{m}}) = S}$$

We will use tail again to enforce an even stronger relation between the endpoint types of peer endpoints. For the time being, observe that if q is not empty, then T must begin with as many external choices as the number of messages in the queue. Therefore, in a well-formed heap every endpoint associated with an endpoint type **end** or $!\{\mathtt{m}_i.T_i\}_{i \in I}$ must have an empty queue.

Although the context Δ in a judgment $\Delta \Vdash \mu$ only specifies type associations for the roots of μ it will occasionally be useful to access the type associations for all of the pointers in $\text{dom}(\mu)$. To this purpose, we define a closure operator over contexts.

Definition 3.1 (Context closure). *Let $\Delta \Vdash \mu$. The* closure *of Δ with respect μ, denoted by $[\Delta; \mu]$, is the set of all the judgments $a : t$ that occur in the derivation of $\Delta \Vdash \mu$.*

The following result provides us with a sanity check on the closure of a context: when $\Delta \Vdash \mu$ the closure of Δ with respect to μ contains the associations for all the objects allocated in μ and every allocated object is reachable from $\text{dom}(\Delta)$.

Proposition 3.1. *If $\Delta \Vdash \mu$, then* $\text{dom}(\mu) = \text{dom}([\Delta; \mu]) \subseteq \text{reach}(\text{dom}(\Delta), \mu)$.

We are now ready to complete the definition of well-typed heap, by imposing additional constraints on peer endpoints.

Definition 3.2 (Well-typed heap). *We say that μ is* well typed *under Δ, notation $\Delta \vdash \mu$, if $\Delta \Vdash \mu$ and for every $a \mapsto [b, \mathtt{m}_1\langle\tilde{c}_1\rangle :: \cdots :: \mathtt{m}_n\langle\tilde{c}_n\rangle] \in \mu$ we have:*

1. $b \mapsto [a, \mathtt{m}'_1\langle\tilde{c}'_1\rangle :: \cdots :: \mathtt{m}'_m\langle\tilde{c}'_m\rangle] \in \mu$*;*
2. $\min\{n, m\} = 0$*;*
3. $\text{tail}(T, \mathtt{m}_1 \cdots \mathtt{m}_n) = \overline{\text{tail}(S, \mathtt{m}'_1 \cdots \mathtt{m}'_m)}$ *where $a : T \in [\Delta; \mu]$ and $b : S \in [\Delta; \mu]$.*

Item (1) requires that, for every endpoint $a \in \text{dom}(\mu)$, its peer is also in $\text{dom}(\mu)$. Item (2) states that at most one of the queues of peer endpoints can contain messages. Item (3) states that the residual endpoint types associated with peer endpoints after removing the prefixes determined by the tags of the messages in the corresponding queues must be dual. Observe that, by rule (T-MEM ENDPOINT), both $\text{tail}(T, \mathtt{m}_1 \cdots \mathtt{m}_n)$ and $\text{tail}(S, \mathtt{m}'_1 \cdots \mathtt{m}'_m)$ are defined and that duality is an involution ($\overline{\overline{T}} = T$), therefore the items hold for both peers of a channel.

Table 5. Typing rules for processes

$$\text{(T-IDLE)}\quad \Gamma;\emptyset \vdash \mathbf{0} \qquad\qquad \text{(T-VAR)}\quad \Gamma,\{X \mapsto \Delta\};\Delta \vdash X \qquad\qquad \text{(T-FREE CELL)}\quad \Gamma;u:*\bullet \vdash \mathbf{free}(u) \qquad\qquad \text{(T-FREE ENDPOINT)}\quad \Gamma;u:\mathbf{end} \vdash \mathbf{free}(u)$$

$$\text{(T-OPEN)}\quad \frac{\Gamma;\Delta,a:T,b:\overline{T} \vdash P}{\Gamma;\Delta \vdash \mathbf{open}(a,b).P} \qquad\qquad \text{(T-CELL)}\quad \frac{\Gamma;\Delta,a:*t \vdash P}{\Gamma;\Delta,u:t \vdash \mathbf{cell}(a,u).P} \qquad\qquad \text{(T-EXPOSE)}\quad \frac{\Gamma;\Delta,u:*\bullet,x:t \vdash P}{\Gamma;\Delta,u:*t \vdash \mathbf{expose}(u,x).P}$$

$$\text{(T-REC)}\quad \frac{\Gamma,\{X \mapsto \Delta\};\Delta \vdash P \qquad \mathrm{dom}(\Delta)=\mathrm{fn}(P)}{\Gamma;\Delta \vdash \mathbf{rec}\,X.P} \qquad\qquad \text{(T-UNEXPOSE)}\quad \frac{\Gamma;\Delta,u:*t \vdash P}{\Gamma;\Delta,u:*\bullet,v:t \vdash \mathbf{unexpose}(u,v).P}$$

$$\text{(T-CHOICE)}\quad \frac{\Gamma;\Delta \vdash P \qquad \Gamma;\Delta \vdash Q}{\Gamma;\Delta \vdash P \oplus Q} \qquad\qquad \text{(T-SEND)}\quad \frac{k \in I \qquad \Sigma \vdash \mathtt{m}_k : \langle \tilde{t} \rangle \qquad \Gamma;\Delta,u:T_k \vdash P}{\Gamma;\Delta,u:!\{\mathtt{m}_i.T_i\}_{i\in I},\tilde{v}:\tilde{t} \vdash u!\mathtt{m}_k\langle \tilde{v} \rangle.P}$$

$$\text{(T-PAR)}\quad \frac{\Gamma;\Delta_1 \vdash P \qquad \Gamma;\Delta_2 \vdash Q}{\Gamma;\Delta_1,\Delta_2 \vdash P \,|\, Q} \qquad\qquad \text{(T-RECEIVE)}\quad \frac{\Sigma \vdash \mathtt{m}_i : \langle \tilde{t}_i \rangle \ ^{(i\in I)} \qquad \Gamma;\Delta,u:T_i,\tilde{x}_i:\tilde{t}_i \vdash P_i \ ^{(i\in I)}}{\Gamma;\Delta,u:?\{\mathtt{m}_i.T_i\}_{i\in I} \vdash \sum_{i\in I} u?\mathtt{m}_i(\tilde{x}_i).P_i}$$

Typing Processes. We now turn our attention to the typing rules for processes, which are inductively defined in Table 5. Judgments have the form $\Gamma;\Delta \vdash P$ where Δ is the same context we use for typing heaps. The additional context Γ is used for typing recursive processes and therefore plays a role in two rules only, (T-VAR) and (T-REC). The unusual premise $\mathrm{dom}(\Delta) = \mathrm{dom}(P)$ in rule (T-REC) enforces a weak form of contractivity on recursive processes. It states that $\mathbf{rec}\ X.P$ is well typed under Δ only if P actually uses the names in $\mathrm{dom}(\Delta)$. Normally, divergent processes such as $\mathbf{rec}\ X.X$ can be typed in every context. If this were the case, the process $\mathbf{open}(a,b).\mathbf{rec}\ X.X$, which leaks a and b, would be well typed. The idle process is well typed in the empty context only. As we have seen in the typing rules for the heap, the empty context can only be used for typing the empty heap, therefore rule (T-IDLE) says that the terminated process has no leaks. Rules (T-FREE CELL) and (T-FREE ENDPOINT) state that a process $\mathbf{free}(u)$ is well typed provided that u is the only name owned by the process and that it corresponds to either an exposed cell (with type $*\bullet$) or to an endpoint on which no further interaction is possible (with type $\mathbf{end}$). Rules (T-OPEN) and (T-CELL) are used for respectively typing the creation of a new channel and of a new cell. In both cases the newly created pointers are visible in the continuation; in (T-OPEN) the rule guesses two dual endpoint types that, associated with the two endpoints, permits to type check the continuation; in (T-CELL), the name used for the cell initialization is discharged from the context and becomes unavailable in the continuation, unless the cell is exposed. Rules (T-CHOICE) and (T-PAR) are standard. Rule (T-SEND) states that a process $u!\mathtt{m}\langle \tilde{v} \rangle.P$ is well typed if u is associated with an endpoint type that permits the output of $\mathtt{m}$-tagged messages and the parameters of the message have the expected type as specified in the global context Σ. Also, the continuation P must be well typed in a context where the parameters have been discharged and the endpoint u is typed according to the endpoint type determined by the tag of the message. Rule (T-RECEIVE) deals with inputs: a process waiting for a message from an endpoint $a : ?\{\mathtt{m}_i.T_i\}_{i\in I}$ is well typed if it can deal with any $\mathtt{m}_i$-tagged message. The continuation process may use the endpoint according to the endpoint type T_i and can access the message's parameters. Rule (T-EXPOSE) states that the exposure of a cell is well typed only if the cell has not already been exposed (its type is $*t$ for some t and not $*\bullet$) and if the continuation correctly uses its content and the exposed cell itself. Rule (T-UNEXPOSE) shows that an exposed cell can be assigned any pointer that the process owns. Observe that the resulting type of the cell may differ from the one the cell had before exposure. This may happen for two reasons: either because the cell is assigned a different content having a different type, or because the type of the cell's content has changed, even if the content itself has not (endpoint types change over time, even though the pointer to them stays the same).

It turns out that the function `foo` in the introduction (properly encoded in CoreSing#) would be well typed in the empty context, as shown by the derivation below:

$$\dfrac{\dfrac{\Sigma \vdash \mathtt{m} : \langle ?\mathtt{m}.\mathbf{end}\rangle \qquad \dfrac{}{a : \mathbf{end} \vdash \mathbf{free}(a)}\ \text{(T-FREE ENDPOINT)}}{a : !\mathtt{m}.\mathbf{end}, b : ?\mathtt{m}.\mathbf{end} \vdash a!\mathtt{m}\langle b\rangle.\mathbf{free}(a)}\ \text{(T-SEND)}}{\vdash \mathbf{open}(a,b).a!\mathtt{m}\langle b\rangle.\mathbf{free}(a)}\ \text{(T-OPEN)}$$

As we have anticipated, this apparently harmless process produces a leak:

$$(\emptyset; \mathbf{open}(a,b).a!\mathtt{m}\langle b\rangle.\mathbf{free}(a)) \Rightarrow (a \mapsto [b,\varepsilon], b \mapsto [a,\mathtt{m}\langle b\rangle]; \mathbf{free}(a))$$

The endpoint b is no longer accessible by the process nor has it been properly deallocated. Furthermore, even if there were a mechanism for accessing b (for example, by peeking into the endpoint located at a) and for reading the message from b's queue, this would be catastrophic from the point of view of types: the actual type of b is $?\mathtt{m}.\mathbf{end}$, but as we remove the message from its queue it turns to $\mathbf{end}$. The b in the message, however, would maintain the "old" type $?\mathtt{m}.\mathbf{end}$, thus the actual and the inferred types for b would no longer coincide, with potentially catastrophic consequences. 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 $b \mapsto [a,\mathtt{m}\langle b\rangle]$ region of the heap needs not be owned by any process because it "owns" itself. In summary, we should tighten our type system and make sure that no cycle involving endpoint queues is created in the heap. In the process above this problem would not be too hard to detect, as the fact that a and b are peer endpoints is apparent from the syntax of the process. In general, however, detecting whether an endpoint is sent over its peer requires a runtime check, which is not a viable solution as we aim at static verification of processes.

An alternative approach for attacking the problem which does not require any change to the typing rules in Table 5 stems from the observation that infinite values (the endpoint located at b above fits well in this category) usually inhabit recursive types. The type of b is in fact recursive, although only implicitly: we have $b : T$ where $T : ?\mathtt{m}.\mathbf{end}$ and $\Sigma \vdash \mathtt{m} : \langle T\rangle$. Forbidding this implicit recursion would prevent the creation of cycles in the heap, but it would also unnecessarily restrict our language. For example, the endpoint type $S = !\mathtt{m}'.\mathbf{end}$ where $\Sigma \vdash \mathtt{m}' : \langle S\rangle$ would never cause the creation of cycles in the heap, even though it is implicitly recursive. The reason is that, if we are sending an endpoint $b : S$ over $a : S$, then its peer endpoint must have the dual type $?\mathtt{m}'.\mathbf{end}$ and therefore must be different from a. In general, cycles in the heap originate from non-empty queues, and non-empty queues are always associated with endpoints whose type begins with an external choice. The idea then is to give each endpoint type a *weight* that estimates the length of any chain of pointers originating from the queue. Endpoint types **end** and internal choices will always have a null weight, since the well formedness conditions for the heap already guarantee emptiness of the queues of the endpoints with these types.

The weight of a type is formally defined thus:

Definition 3.3 (Weight). *Let $\downarrow$ be the largest relation such that $t \downarrow n$ implies either:*

- $t = *\bullet$ *or* $t = \mathbf{end}$ *or* $t = !\{\mathtt{m}_i.T_i\}_{i\in I}$, *or*
- $t = *s$ *and* $s \downarrow n$, *or*
- $t = \mathbf{rec}\, X.T$ *and* $T\{t/X\} \downarrow n$, *or*
- $t = ?\{\mathtt{m}_i.T_i\}_{i\in I}$ *and* $n > 0$ *and* $\Sigma \vdash \mathtt{m}_i : \langle s_1, \dots, s_n\rangle$ *implies* $s_j \downarrow n-1$, *and* $T_i \downarrow n$ *for every* $i \in I$.

We define the weight *of a type t as* $\|t\| = \min\{n \in \mathbb{N} \cup \{\infty\} \mid t \downarrow n\}$.

Intuitively, $t \downarrow n$ means that the weight of t is bounded by n, hence $\|t\|$ is the least of t's bounds. In the examples we have discussed above, we have $\|T\| = \infty$ and $\|S\| = 0$. In the following we let $\|\tilde{t}\| = \max_{s\in\tilde{t}} \|s\|$. Observe that the weight of an endpoint type may change over time, as the endpoint type changes. For example $\|!\mathtt{m}.?\mathtt{m}.\mathbf{end}\| = 0$ and $\|?\mathtt{m}.\mathbf{end}\| = 1$ assuming $\Sigma \vdash \mathtt{m} : \langle\rangle$. For our purposes, the following property is particularly important:

Proposition 3.2. *Let* $\Sigma \vdash \mathtt{m}_i : \langle\tilde{t}_i\rangle$. *Then* $\max_{i\in I} \|\tilde{t}_i\| < \|?\{\mathtt{m}_i.T_i\}_{i\in I}\|$.

The reason why the restriction to types with finite weight is sufficient for proving the soundness of the type system lies in the following proposition relating the reachability of pointers and the weights of the corresponding types.

Proposition 3.3. *Let* $a : t \Vdash \mu$ *and* $b : s \in [a : t; \mu]$. *Then* $\|s\| \leq \|t\|$.

Proposition 3.3 is useful for deducing that some pointer b is not reachable from a. If $a : t \Vdash \mu$ and we know that b must be associated with some type s such that $\|t\| < \|s\|$, then we can conclude $b \not\in \mathrm{dom}([a : t; \mu])$. In the specific case of endpoints, suppose we are type checking a process $c!\mathtt{m}\langle a\rangle.\mathbf{0}$ where $a : t$. Clearly c must be associated with some endpoint type of the form $!\{\mathtt{m}_i.T_i\}_{i\in I}$ and its peer, say b, with the corresponding dual endpoint type $s = ?\{\mathtt{m}_i.\overline{T}_i\}_{i\in I}$. From Proposition 3.2 we deduce that $\|t\| < \|s\|$, therefore we can conclude $a \neq b$ by Proposition 3.3. In particular, we are not enqueueing b into its own queue.

We should remark that restricting endpoint types so that they have finite weight does not necessarily guarantee that the heap is free of cycles. For instance, both the following partially specified processes create cycles in the heap:

$$P_1 \stackrel{\text{def}}{=} \mathbf{open}(a,b).\mathbf{cell}(c,b).\mathbf{expose}(c,x).a!\mathtt{m}\langle c\rangle.Q_1$$
$$P_2 \stackrel{\text{def}}{=} \mathbf{open}(a,b).\mathbf{cell}(c,a).\mathbf{cell}(d,c).\mathbf{expose}(d,x).\mathbf{expose}(x,y).\mathbf{unexpose}(x,d).Q_2$$

and the reader may easily verify that it is possible to find appropriate Q_1 and Q_2 such that P_1 and P_2 are closed and well typed. The cycles created in these examples are harmless because the exposed cell type $*\bullet$ is a boundary which processes cannot trespass.

We can now present the subject reduction theorem followed by the soundness of the type system. Subject reduction is slightly non-standard, in the sense that types in the context may change as the process reduces. This is common in session type theories, since session types are behavioral types. In our case, also cell types can change (from $*t$ to $*\bullet$ and vice-versa). In addition, we need to type heaps as well as processes, and we write $\Delta \vdash (\mu;P)$ when $\Delta \vdash \mu$ and $\Delta \vdash P$.

Theorem 3.1 (Subject reduction). *Let $\Delta \vdash (\mu;P)$ and $(\mu;P) \rightarrow (\mu';P')$. Then $\Delta' \vdash (\mu';P')$ for some Δ'.*

In the statement of Theorem 3.1 we content ourselves of finding an appropriate context Δ' for typing the reduced process correctly as this is all one needs for proving the soundness of the type system. However, the proof of Theorem 3.1 necessarily relies on a stronger statement showing how Δ' is derived from Δ. Technically the most difficult part of the proof deals with regions of the heap that change owner: when a process $P_1 \mid P_2$ reduces because P_1 sends a message, the parameters of the message may transfer to the region of the heap owned by P_2. Thus, even though the context used for typing P_2 does not change (it is P_1 that reduces), the proof that the heap owned by P_2 is still well formed may change significantly. The finite-weight restriction on session types plays a crucial role to ensure that the new heap is well formed.

Theorem 3.2 (Soundness). *If $\vdash P$, then P is well behaved.*

We conclude this section with two remarks. The first one is that item (4) in Definition 2.2 is weaker than deadlock freedom. For example, the process

$$\mathbf{open}(a,b).a?\mathtt{m}().b!\mathtt{m}\langle\rangle.(\mathbf{free}(a) \mid \mathbf{free}(b))$$

is well typed assuming $a : ?\mathtt{m}.\mathbf{end}$ and $b : !\mathtt{m}.\mathbf{end}$ where $\Sigma \vdash \mathtt{m} : \langle\rangle$. This process deadlocks after the creation of the two endpoints, because it attempts at reading from endpoint a before any message is sent on its peer b. Incidentally, the example above shows that it is possible to have a deadlock also when only one channel is created.

The second remark regards the expressiveness of our framework, and in particular the possible limitations due to the finite-weight restriction we impose on types. The following example shows how to work around this restriction in a scenario where the use of types with infinite weight would be natural (a similar workaround was suggested in [6]).

Example 3.1 (Linear lists). We can represent a linear list as an endpoint from which one of two kinds of messages can be received: a `nil`-tagged message indicates that the list is empty; a `cons`-tagged message indicates that the list has at least one element, and the parameters of the message are the head of the list and its tail, which is itself a list. Reading a message from the endpoint corresponds to deconstructing the list and the tag-based dispatching of messages implements pattern matching. According to this intuition, the type of lists with elements of type t would be encoded as

$\mathtt{List}(t) = ?\{\mathtt{nil}.\mathbf{end}, \mathtt{cons}_t.\mathbf{end}\}$ where $\Sigma \vdash \mathtt{nil} : \langle\rangle$ and $\Sigma \vdash \mathtt{cons}_t : \langle t, \mathtt{List}(t)\rangle$, except that in this case we have $\|\mathtt{List}(t)\| = \infty$. To fix this, we require users of the list to signal the imminent deconstruction by sending a dummy, use-tagged message on the endpoint. This corresponds to defining $\mathtt{List}(t) = !\mathtt{use}.?\{\mathtt{nil}.\mathbf{end}, \mathtt{cons}_t.\mathbf{end}\}$ where $\Sigma \vdash \mathtt{use} : \langle\rangle$. One can now define syntactic sugar for creating and deconstructing linear lists, thus:

$$\begin{array}{rcl}
\mathtt{let}\ a = \mathtt{nil}\ \mathtt{in}\ P & \stackrel{\text{def}}{=} & \mathbf{open}(a,b).(P \mid b?\mathtt{use}().b!\mathtt{nil}\langle\rangle.\mathbf{free}(b)) \\
\mathtt{let}\ a = \mathtt{cons}_t\langle u,v\rangle\ \mathtt{in}\ P & \stackrel{\text{def}}{=} & \mathbf{open}(a,b).(P \mid b?\mathtt{use}().b!\mathtt{cons}_t\langle u,v\rangle.\mathbf{free}(b)) \\
\mathtt{match}\ u\ \mathtt{with} & \stackrel{\text{def}}{=} & u!\mathtt{use}\langle\rangle. \\
\mathtt{nil} \to P & & (\quad u?\mathtt{nil}().(\mathbf{free}(u) \mid P) \\
\mathtt{cons}_t(x,y) \to Q & & \quad + u?\mathtt{cons}_t(x,y).(\mathbf{free}(u) \mid Q) \quad)
\end{array}$$

The interested reader can verify that these processes are well typed . ■

4 Extensions

Non-linear usage of endpoints in output actions. An interesting feature of [14] is the possibility of sending an endpoint over itself. In our setting, this would be modeled as a process $P = u!\mathtt{m}\langle u\rangle.Q$ for some tag m with a parameter of the appropriate type. In [14] this feature actually plays a fundamental role: there the two endpoints of a channel must be closed simultaneously, hence it is natural to have a mechanism that reunites two peer endpoints by receiving one from the other. The process P above fails to type check according to the rules described in Section 3 because the endpoint u is used non-linearly both as the endpoint on which the communication occurs and as the message parameter. It is possible to relax the type system by adding the following rule:

$$\text{(T-SELF SEND)}\quad \frac{k \in I \qquad \Sigma \vdash \mathtt{m}_k : \langle \tilde{t}_1, T_k, \tilde{t}_2\rangle \qquad \Gamma;\Delta \vdash P}{\Gamma;\Delta, u : !\{\mathtt{m}_i.T_i\}_{i\in I}, \tilde{v}_1 : \tilde{t}_1, \tilde{v}_2 : \tilde{t}_2 \vdash u!\mathtt{m}_k\langle \tilde{v}_1, u, \tilde{v}_2\rangle.P}$$

There are two differences with respect to (T-SEND), all of which regard the endpoint u: first of all, u occurs twice, as the endpoint on which the message is sent as well as in the parameters of the message itself, possibly preceded and followed by more parameters. The two occurrences of u are treated differently with respect to types: the parameter u is preventively given type T_k, which corresponds to the correct type of u *after* the sending action has occurred. This is safe because, by the time the message is received, the send action has already completed. The second difference is that the endpoint u is no longer available in the continuation P, since it has been sent away.

Rule (T-SELF SEND) cannot compromise well formedness of the heap, in the sense that no cycles can be created even though the rule appears to embed some circularity. The reason is that, by sending an endpoint over itself, we can rest assured that the endpoint will be enqueued in the queue of a *different* endpoint, namely its peer.

Subtyping. The most common way to increase flexibility of a type system is to introduce a *subtyping* relation $\leqslant$ that establishes an (asymmetric) compatibility between different types: any value of type t can be safely used where a value of type s is expected when $t \leqslant s$. In the flourishing literature on session types several notions of subtyping relations have been investigated [5,2,13,10]. Here we show how to extend our model with a subtyping relation which is a straightforward adaptation of the subtyping defined in [5].

Definition 4.1 (Subtyping relation). *The* subtyping relation $\leqslant$ *is the largest relation such that* $t \leqslant s$ *implies:*

1. $t = *\bullet$ *implies* $s = *\bullet$*;*
2. $t = *t'$ *implies* $s = *s'$ *and* $t' \leqslant s'$*;*
3. $t = \mathbf{end}$ *implies* $s = \mathbf{end}$*;*
4. $t = !\{\mathtt{m}_i.T_i\}_{i\in I}$ *implies* $s = !\{\mathtt{m}_j.S_j\}_{j\in J}$ *with* $J \subseteq I$ *and* $T_j \leqslant S_j$ *for every* $j \in J$*;*
5. $t = ?\{\mathtt{m}_i.T_i\}_{i\in I}$ *implies* $s = ?\{\mathtt{m}_j.S_j\}_{j\in J}$ *with* $I \subseteq J$ *and* $T_i \leqslant S_i$ *for every* $i \in I$*.*

The key point for understanding $\leqslant$ is to focus on the concept of endpoint use: using an endpoint with type $\mathbf{end}$ means closing it; using an endpoint with type $!\{\mathtt{m}_i.T_i\}_{i\in I}$ means sending a message with tag $\mathtt{m}_k$ for some $k \in I$ and using the endpoint according to T_k thereafter; using an endpoint with type $?\{\mathtt{m}_i.T_i\}_{i\in I}$ means being ready to receive any message with tag $\mathtt{m}_i$ for every $i \in I$ and using the endpoint according to T_k thereafter. This leads to the invariance of $\mathbf{end}$, contravariance of internal choice, and covariance of external choice in Definition 4.1. For instance, it is safe to use an endpoint $u : !\{\mathtt{m}_i.T_i\}_{i\in I\cup J}$ where $v : !\{\mathtt{m}_i.T_i\}_{i\in I}$ is expected because the process using v will send a message with tag $\mathtt{m}_k$ for some $k \in I$. Then $k \in I \cup J$, hence the process is using u correctly according with its type. It may look surprising that $\leqslant$ is covariant with respect to cell types, namely that $t \leqslant s$ implies $*t \leqslant *s$. This relation is sound in our setting because access to the content of a cell is disciplined by means of open cell types: the type $*t$ only states the type of the values that can be *read* from the cell, not the type of the values that can be *written* in the cell.

Using this subtyping relation, we can relax the type system with a weaker rule for output actions, along with a subsumption rule:

$$\text{(T-SEND')}\ \frac{\Sigma \vdash \mathtt{m} : \langle \tilde{s} \rangle \qquad \tilde{t} \leqslant \tilde{s} \qquad \Gamma;\Delta, u : T \vdash P}{\Gamma;\Delta, u : !\mathtt{m}.T, \tilde{v} : \tilde{t} \vdash u!\mathtt{m}\langle \tilde{v} \rangle.P} \qquad \text{(T-SUB)}\ \frac{\Gamma;\Delta, u : S \vdash P \qquad T \leqslant S}{\Gamma;\Delta, u : T \vdash P}$$

In rule (T-SEND') the endpoint type associated with u precisely states the type of message that is sent and the actual parameters of the message are allowed to have a subtype of the expected type ($\tilde{t} \leqslant \tilde{s}$ is the point-wise extension of $\leqslant$ to tuples of types). Rule (T-SUB) is a bit unusual, since it allows to have a smaller type in the conclusion. This can be explained as the fact that the typing rules do not assign types to processes, and the subtyping relation changes the type of a name in the context.

We have seen in Section 3 that type weights play a crucial role in ensuring that the type theory is sound. In principle, since it is the (expected) type of the parameters of a message that determines the weight of the endpoint type where the message tag occurs, one might fear that subtyping could be exploited for circumventing the finiteness restriction we impose on the weight of types. This is not the case, and the rationale is remarkably simple. As the following proposition states, the weight of types changes in accordance with the subtyping relation, therefore any application of subtyping can only *decrease* the actual weight of types.

Proposition 4.1. *$t \leqslant s$ implies $\|t\| \leq \|s\|$.*

According to Definition 4.1, being a subtype does not necessarily mean being a “simpler” type. For example, we have $!\{\mathtt{m}_1.T_1, \mathtt{m}_2.T_2\} \leqslant !\mathtt{m}_1.T_1$. Yet, the weight of internal choices is zero, regardless of the number of (and the tags occurring in) branches. Had we assigned weights to endpoint types merely looking at the message tags occurring in them, as we initially suggested in Section 3, Proposition 4.1 would not hold.

Unrestricted closing of endpoints. A difference between Sing# and CoreSing# is that in the former language endpoints can be closed at any time while in CoreSing# this can only happen when their endpoint type is **end**. The restricted viewpoint we have adopted is consistent with the philosophy of session types, where the type of an endpoint describes not only the capabilities of a channel, but also the obligations that a process owning that channel is bound to. Sing# is more permissive about the closing of endpoints because, in a controlled environment such as Singularity, it might be desirable to abruptly terminate communications under exceptional circumstances.

If we relax the obligation to send messages on an endpoint when its type says so, we might introduce new deadlocks. Therefore we need to devise appropriate mechanisms for notifying processes that are blocked on input operations on closed endpoints and for spawning dedicated handlers when these situations occur. Closed endpoints are easily detectable in CoreSing# because of persistent **free**(a) processes. Therefore, we only need to annotate input processes thus

$$P \ ::= \ \cdots \ \mid \ \textstyle\sum_{i\in I} u?\mathtt{m}_i(\tilde{x}_i).P_i \ \mathtt{or}\ P \ \mid \ \cdots$$

by adding a *handler* branch ‘$\cdots$ or P’ which is selected if the queue associated with u is empty and the peer endpoint of u has been closed. This behavior is formalized by rule (R-RECEIVE CLOSE) in Table 6. In the reduced process a **free**(a) term appears, meaning that the endpoint used for input cannot be used by the handler.

This is not enough to avoid deadlocks caused by prematurely closed endpoints, because the queue of a closed endpoint may contain messages that that have other endpoints as parameters. Since no process can access these messages anymore, we must properly close every endpoint occurring in the queue of a closed endpoint. To this aim we enrich the reduction relation with rule (R-COLLECT), which formalizes a basic step of the garbage collector we have just described: any message $\mathtt{m}\langle\tilde{c}\rangle$ in the queue of a closed endpoint is dequeued and deallocated by a process **collect**$(\tilde{c} : \tilde{t})$ which is inductively defined at the bottom of Table 6. The process uses information provided by the type system to navigate through the structure of the objects to be deallocated. This will possibly fire rules (R-RECEIVE CLOSE) and (R-COLLECT) recursively.

Table 7 presents the revised typing rules for handling the unrestricted closing of endpoints. It is now possible to prove that well-typed processes are well behaved and that, in addition, if a process is blocked waiting for a message on some endpoint a it is because the peer endpoint of a has not been closed.

Table 6. Garbage collector and additional reduction rules

Reduction rules

(R-RECEIVE CLOSE)

$$(\mu, a \mapsto [b,\varepsilon]; \textstyle\sum_{i\in I} a?\mathtt{m}_i(\tilde{x}_i).P_i \;\mathtt{or}\; Q \mid \mathbf{free}(b)) \rightarrow (\mu, a \mapsto [b,\varepsilon]; Q \mid \mathbf{free}(a) \mid \mathbf{free}(b))$$

(R-COLLECT)

$$\frac{\Sigma \vdash \mathtt{m} : \langle \tilde{t} \rangle}{(\mu, a \mapsto [b, \mathtt{m}\langle \tilde{c} \rangle :: q]; \mathbf{free}(a)) \rightarrow (\mu, a \mapsto [b,q]; \mathbf{free}(a) \mid \mathbf{collect}(\tilde{c} : \tilde{t}))}$$

Garbage collector process

$$\begin{aligned} \mathbf{collect}(u : *\bullet) = \mathbf{collect}(u : T) &= \mathbf{free}(u) \\ \mathbf{collect}(u : *t) &= \mathbf{expose}(u,x).(\mathbf{free}(u) \mid \mathbf{collect}(x : t)) \\ \mathbf{collect}(u_1 : t_1, \dots, u_n : t_n) &= \mathbf{collect}(u_1 : t_1) \mid \cdots \mid \mathbf{collect}(u_n : t_n) \end{aligned}$$

Table 7. Updated typing rules for unrestricted endpoint closing

(T-FREE ENDPOINT')

$$\Gamma; u : T \vdash \mathbf{free}(u)$$

(T-RECEIVE')

$$\frac{\Sigma \vdash \mathtt{m}_i : \langle \tilde{t}_i \rangle \;^{(i\in I)} \qquad \Gamma; \Delta, u : T_i, \tilde{x}_i : \tilde{t}_i \vdash P_i \;^{(i\in I)} \qquad \Gamma; \Delta \vdash Q}{\Gamma; \Delta, u : ?\{\mathtt{m}_i.T_i\}_{i\in I} \vdash \sum_{i\in I} u?\mathtt{m}_i(\tilde{x}_i).P_i \;\mathtt{or}\; Q}$$

5 Conclusions

We have defined the static analysis for a calculus where processes communicate through the exchange of *pointers*. Verified processes are guaranteed to be free from memory faults, they do not leak memory, and do not fail on input actions. In this respect our work shares many objectives with [14], although our approach is *type-based*, whereas the one in [14] is *proof-based*. The proof-based approach relies on an expressive variant of separation logic and is therefore more general, in terms of properties of the heap that can be stated and verified. The type-based approach we have presented is tailored for guaranteeing the three properties above only, but it is simple to understand and efficient in practice (type checking and subtyping can be implemented in linear time).

Our type system has been inspired by session type theories, in a broad sense. The basic idea of session types, and of behavioral types in general, is that operating on a (linearly used) value may change its type, and thus the capabilities of that value thereafter. Endpoint types express the capabilities of endpoints, in terms of the type of messages that can be sent or received and in which order. Cell types and open cell types are, in a sense, behavioral types for (linear) heap cells: accessing the content of a cell changes its type from $*t$ to $*\bullet$; assigning the content of a cell does the inverse. In fact, it would be possible to encode cells and cell types in terms of endpoints and endpoint types, if only endpoint types were polymorphic.

The finite-weight restriction we impose on endpoint types is original to the best of our knowledge. Singularity restricts communications so that only endpoints in a *send-state*, those whose type begins with an internal choice, can be safely sent as messages.

The restriction is motivated by the implementation of ownership transfer in Singularity, where it is the sender's responsibility to explicitly tag sent messages with their new owner; therefore, a race can arise if the endpoint that the message is sent to changes owner (a detailed description can be found in [4]). We have shown that uncontrolled sending of endpoints may also produce memory leaks. Our finite-weight relaxes the send-state restriction, because endpoints in a send-state always have a null weight.

Acknowledgments. We are grateful to Lorenzo Bettini for discussions on the notion of memory leak, to Nobuko Yoshida for comments on an early version of this paper, and to the anonymous referees for the detailed and useful reviews.

References

1. Bettini, L., Coppo, M., D'Antoni, L., De Luca, M., Dezani-Ciancaglini, M., Yoshida, N.: Global Progress in Dynamically Interleaved Multiparty Sessions. In: van Breugel, F., Chechik, M. (eds.) CONCUR 2008. LNCS, vol. 5201, pp. 418–433. Springer, Heidelberg (2008)
2. Castagna, G., Dezani-Ciancaglini, M., Giachino, E., Padovani, L.: Foundations of session types. In: PPDP 2009, pp. 219–230. ACM, New York (2009)
3. Dezani-Ciancaglini, M., de Liguoro, U., Yoshida, N.: On Progress for Structured Communications. In: Barthe, G., Fournet, C. (eds.) TGC 2007 and FODO 2008. LNCS, vol. 4912, pp. 257–275. Springer, Heidelberg (2008)
4. Fähndrich, M., Aiken, M., Hawblitzel, C., Hodson, O., Hunt, G., Larus, J.R., Levi, S.: Language support for fast and reliable message-based communication in singularity os. In: EuroSys 2006, pp. 177–190. ACM, New York (2006)
5. Gay, S., Hole, M.: Subtyping for session types in the π-calculus. Acta Informatica 42(2-3), 191–225 (2005)
6. Gay, S., Vasconcelos, V.T.: Linear type theory for asynchronous session types. Journal of Functional Programming 20(01), 19–50 (2010)
7. Honda, K.: Types for dyadic interaction. In: Best, E. (ed.) CONCUR 1993. LNCS, vol. 715, pp. 509–523. Springer, Heidelberg (1993)
8. Honda, K., Vasconcelos, V.T., Kubo, M.: Language primitives and type disciplines for structured communication-based programming. In: Hankin, C. (ed.) ESOP 1998. LNCS, vol. 1381, pp. 122–138. Springer, Heidelberg (1998)
9. O'Hearn, P.W., Reynolds, J.C., Yang, H.: Local reasoning about programs that alter data structures. In: Fribourg, L. (ed.) CSL 2001 and EACSL 2001. LNCS, vol. 2142, pp. 1–19. Springer, Heidelberg (2001)
10. Padovani, L.: Session types at the mirror. EPTCS 12, 71–86 (2009)
11. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: LICS 2002, pp. 55–74. IEEE, Los Alamitos (2002)
12. Stengel, Z., Bultan, T.: Analyzing singularity channel contracts. In: ISSTA 2009, pp. 13–24. ACM, New York (2009)
13. Vasconcelos, V.T.: Fundamentals of session types. In: Bernardo, M., Padovani, L., Zavattaro, G. (eds.) SFM 2009. LNCS, vol. 5569, pp. 158–186. Springer, Heidelberg (2009)
14. Villard, J., Lozes, É., Calcagno, C.: Proving copyless message passing. In: Hu, Z. (ed.) APLAS 2009. LNCS, vol. 5904, pp. 194–209. Springer, Heidelberg (2009)
15. Villard, J., Lozes, É., Calcagno, C.: Tracking heaps that hop with heap-hop. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 275–279. Springer, Heidelberg (2010)
16. Walker, D., Gregory Morrisett, J.: Alias types for recursive data structures. In: Harper, R. (ed.) TIC 2000. LNCS, vol. 2071, pp. 177–206. Springer, Heidelberg (2001)