Polymorphic Types for Leak Detection in a Session-Oriented Functional Language

Viviana Bono, Luca Padovani, Andrea Tosatto

Polymorphic Types for Leak Detection in a Session-Oriented Functional Language

Viviana Bono, Luca Padovani, and Andrea Tosatto

Dipartimento di Informatica, Università di Torino, Italy

Abstract. Copyless message passing is a communication paradigm in which only pointers are exchanged between sender and receiver processes. Because of its nature, this paradigm requires that messages are treated as *linear* resources. Yet, even linear type systems leave room for scenarios where apparently well-typed programs may leak memory. In this work we develop a polymorphic type system for leak-free copyless messaging in a functional setting, where first-class functions can be used as messages.

1 Introduction

When communicating processes can access a shared address space, it is sensible to consider a *copyless* form of communication whereby only pointers to messages (instead of the messages themselves) are copied from senders to receivers. The Singularity Operating System [9,10] is a notable example of system making pervasive use of copyless communication. In Singularity, messages live in a shared area called *exchange heap* that, for practical reasons, cannot be garbage collected: data in this area must be explicitly allocated and deallocated. Messages travel through channels that are represented as pairs of *peer endpoints*: a message sent over one endpoint is received from the corresponding peer. Because channel endpoints can be sent as messages, they are also allocated in the exchange heap and explicitly managed.

Explicit memory management is a well-known source of hard-to-trace bugs. For this reason, it calls for the development of static analysis techniques meant to spot dangerous code. In [1,2] we have developed a type system for a language of processes that interact through copyless messaging: well-typed processes are guaranteed to be free from memory faults, memory leaks, and communication errors. The type system associates channel endpoints with *endpoint types* reminiscent of session types [7,8]. The present work extends the results of [2] to a language with first-class functions. For example,

$$g \stackrel{\text{def}}{=} \lambda c.\lambda x.\texttt{let}\ f, c' = \texttt{receive}\ c\ \texttt{in}\ \texttt{close}\ c'; (f\ x)$$

is a function that, when applied to a channel endpoint c and a value x, transforms x through a function received from c. The `receive` c application evaluates to a pair consisting of the message received from c and c itself, the `let` deconstructs such pair and binds its components to the local variables f and c', and

`close` c' deallocates c'. The explicit re-binding of c enables the type system to keep track of resource allocation and to spot violations in the memory management. Indeed, endpoints are linear resources that are *consumed* when used in a function application (like c and c' in `receive` c and `close` c') and *acquired* when obtained as result of a function application (like for f and c' returned by `receive`). In addition, the re-binding allows to assign different types to the same channel endpoint according to how the code uses it: the above function can be typed with the assignments $c : \mathtt{?}(\mathtt{Int} \multimap \mathtt{Int}).\mathtt{end}$ and $c' : \mathtt{end}$ where the type of c denotes the fact that it can be used for receiving a message of type $\mathtt{Int} \multimap \mathtt{Int}$ (a linear function from integers to integers) and the type `end` of c' is the residual of c's type after receiving this message; `end` indicates that c' can be deallocated.

In [2] it was observed that it is possible to write apparently correct code that yields *memory leaks*, whereby an allocated region of the heap becomes inaccessible. This phenomenon manifests itself in the program

$$\mathtt{let}\ a, b = \mathtt{open}\ \mathtt{unit}\ \mathtt{in}\ \mathtt{close}\ (\mathtt{send}\ b\ a)$$

which creates a new channel represented as the two peer endpoints a and b and sends b over its own peer a. This code fragment can be typed using the assignment $\{a : T_1, b : S_1\}$ where $T_1 = \mathtt{!}S_1.\mathtt{end}$ and S_1 is the recursive type satisfying the equation $S_1 = \mathtt{?}S_1.\mathtt{end}$. Note that in this code fragment every resource that is acquired is also consumed. Yet, after the execution of this code only endpoint a is actually deallocated, while endpoint b has become inaccessible because stored within its own queue. In [2] we rule out code like this by restricting the values that can be sent as messages depending on their type. The idea consists in looking at types for estimating the length of the chains of pointers originating from values with that type – we call such measure *type weight* – and then restricting messages to values whose type has a bounded weight. For example, the queue of an endpoint of type S_1 may contain a message of type S_1, therefore the weight of S_1 is unbounded, whereas the weight of T_1 is zero because a can only be used for sending messages, so its queue will never contain a message.

It turns out that the same technique does not work "out of the box" in a language with first-class functions. The problem is that arrow types only tell us what a function accepts and produces, but not which other (heap-allocated) values the function may use, while this information is essential for determining the weight of an arrow type. To illustrate the issue, consider the code fragment

$$\mathtt{let}\ a, b = \mathtt{open}\ \mathtt{unit}\ \mathtt{in}\ \mathtt{close}\ (\mathtt{send}\ (g\ b)\ a)$$

which is a little twist from the previous one. According to the definition of g, $(g\ b)$ is a message that *contains* b. This code fragment can be typed with the assignment $\{a : T_2, b : S_2\}$, where $T_2 = \mathtt{!}(\mathtt{Int} \multimap \mathtt{Int}).\mathtt{end}$ and $S_2 = \mathtt{?}(\mathtt{Int} \multimap \mathtt{Int}).\mathtt{end}$. As before, this closed code fragment yields a memory leak due to b not being deallocated, but in this case the type $\mathtt{Int} \multimap \mathtt{Int}$ of the message in S_2 does not provide much information: we only know that $(g\ b)$ is a function that may make use of a linear value, but the type of such linear value, which is key in order to assess the weight of $\mathtt{Int} \multimap \mathtt{Int}$, is unknown. The solution we put

forward consists in decorating linear arrow types with an explicit weight, as in $\texttt{Int}\, w \multimap \texttt{Int}$, to keep track of this information. In the above example, the weight of S_2 should be strictly greater than w, because endpoint b carries messages of type $\texttt{Int}\, w \multimap \texttt{Int}$. At the same time, w should not be smaller than the weight of S_2, because $(g\ b)$ contains b. From this train of thoughts, one infers that there is no finite bound for w, and consequently that $(g\ b)$ cannot be safely sent over a.

Polymorphism adds another dimension to the problem and forces us to consider a more structured representation of type weights. For example, the function

$$\mathit{forward} \stackrel{\text{def}}{=} \lambda x.\lambda y.\texttt{let}\ m, x' = \texttt{receive}\ x\ \texttt{in}\ (x', \texttt{send}\ m\ y)$$

which forwards a message from an endpoint x to another endpoint y, can be given the polymorphic type $\texttt{?}a.A \to \texttt{!}a.B\ w \multimap A \otimes B$. The issue is how to determine the weight w, given that x occurs free in the function $\lambda y \cdots$ and that it has the partially specified type $\texttt{?}a.A$. The actual weight of $\texttt{?}a.A$ *depends* on the weight of the types with which a and A are instantiated. In particular, it is the maximum between the weight of A and the weight of a plus 1 (because the queue for x may contain a value of type a). We keep track of this dependency by letting $w = \{a, A\} + 1$.

In the rest of the paper we formalize all the notions sketched so far. We begin by defining syntax and reduction semantics of a core functional language equipped with session-oriented communication primitives (Section 2). We also provide a precise definition of "correct" programs as those that are free from memory faults, memory leaks, and communication errors. We proceed by presenting the type language (Section 3), the type system and its soundness results (Section 4). Related work (Section 5) and a few concluding remarks (Section 6) end the main body of the paper. Proofs of the results can be found in the long version of the paper [3].

2 Language

The syntax of our language is described in Table 1, where we use the following syntactic categories: x, y range over an infinite set of *variables*; $\texttt{p}$, $\texttt{q}$ range over an infinite set $\mathsf{Pointers}$ of *pointers*; u ranges over *names*, which are either variables or pointers, and $\texttt{U}$, $\texttt{V}$ over sets of names; E ranges over *expressions* and v over *values*; we write $\tilde{v}$ to denote *queues*, namely finite sequences of values; $\texttt{k}$ ranges over *constants* from the set $\{\texttt{unit}, \texttt{fix}, \texttt{fork}, \texttt{open}, \texttt{close}, \texttt{send}, \texttt{receive}\}$; P, Q range over *processes*; μ ranges over *heaps*. The sub-language of expressions is almost standard, except for the $\texttt{let}$ construct which deconstructs pairs and binds their components to two variables. As usual, $\lambda x.E$ binds x in E and $\texttt{let}\ x, y = E_1\ \texttt{in}\ E_2$ binds x and y in E_2, therefore, bound and free names are defined in the usual way. We will sometimes write $\texttt{let}\ x = E_1\ \texttt{in}\ E_2$ in place of $\texttt{let}\ x, y = (E_1, \texttt{unit})\ \texttt{in}\ E_2$ where y is some fresh variable. Processes are parallel compositions of expressions, each expression representing a thread of execution. We identify processes modulo commutativity and associativity of

Table 1. Syntax of expressions, processes, values, and heaps.

$E ::=$		**Expression**	$v ::=$		**Value**
	v	(value)		$\mathtt{p}$	(pointer)
$\mid$	x	(variable)	$\mid$	$\mathtt{k}$	(constant)
$\mid$	(E, E)	(pair)	$\mid$	$\lambda x.E$	(abstraction)
$\mid$	EE	(application)	$\mid$	(v, v)	(pair)
$\mid$	$\mathtt{let}\ x, y = E\ \mathtt{in}\ E$	(pattern match)			
			$\mu ::=$		**Heap**
$P ::=$		**Process**		$\varnothing$	(empty heap)
	$\langle E \rangle$	(thread)	$\mid$	$\mathtt{p} \mapsto [\mathtt{q}; \tilde{v}]$	(endpoint)
$\mid$	$P \parallel P$	(composition)	$\mid$	μ, μ	(composition)

$\parallel$. We write $\mathsf{fn}(E)$ and $\mathsf{fn}(P)$ for denoting the set of free names occurring in E and P.

In order to express the operational semantics of processes, we need an explicit representation of *heaps* as finite maps from pointers to endpoint structures $[\mathtt{p}, \tilde{v}]$, which, in turn, are a pair containing a pointer and a queue of values, representing the messages received at that endpoint. The set of pointers to allocated endpoint structures is $\mathsf{dom}(\mu)$, and we assume that the composition μ_1, μ_2 is defined only when $\mathsf{dom}(\mu_1) \cap \mathsf{dom}(\mu_2) = \emptyset$. A *system* is a pair $\mu \fatsemi P$ of a heap μ and a process P.

Table 2 defines the reduction semantics of expressions and of systems. Expressions reduce according to a conventional call-by-value semantics extended with pattern matching over pairs. Systems reduce as a consequence of expressions that are evaluated in threads and of primitive functions forking new threads and implementing the communication operations. Rule (R-Thread) performs a step of computation within a thread. The *evaluation context* $\mathcal{E}$ [12,6] is an expression with a hole, denoted by $[\,]$, where computation in a thread happens next. Evaluation contexts are defined by

$$\mathcal{E} ::= [\,] \mid (\mathcal{E}, E) \mid (v, \mathcal{E}) \mid \mathcal{E}E \mid v\mathcal{E} \mid \mathtt{let}\ x, y = \mathcal{E}\ \mathtt{in}\ E \mid \mathtt{let}\ x, y = v\ \mathtt{in}\ \mathcal{E}$$

and $\mathcal{E}[E]$ denotes the result of filling the hole in $\mathcal{E}$ with the expression E.

Rule (R-Par) singles out threads running in parallel. Rule (R-Fork) spawns a new thread. Rule (R-Open) creates a channel as a pair of peer endpoints, by allocating two endpoint structures in the heap which point to each other and initially have an empty queue (ε denotes the empty sequence of values). Rule (R-Send) inserts a value v on the queue of the peer endpoint of $\mathtt{p}$. Rule (R-Receive) extracts the head value from the queue associated with the endpoint pointed to by $\mathtt{p}$. In both `send` and `receive`, the operation evaluates to the endpoint being used for communication, which is thus available for further operations. In the following we write $\Longrightarrow$ for the reflexive, transitive closure of $\longrightarrow$ and we write $\mu \fatsemi P \not\longrightarrow$ if there exist no μ' and P' such that $\mu \fatsemi P \longrightarrow \mu' \fatsemi P'$.

In this work, as in [2], 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

Table 2. Reduction semantics of expressions and systems.

Reduction of expressions

$$(\lambda x.E)v \longrightarrow_{\mathsf{v}} E\{v/x\} \qquad \texttt{fix}(\lambda x.E) \longrightarrow_{\mathsf{v}} E\{\texttt{fix}(\lambda x.E)/x\}$$

$$\texttt{let}\ x, y = (v, w)\ \texttt{in}\ E \longrightarrow_{\mathsf{v}} E\{v, w/x, y\}$$

Reduction of systems

$$\text{(R-Thread)}\ \frac{E \longrightarrow_{\mathsf{v}} E'}{\mu \fatsemi \langle \mathcal{E}[E] \rangle \longrightarrow \mu \fatsemi \langle \mathcal{E}[E'] \rangle} \qquad \text{(R-Fork)}\ \mu \fatsemi \langle \mathcal{E}[\texttt{fork}\ E] \rangle \longrightarrow \mu \fatsemi \langle \mathcal{E}[\texttt{unit}] \rangle \parallel \langle E \rangle$$

$$\text{(R-Par)}\ \frac{\mu \fatsemi P_1 \longrightarrow \mu' \fatsemi P_1'}{\mu \fatsemi P_1 \parallel P_2 \longrightarrow \mu' \fatsemi P_1' \parallel P_2} \qquad \text{(R-Open)}\ \mu \fatsemi \langle \mathcal{E}[\texttt{open unit}] \rangle \longrightarrow \mu, \mathtt{p} \mapsto [\mathtt{q}; \varepsilon], \mathtt{q} \mapsto [\mathtt{p}; \varepsilon] \fatsemi \langle \mathcal{E}[(\mathtt{p}, \mathtt{q})] \rangle$$

$$\text{(R-Send)}\ \mu, \mathtt{p} \mapsto [\mathtt{q}; \mathcal{Q}], \mathtt{q} \mapsto [\mathtt{p}; \mathcal{Q}'] \fatsemi \langle \mathcal{E}[\texttt{send}\ (v, \mathtt{p})] \rangle \longrightarrow \mu, \mathtt{p} \mapsto [\mathtt{q}; \mathcal{Q}], \mathtt{q} \mapsto [\mathtt{p}; \mathcal{Q}' v] \fatsemi \langle \mathcal{E}[\mathtt{p}] \rangle$$

$$\text{(R-Receive)}\ \mu, \mathtt{p} \mapsto [\mathtt{q}; v\mathcal{Q}] \fatsemi \langle \mathcal{E}[\texttt{receive}\ \mathtt{p}] \rangle \longrightarrow \mu, \mathtt{p} \mapsto [\mathtt{q}; \mathcal{Q}] \fatsemi \langle \mathcal{E}[(v, \mathtt{p})] \rangle$$

corresponding to an allocated endpoint; we wish every system to be *leak free*, where a leak is an endpoint 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 avoid *communication errors*, by enjoying (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. In order to do so, 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 $\mathsf{fn}(P)$ (we can think of the pointers in $\mathsf{fn}(P)$ as of the local variables of the process stored in its stack); from these pointers, the process may reach transitively other heap objects by reading messages from the queue of the endpoints it can reach.

Definition 2.1 (reachable pointers). *We say that* $\mathtt{p}$ *is reachable from* $\mathtt{q}$ *in* μ *(written* $\mathtt{p} \prec_\mu \mathtt{q}$*) if* $\mathtt{q} \mapsto [\mathtt{r}; \tilde{w} v \tilde{w}'] \in \mu$ *and* $\mathtt{p} \in \mathsf{fn}(v)$*. We write* $\preceq_\mu$ *for the reflexive and transitive closure of* $\prec_\mu$*. The pointers reachable from* $\mathtt{U}$ *in* μ *are defined as* $\mu\text{-}\mathsf{reach}(\mathtt{U}) = \{\mathtt{p} \in \mathsf{Pointers} \mid \exists \mathtt{q} \in \mathtt{U} : \mathtt{p} \preceq_\mu \mathtt{q}\}$*.*

We are now ready to define formally well-behaved processes.

Definition 2.2 (well-behaved process). *We say that* P *is* well behaved *if for every possible reduction* $\varnothing \fatsemi P \Rightarrow \mu \fatsemi Q$ *the following properties hold:*

1. $\mathsf{dom}(\mu) = \mu\text{-}\mathsf{reach}(\mathsf{fn}(Q))$.

Table 3. Syntax of types.

$\sigma ::=$		**Type Scheme**	$T ::=$		**Endpoint Type**
	t	(monomorphic type)		$\mathtt{end}$	(termination)
	$\forall\alpha :: \rho.\sigma$	(polymorphic type)		A	(variable)
				$\overline{A}$	(dualized variable)
$t ::=$		**Type**		$?t.T$	(input)
	$\mathtt{Unit}$	(unit type)		$!t.T$	(output)
	a	(variable)		$\mathtt{rec}\ A.T$	(recursion)
	T	(endpoint type)			
	$t \otimes t$	(linear pair)	$w ::=$		**Weight**
	$t \to t$	(function)		∞	(unbounded weight)
	$t\ w{\multimap}\ t$	(linear function)		$X + n$	(bounded weight)

2. *if* $Q = P_1 \parallel P_2$, *then* $\mu\text{-}\mathsf{reach}(\mathsf{fn}(P_1)) \cap \mu\text{-}\mathsf{reach}(\mathsf{fn}(P_2)) = \emptyset$.
3. *if* $Q = \langle E\rangle \parallel Q'$ *and* $\mu \fatsemi \langle E\rangle \not\rightarrow$, *then either* $E = \mathtt{unit}$, *or* $E = \mathcal{E}[\mathtt{receive}\ \mathsf{p}]$ *and* $\mathsf{q} \mapsto [\mathsf{p}; \varepsilon] \in \mu$, *or* $E = \mathcal{E}[\mathtt{close}\ \mathsf{p}]$ *and* $\mathsf{p} \mapsto [\mathsf{q}; \varepsilon] \in \mu$.

Conditions (1) and (2) ask for the absence of faults and leaks. In detail, condition (1) states that every allocated pointer in the heap is reachable by one process, and that every reachable pointer corresponds to an object allocated in the heap. Condition (2) states that processes are isolated, namely that no pointer is reachable from two or more processes. Since expressions of the form $\mathtt{close}\ \mathsf{p}$ are persistent (they do not reduce), this condition rules out memory faults whereby the same endpoint is deallocated multiple times. Condition (3) requires the absence of communication errors, namely that if $\mu \fatsemi Q$ 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. Condition (3) also ensures the absence of so-called *orphan messages*: no message accumulates in the queue of closed endpoints.

3 Types

Table 3 gives the syntax of types using the following syntactic categories: m, n range over natural numbers; A, B, ... range over an infinite set of *endpoint type variables*; a, b, ... range over an infinite set of *value type variables*; α, β range over *type variables*, which are either endpoint or value type variables without distinction; X, Y range over finite sets of type variables; ρ ranges over *qualifiers*, which are elements of $\{\mathtt{any}, \mathtt{fin}\}$; w ranges over *weights*; t, s range over *types*; σ range over *type schemes*; T, S range over *endpoint types*.

Endpoint types denote pointers to channel endpoints; they are fairly standard session types with input/output prefixes $?t/!t$, recursion, and a terminal state $\mathtt{end}$. Endpoint type variables A can occur in *dualized form* $\overline{A}$, as in [4]. This is

necessary for typing some functions, beside simplifying the definition of *duality*. For simplicity we omit choices and branches; they can be added without posing substantial problems. *Types* include the conventional constructs of functional languages *à la ML*, comprising a `Unit` type (other data types can be added as needed), linear functions, and linear pairs. The linear types are necessary to denote objects (functions, pairs) that contain channel endpoints and that, for this reason, must be owned and used linearly. In particular, the linear arrow type $t\ w \multimap s$ denotes a function whose body may contain pointers and has an explicit decoration w determining its weight. A *weight* is a term representing the length of a chain of pointers in the program heap. It can be either ∞, denoting an unbound length, or $X + n$ denoting a length that is bound by the weight of the types that will instantiate the type variables in X plus the value of the constant n. We will often write X instead of $X + 0$ and n instead of $\emptyset + n$. *Type schemes* are almost standard, except that polymorphic type variables are associated with a *qualifier* ρ: if the qualifier is `any`, then there is no constrain as to which types may instantiate the type variable; if the qualifier is `fin`, then only finite-weight types may instantiate the type variable. We will write $\tilde{t}$ for denoting sequences $t_1, \ldots, t_n$ of types and we will often write $\forall\tilde{\alpha} :: \tilde{\rho}.t$ in place of $\forall\alpha_1 :: \rho_1 \cdots \forall\alpha_n :: \rho_n.t$ for some n.

A type is *well formed* if none of its endpoint type variables bound by a `rec` occurs in a weight. For example, both $\texttt{Unit}\{A\} \multimap \texttt{Unit}$ and $\forall A :: \texttt{any}.\texttt{Unit}\{A\} \multimap \texttt{Unit}$ are well formed, but $\texttt{rec}\ A.!(\texttt{Unit}\ \{A\} \multimap \texttt{Unit}).\texttt{end}$ is not. From now on we implicitly assume to work with well-formed types.

The predicate $\mathsf{lin}(\sigma)$ identifies *linear types*:

$$\mathsf{lin}(\alpha) \qquad \mathsf{lin}(T) \qquad \mathsf{lin}(t_1 \otimes t_2) \qquad \mathsf{lin}(t_1\ w \multimap t_2) \qquad \frac{\mathsf{lin}(t)}{\mathsf{lin}(\forall\tilde{\alpha} :: \tilde{\rho}.t)}$$

We say that σ is *unlimited*, notation $\mathsf{un}(\sigma)$, if not $\mathsf{lin}(\sigma)$. Note that a type variable is always considered linear because it *may* be instantiated by a linear type. A full-fledged type system might distinguish between linear and unlimited type variables for better precision; we leave this as a straightforward extension for the sake of simplicity.

There are three crucial notions regarding types that we need to define next, namely *duality*, *type weight*, *substitution*. It turns out that these notions are mutually dependent on one another and their formal definition requires a carefully ordered sequence of intermediate steps that relies on type well formedness. Here we only present the "final" definitions and highlight peculiarities and pitfalls of each, while the detailed development can be found in [3].

Duality. Communication errors are prevented by associating peer endpoints with dual endpoint types, so that when one endpoint type allows sending a message of type t, the dual endpoint type allows receiving messages of type t and when one endpoint should be closed the other endpoint should be closed as well. Roughly, the dual of an endpoint type T, denoted by $\overline{T}$, is obtained from T by swapping `?`'s with `!`'s so that, for example, the dual of $?t.!s.\texttt{end}$ is $!t.?s.\texttt{end}$. In practice,

things are a little more complicated because of recursive behaviors. For example, the dual of $T = \mathtt{rec}\ A.!A.\mathtt{end}$ is *not* $S = \mathtt{rec}\ A.?A.\mathtt{end}$. Indeed, in T the recursion variable occurs within a prefix, denoting the fact that an endpoint of type T carries messages which have themselves type T. That is, $T = !T.\mathtt{end}$. By contrast, we have $S = ?S.\mathtt{end}$, hence from an endpoint of type S we can receive another endpoint having type S. In fact, we have $\overline{T} = ?T.\mathtt{end} \neq S$.

The *dual* of an endpoint type is inductively defined by the equations:

$$\overline{\mathtt{end}} = \mathtt{end} \qquad \overline{\mathtt{rec}\ A.T} = \mathtt{rec}\ A.\overline{T\{\overline{A}/A\}} \qquad \overline{A} = \overline{A} \qquad \overline{\overline{A}} = A \qquad \overline{?t.T} = !t.\overline{T} \qquad \overline{!t.T} = ?t.\overline{T}$$

where $T\{\overline{A}/A\}$ denotes the endpoint type T where free occurrences of A have been replaced by its dualized form and free occurrences of $\overline{A}$ by A. For example, we have $\overline{\mathtt{rec}\ A.!A.\mathtt{end}} = \mathtt{rec}\ A.!\overline{A}.\mathtt{end} = \mathtt{rec}\ A.?\overline{A}.\mathtt{end}$.

Weight. The weight of a type (scheme) gives information about the length of the chains of pointers originating from values having that type (scheme). For example, the weight of `end` is 0, because the queue of an endpoint of type `end` will never contain any message, hence no chains of pointers can originate from an endpoint of this type. On the contrary, an endpoint of type `?end.end` *may* contain a pointer to another endpoint of type `end`, therefore its weight is 1. Because types may contain type variables, in general the weight of a type depends on how these type variables are instantiated. In order to compute the weight of a type, we must be able to compare weights:

Definition 3.1 (weight order). *We define the relation $\leq$ over weights as the least partial order such that $w \leq \infty$ and $X + m \leq Y + n$ if $X \subseteq Y$ and $m \leq n$.*

Observe that, if $\mathcal{W}$ is the set of all weights, then $(\mathcal{W}, \leq)$ is a complete lattice with least element $\emptyset + 0$ and greatest element ∞. In what follows we will use the operators $\vee$ and $\wedge$ to respectively compute the join and meet of possibly infinite sets of weights.

Definition 3.2 (weight). *Let $\downarrow$ be the largest relation such that $t \downarrow w$ implies either*

- *$w = \infty$, or*
- *$t = \mathtt{Unit}$ or $t = t_1 \to t_2$ or $t = \mathtt{end}$ or $t = !s.T$, or*
- *$t = \alpha$ and $w = (X \cup \{\alpha\}) + n$, or*
- *$t = t_1 \otimes t_2$ and $t_1 \downarrow w$ and $t_2 \downarrow w$, or*
- *$t = ?s.T$ and $w = X + (n + 1)$ and $s \downarrow (X + n)$ and $T \downarrow w$, or*
- *$t = t_1\ w' \multimap t_2$ and $w' \leq w$.*

The weight *of a type t, denoted $\|t\|$, is defined as $\|t\| \stackrel{\text{def}}{=} \bigwedge_{t \downarrow w} w$.*

Intuitively, the relation $t \downarrow w$ says that w is an upper bound for the length of the chains of pointers originating from values of type t, and $\|t\|$ is the least of such upperbounds. It is easy to see that every unlimited type has a null weight

(a value with unlimited type cannot contain any pointer) and that, for instance, $\|\alpha\| = \{\alpha\}$ and $\|t \otimes s\| = \|t\| \vee \|s\|$. Also, endpoints with type $\mathtt{end}$ or $\mathtt{!}t.T$ have null weight because their queues must be empty (this property will be enforced by the type system in Section 4). However, we have that $\|\mathtt{?}a.\mathtt{end}\| = \{a\} + 1$ because an endpoint of such type may contain a value of type a, so the length of the longest chain of pointers originating from such an endpoint is 1 plus the length of longest chain of pointers originating from a value with type that instantiates a. In general, we have $\|\mathtt{?}t.T\| = (\|t\| + 1) \vee \|T\|$. If we take the endpoint type $S_1 = \mathtt{rec}\ A.\mathtt{?}A.\mathtt{end}$ from Section 1 we have $\|S_1\| = \infty$ because S_1 has no finite upperbound. Finally, note that $\|\overline{A}\| = \infty$. This is because, in general, there is no relationship between the weight of an endpoint type and that of its dual. For instance, we have $\|\mathtt{!}S_1.\mathtt{end}\| = 0$ but $\|\overline{\mathtt{!}S_1.\mathtt{end}}\| = \|\mathtt{?}S_1.\mathtt{end}\| = \infty$. It would be possible to allow dualized type variables in the syntax of weights, but since such variables occur seldom in types we leave this extension out of our formal treatment and conservatively approximate their weight to ∞.

Substitution. Intuitively, a substitution $t\{s/\alpha\}$ represents the type obtained by replacing the occurrences of α in t with s. This notion is standard, except for two features that are specific of our type language. The first feature is the presence of dualized endpoint type variables $\overline{A}$. The idea is that, when A is replaced by an endpoint type T, $\overline{A}$ is replaced by $\overline{T}$, namely by the dual endpoint type of T that we have just introduced. The second feature is the presence of type variables in weights which decorate linear function types. In particular, a substitution $(t_1\ w \multimap t_2)\{s/\alpha\}$ may need to update $w = X + n$ if $\alpha \in X$. Formally, we define a weight substitution operation $w\{w'/\alpha\}$ such that

$$w\{w'/\alpha\} \stackrel{\text{def}}{=} \begin{cases} ((X \setminus \{\alpha\}) \vee w') + n & \text{if } w = X + n \text{ and } \alpha \in X \\ w & \text{otherwise} \end{cases}$$

where we define a meta operator $w + n$ such that $\infty + n = \infty$ and $(X + m) + n = X + (m + n)$. Then $t\{s/\alpha\}$ is defined in the standard way except that

$$\overline{A}\{T/A\} = \overline{T} \qquad \text{and} \qquad (t_1\ w \multimap t_2)\{s/\alpha\} = t_1\{s/\alpha\}\ w\{\|s\|/\alpha\} \multimap t_2\{s/\alpha\}$$

Finally, we generalize the notion of weight to type schemes so that $\|\forall\tilde{\alpha} :: \tilde{\rho}.t\| \stackrel{\text{def}}{=} \bigvee \|t\{\tilde{s}/\tilde{\alpha}\}\|$. Note that we do not worry about instantiating $\mathtt{fin}$-qualified type variables with infinite-weight types, since such type variables can be instantiated with types having arbitrarily large weight anyway. Therefore, if the weight of t depends in any way from one of the α_i, the overall weight of the type scheme will be ∞, no matter what.

We identify types modulo folding/unfolding of recursions. That is, $\mathtt{rec}\ A.T = T\{\mathtt{rec}\ A.T/A\}$ (we have already used this property in Definition 3.2).

4 Type System

We give the types of the constants in Table 4. The types in the l.h.s. of the table are unremarkable. The $\mathtt{open}$ primitive returns a pair of peer channel endpoints

Table 4. Type of constants.

$\texttt{unit} : \texttt{Unit}$	$\texttt{open} : \forall A :: \texttt{any}.\texttt{Unit} \to (A \otimes \overline{A})$
$\texttt{fix} : \forall a :: \texttt{any}.(a \to a) \to a$	$\texttt{close} : \texttt{end} \to \texttt{Unit}$
$\texttt{fork} : \texttt{Unit} \to \texttt{Unit}$	$\texttt{send} : \forall a :: \texttt{fin}.\forall A :: \texttt{any}.(a \otimes \texttt{!}a.A) \to A$
	$\texttt{receive} : \forall a :: \texttt{fin}.\forall A :: \texttt{any}.\texttt{?}a.A \to (a \otimes A)$

when applied to the `unit` value. For this reason, the resulting type is a pair of dual endpoint types. Because `open` is polymorphic, this can only be expressed using a dualized endpoint type variable. Note how `open` is an example of resource-producing function, accepting an unlimited value `unit` and returning a linear pair. The `close` primitive accepts an endpoint provided that it has type `end` and deallocates it. Being the converse of `open`, `close` is an example of resource-consuming function, accepting a linear value and not returning it. The `send` and `receive` constants implement the communication primitives: `send` accepts a message of type a, an endpoint of type $\texttt{!}a.A$ that allows sending such a message, and returns the same endpoint with the residual type A; `receive` accepts an endpoint of type $\texttt{?}a.A$, reads a message of type a from such an endpoint, and returns the pair consisting of the received message and the endpoint with the residual type A. Observe that, in both `send` and `receive`, the value type variable a is qualified by `fin`, meaning that only values with finite-weight type can be sent and received. On the contrary, no constraint is imposed on A. In the following we write $\mathsf{TypeOf}(\texttt{k})$ for the type scheme associated with `k` according to Table 4.

Judgments of the type system depend on two finite maps: the *type variable environment* $\Sigma = \{\alpha_i :: \rho_i\}_{i \in I}$ associates type variables with qualifiers, while the *name environment* $\Gamma = \{u_i : \sigma_i\}_{i \in I}$ associates names with type schemes. In both cases we use $\mathsf{dom}(\cdot)$ for denoting the set of type variables/names for which there is an association in the environment. We also write $\Sigma, \alpha :: \rho$ (respectively, $\Gamma, u : \sigma$) to extend the environment whenever $\alpha \notin \mathsf{dom}(\Sigma)$ (respectively, $u \notin \mathsf{dom}(\Gamma)$). Finally, we write $\Gamma|_{\mathtt{U}}$ for the restriction of Γ to the names in $\mathtt{U}$. Because name environments may contain linear entities (pointers) as well as unlimited ones, it is convenient to define also a more flexible (partial) operator $+$ for extending them. As in [5], we let

$$\Gamma + u : \sigma = \begin{cases} \Gamma & \text{if } u : \sigma \in \Gamma \text{ and } \mathsf{un}(\sigma) \\ \Gamma, u : \sigma & \text{if } u \notin \mathsf{dom}(\Gamma) \\ \text{undefined} & \text{otherwise} \end{cases}$$

and we extend $+$ to pairs of environments $\Gamma_1 + \Gamma_2$ by induction on Γ_2 in the natural way. We write $\mathsf{lin}(\Gamma)$ if $\mathsf{lin}(\Gamma(u))$ for some $u \in \mathsf{dom}(\Gamma)$ and $\mathsf{un}(\Gamma)$ otherwise.

Sometimes we will need to reason on the finiteness of a weight which contains type variables. In such cases, we use the information contained in a type variable environment for determining whether a weight is finite or not. More precisely, we write $\Sigma \vdash X + n < \infty$ whenever $\alpha :: \texttt{fin} \in \Sigma$ for every $\alpha \in X$.

Table 5. Typing rules for processes and expressions.

$$\frac{\emptyset;\Gamma \vdash E : \texttt{Unit}}{\Gamma \vdash \langle E \rangle}\ (\text{T-Thread}) \qquad \frac{\Gamma_1 \vdash P_1 \qquad \Gamma_2 \vdash P_2}{\Gamma_1 + \Gamma_2 \vdash P_1 \parallel P_2}\ (\text{T-Par}) \qquad \frac{\mathsf{un}(\Gamma) \qquad \Sigma \vdash \mathsf{TypeOf}(\texttt{k}) \succ t}{\Sigma;\Gamma \vdash \texttt{k} : t}\ (\text{T-Const})$$

$$\frac{\mathsf{un}(\Gamma) \qquad \Sigma \vdash \sigma \succ t}{\Sigma;\Gamma, u : \sigma \vdash u : t}\ (\text{T-Name}) \qquad \frac{\Sigma, \tilde{\alpha} :: \tilde{\rho};\Gamma_1 \vdash E_1 : t_1 \qquad \Sigma;\Gamma_2, x : \forall \tilde{\alpha} :: \tilde{\rho}.t_1 \vdash E_2 : t_2}{\Sigma;\Gamma_1 + \Gamma_2 \vdash \texttt{let}\ x = E_1\ \texttt{in}\ E_2 : t_2}\ (\text{T-Let 1})$$

$$\frac{\forall i \in \{1,2\} : \Sigma;\Gamma_i \vdash E_i : t_i}{\Sigma;\Gamma_1 + \Gamma_2 \vdash (E_1, E_2) : t_1 \otimes t_2}\ (\text{T-Pair}) \qquad \frac{\Sigma;\Gamma_1 \vdash E_1 : t_1 \otimes t_2 \qquad \Sigma;\Gamma_2, x : t_1, y : t_2 \vdash E_2 : t}{\Sigma;\Gamma_1 + \Gamma_2 \vdash \texttt{let}\ x, y = E_1\ \texttt{in}\ E_2 : t}\ (\text{T-Let 2})$$

$$\frac{\Sigma;\Gamma, x : t \vdash E : s \qquad \mathsf{un}(\Gamma)}{\Sigma;\Gamma \vdash \lambda x.E : t \to s}\ (\text{T-Arrow}) \qquad \frac{\Sigma;\Gamma, x : t \vdash E : s \qquad \bigvee_{u \in \mathsf{dom}(\Gamma)} \|\Gamma(u)\| \leq w}{\Sigma;\Gamma \vdash \lambda x.E : t\ w\!\multimap s}\ (\text{T-Arrow Lin})$$

$$\frac{\Sigma;\Gamma_1 \vdash E_1 : t \to s \qquad \Sigma;\Gamma_2 \vdash E_2 : t}{\Sigma;\Gamma_1 + \Gamma_2 \vdash E_1 E_2 : s}\ (\text{T-App}) \qquad \frac{\Sigma;\Gamma_1 \vdash E_1 : t\ w\!\multimap s \qquad \Sigma;\Gamma_2 \vdash E_2 : t}{\Sigma;\Gamma_1 + \Gamma_2 \vdash E_1 E_2 : s}\ (\text{T-App Lin})$$

A type scheme $\forall \tilde{\alpha} :: \tilde{\rho}.t$ denotes the family of types obtained from t by instantiating each type variable α_i with a type whose weight respects the qualifier ρ_i. This is formally expressed by an *instantiation* relation $\Sigma \vdash \sigma \succ t$ defined by the rule

$$\frac{\rho_i = \texttt{fin} \Rightarrow \Sigma \vdash \|s_i\| < \infty^{\ (i=1..n)}}{\Sigma \vdash \forall \tilde{\alpha} :: \tilde{\rho}.t \succ t\{\tilde{s}/\tilde{\alpha}\}}$$

For example, if we consider once again the endpoint types $T_1 = \texttt{rec}\ A.!S_1.\texttt{end}$ and $S_1 = \texttt{rec}\ A.?A.\texttt{end}$ from Section 1 we have $\vdash \mathsf{TypeOf}(\texttt{send}) \succ (T_1, !T_1.\texttt{end}) \to \texttt{end}$ because T_1 has finite weight so it can instantiate the type variable a in $\mathsf{TypeOf}(\texttt{send})$. On the contrary, $\vdash \mathsf{TypeOf}(\texttt{send}) \not\succ (S_1, !S_1.\texttt{end}) \to \texttt{end}$ because $\|S_1\| = \infty$. Therefore, it is forbidden to send endpoints of type S_1.

The typing rules make use of two judgments, $\Gamma \vdash P$ stating that the process P is well typed in the name environment Γ, and $\Sigma;\Gamma \vdash E : t$ stating that E is well typed and has type t in the type variable environment Σ and name environment Γ. A judgment $\Gamma \vdash P$ is well formed if $\mathsf{dom}(\Gamma) \subseteq \mathsf{Pointers}$ and $\Gamma(\mathsf{p})$ is a closed type for every $\mathsf{p} \in \mathsf{dom}(\Gamma)$ and a judgment $\Sigma;\Gamma \vdash E$ is well formed if all type variables occurring free in Γ are in $\mathsf{dom}(\Sigma)$. Table 5 defines the typing rules for processes and expressions. Rule (T-Thread) and (T-Par) say that a process is well typed if so is each thread in it. Note that linear names are distributed linearly among threads by definition of $\Gamma_1 + \Gamma_2$. Rule (T-Const) instantiates the type of a constant, while rule (T-Name) retrieves and possibly instantiates the type of a name from the name environment. In both rules the unused part of the name environment must not contain linear resources. Rule (T-Let 1) is a linearity-aware

version of the rule to have `let`-polymorphism *à la ML*. The name environment is split between E_1 and E_2 knowing that, if $\mathsf{lin}(t_1)$, then x *must* occur in E_2. Note that, by well formedness of $\Sigma, \tilde{\alpha} :: \tilde{\rho}$, none of the type variables in $\tilde{\alpha}$ can be in $\mathsf{dom}(\Sigma)$ and hence can occur free in Γ_2. Therefore, they can be safely generalized when typing E_2. Overall, this treatment of universal polymorphism is borrowed from [12]: generalization and instantiation are embedded, respectively, in rule (T-Let 1), and in rules (T-Const) and (T-Name). Rules (T-Pair) and (T-Let 2) are, respectively, the construction and the de-construction (via pattern matching) of linear pairs. Rules (T-Arrow) and (T-App) are the standard ways of introducing and eliminating (unlimited) arrow types. The rule for arrow introduction requires the side condition $\mathsf{un}(\Gamma)$, meaning that the body of the function does not make use of any pointer. Finally, rules (T-Arrow Lin) and (T-App Lin) introduce and eliminate linear arrow types. In (T-Arrow Lin), the weight w that annotates the linear arrow is chosen in such a way that it is an upper bound for the weights of the types of all names occurring in E.

Example 4.1. The following derivation, where we omit $\Sigma = a :: \mathtt{fin}, A :: \mathtt{any}, B :: \mathtt{any}$ and we let $w = \|?a.A\| = \{a, A\}+1$, shows that the function *forward* defined at the end of Section 1 is well typed.

$$\dfrac{\dfrac{\dfrac{\dfrac{x : ?a.A \vdash \mathtt{receive}\ x : a \otimes A \qquad \dfrac{x' : A \vdash x' : A \qquad y : !a.B, m : a \vdash \mathtt{send}\ m\ y : B}{y : !a.B, m : a, x' : A \vdash (x', \mathtt{send}\ m\ y) : A \otimes B}}{x : ?a.A, y : !a.B \vdash \mathtt{let}\ m, x' = \mathtt{receive}\ x\ \mathtt{in}\ (x', \mathtt{send}\ m\ y) : A \otimes B}}{x : ?a.A \vdash \lambda y.\mathtt{let}\ m, x' = \mathtt{receive}\ x\ \mathtt{in}\ (x', \mathtt{send}\ m\ y) : !a.B\ w\!\multimap A \otimes B}}{\vdash \lambda x.\lambda y.\mathtt{let}\ m, x' = \mathtt{receive}\ x\ \mathtt{in}\ (x', \mathtt{send}\ m\ y) : ?a.A \to !a.B\ w\!\multimap A \otimes B}$$

Note that w is the smallest weight allowable in this derivation. Therefore, the obtained type is also the most precise and general one for *forward*. ■

Example 4.2. In a functional language, multi-argument functions are commonly represented in *curried form*, whereby such functions accept their arguments one at a time. On the contrary, the `send` constant is *un*curried, because it accepts both its arguments at once in a pair. The *curry* combinator transforms an uncurried binary function into its curried form and is defined as $\mathit{curry} = \lambda f.\lambda x.\lambda y.f\ (x, y)$. Below is the derivation showing that *curry* is well typed, where we let $\Sigma = a :: \mathtt{any}, b :: \mathtt{any}, c :: \mathtt{any}$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Sigma; f : (a \otimes b) \to c \vdash f : (a \otimes b) \to c \qquad \dfrac{\Sigma; x : a \vdash x : a \qquad \Sigma; y : b \vdash y : b}{\Sigma; x : a, y : b \vdash (x, y) : a \otimes b}}{\Sigma; f : (a \otimes b) \to c, x : a, y : b \vdash f\ (x, y) : c}}{\Sigma; f : (a \otimes b) \to c, x : a \vdash \lambda y.f\ (x, y) : b\ \{a\}\!\multimap c}}{\Sigma; f : (a \otimes b) \to c \vdash \lambda x.\lambda y.f\ (x, y) : a \to b\ \{a\}\!\multimap c}}{\Sigma; \emptyset \vdash \lambda f.\lambda x.\lambda y.f\ (x, y) : ((a \otimes b) \to c) \to a \to b\ \{a\}\!\multimap c}$$

Observe that the function returned by *curry* has type $a \to b\{a\}\multimap c$, where the linear arrow type has been decorated with the weight $\{a\}$. Indeed, the function $\lambda y.f\ (x, y)$ with this type has two free variables, f having an unlimited type with null weight, and x having type a. We can now obtain the curried form of `send` as *curry* `send` which can be given the polymorphic type $\forall a :: \mathtt{fin}.\forall A :: \mathtt{any}.a \to\ !a.A\ \{a\}\multimap A$. ■

Example 4.3. The *curry* function in Example 4.2 can only be applied to functions with unlimited type. It makes sense to consider also a linear variant *lcurry* of *curry* which has the same implementation of *curry* but can be used for currying linear functions (observe that the definition of *curry* uses its first argument f exactly once). Using a derivation very similar to that shown in Example 4.2, *lcurry* could be given the type

$$\forall a :: \mathtt{any}.\forall b :: \mathtt{any}.\forall c :: \mathtt{any}.((a \otimes b)\ w\multimap c) \to a\ w\multimap b\ (w \vee \{a\})\multimap c$$

except that this type depends on the weight w of the linear function being curried. This means that, in principle, we actually need a whole family $lcurry_w$ of combinators, one for each possible weight of the linear function to be curried. However, by combining polymorphism and explicit weight annotations in linear arrow types, we can provide *lcurry* with the most general type. The idea is to introduce another type variable, say d, which does not correspond to any actual argument of the function, but which represents an arbitrary weight, and to let $w = \{d\}$. This way we can give *lcurry* the type

$$\forall a :: \mathtt{any}.\forall b :: \mathtt{any}.\forall c :: \mathtt{any}.\forall d :: \mathtt{any}.((a \otimes b)\ \{d\}\multimap c) \to a\ \{d\}\multimap b\ \{a, d\}\multimap c$$

where we can instantiate d with a type having exactly the weight of the linear function to be curried. For example, suppose we wish to apply *lcurry* to some function $f : (a \otimes b)\ n\multimap c$. Then it is enough to instantiate the d variable in the type of *lcurry* with the type $\mathtt{T}^{[n]}$ defined by

$$\mathtt{T}^{[0]} = \mathtt{end} \qquad \mathtt{T}^{[m+1]} = ?\mathtt{T}^{[m]}.\mathtt{end}$$

and obtain *lcurry* $f : a\ n\multimap b\ (\{a\} + n)\multimap c$ as expected. ■

Properties. In order to show that every well-typed process is well behaved (Definition 2.2) we need, as usual, a subject reduction result showing that well-typedness is preserved under reductions. Since in our language processes allocate and modify the heap, we need to define a concept of *well-typed heap* just as we have defined a concept of well-typed process. Intuitively, a heap μ is well typed with respect to an environment Γ if the endpoints allocated in μ are consistent with their type in Γ. In particular, we want that whenever a message is inserted into the queue of an endpoint, the type of the message is consistent with the type of endpoint. To this aim, we define a function $\mathsf{tail}(T, \tilde{t})$ that, given an endpoint type T and a sequence of types $\tilde{t}$ of messages, computes the residual of T after all the messages have been received:

$$\mathsf{tail}(T, \varepsilon) = T \qquad \frac{\mathsf{tail}(T, \tilde{s}) = S}{\mathsf{tail}(?t.T, t\tilde{s}) = S}$$

Note that $\mathsf{tail}(T, \tilde{s})$ is undefined if $\tilde{s}$ is not empty and T does not begin with input actions: only endpoints whose type begins with input actions can have messages in their queue. The weight of `end` and output endpoint types is zero because of this property (Definition 3.2).

The notion of well-typed heap is relative to a pair $\Gamma_0; \Gamma$ of disjoint name environments: the overall environment Γ_0, Γ determines the type of *all* the objects allocated in the heap; the sub-environment Γ distinguishes the *roots* of the heap (the pointers that are not reachable from any other pointer) from the sub-environment Γ_0 of the pointers that are stored within other structures in the heap and that are reachable from some root.

Definition 4.1 (well-typed heap). *Let* $\mathsf{dom}(\Gamma) \cap \mathsf{dom}(\Gamma_0) = \emptyset$. *We say that* μ *is* well typed *in* $\Gamma_0; \Gamma$, *written* $\Gamma_0; \Gamma \Vdash \mu$, *if all of the following properties hold:*

1. *For every* $\mathtt{p} \mapsto [\mathtt{q}; \tilde{v}] \in \mu$ *we have* $\mathtt{p} \mapsto [\mathtt{q}; \tilde{w}] \in \mu$ *and either* $\tilde{v} = \varepsilon$ *or* $\tilde{w} = \varepsilon$.
2. *For every* $\mathtt{p} \mapsto [\mathtt{q}; \tilde{v}] \in \mu$ *we have* $\mathsf{tail}(T, \tilde{s}) = S$ *where* $\mathtt{p} : T \in \Gamma_0, \Gamma$ *and* $\Gamma_0|_{\mathsf{fn}(v_i)} \vdash v_i : s_i$ *and* $\|s_i\| < \infty$ *and* $\mathtt{q} \mapsto [\mathtt{p}; \tilde{\varepsilon}] \in \mu$ *then* $\mathtt{q} : \overline{S} \in \Gamma_0, \Gamma$.
3. $\mathsf{dom}(\mu) = \mathsf{dom}(\Gamma_0, \Gamma) = \mu\text{-}\mathsf{reach}(\mathsf{dom}(\Gamma))$.
4. *For every* $\mathtt{U}, \mathtt{V} \subseteq dom(\Gamma)$ *with* $\mathtt{U} \cap \mathtt{V} = \emptyset$ *we have* $\mu\text{-}\mathsf{reach}(\mathtt{U}) \cap \mu\text{-}\mathsf{reach}(\mathtt{V}) = \emptyset$.

In words, condition (1) states that in any pair of peer endpoints, one queue is always empty. This condition corresponds to *half-duplex communication*, whereby it is not possible to send messages over one endpoint before all pending messages from that endpoint have been read. Condition (2) states that the content of the queue associated with an endpoint is consistent with the type of the endpoint, that all messages in a queue have a type with finite weight, and that the residual type of an endpoint after all of the enqueued messages are received is dual of the type of its peer. Condition (3) states that all objects in the heap are reachable from the roots. Since the root pointers will be distributed linearly among the processes in the system, this means that there are no leaks. Finally, condition (4) says that every object in the heap is reachable from exactly one root, ensuring process isolation. Now we formalize the notion of well-typed system.

Definition 4.2 (well-typed system). *We say that the system* $\mu \fatsemi P$ *is* well typed *under* $\Gamma_0; \Gamma$, *written* $\Gamma_0; \Gamma \vdash \mu \fatsemi P$, *if* $\Gamma_0; \Gamma \Vdash \mu$ *and* $\Gamma \vdash P$.

We conclude this section by stating the two main results: well-typedness of systems is preserved by reductions and well-typed processes are well behaved. The proof of Theorem 4.1 relies on the finite-weight restriction on the type of messages for ensuring that no cycles are generated in the heap.

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

Theorem 4.2 (soundness). *If* $\vdash P$ *then* P *is well behaved.*

5 Related Work

This work is the convergence point of several lines of research, including the study and development of Singularity OS [9,10], the formalization of copyless messaging as a communication paradigm [1,2], the development of type systems for session-oriented functional languages [6], and polymorphic session types [4]. The fact that a linear type system is insufficient for preventing memory leaks in copyless messaging was first pointed out in [1,11]. In particular, in [1] and later in [2] we have put forward the idea of *type weight* as the characteristic quantity that allows us to discriminate between safe and unsafe messages. The main limit of the notion of type weight in [1,2] is that it is defined for endpoint types only, for which the weight is entirely determined by the structure of types. In this work we have shown that this is not always the case. Our motivation for studying the extension of the technique developed in [1,2] to a functional language is twofold: first of all, [6] already presents an elegant type system for such a language, even though [6] does not consider explicit memory management. Second, the $\texttt{Sing}^{\#}$ programming language used for the development of Singularity OS includes features such as first-class and anonymous functions, which are commonly found in functional languages. In this setting, the idea of having functions as messages turns out to be a natural one. Another major difference between the present work and [6] is that we develop a truly polymorphic type system in the style of [12], while [6] only considers monomorphic types except for communication primitives which benefit from a form of *ad hoc* polymorphism. In this sense, the present work constitutes also a smooth extension of the type system in [6] with ML-style polymorphism. Interestingly, the polymorphic type of the `open` primitive crucially relies on dualized endpoint type variables, which were introduced in [4] for totally different reasons. Note also that [6] introduces a notion of "size" for session types that may be easily confused with out notion of type weight. In [6], the size estimates the maximum number of enqueued messages in an endpoint and it is used for efficient, static allocation of endpoints with finite-size type. Our weights are unrelated to the size of queues and concern the length of chains of pointers involving queues.

6 Conclusions and Future Work

The type language we have developed is a relatively simple variant of that required for ML-style functional languages. Many features that are practically relevant can be added without posing substantial issues. For instance, it is feasible to devise a subtyping relation it in the style of [6] whereby unlimited functions can be used in place of linear ones ($t \to s \leq t\ w{\multimap}\ s$). Subtyping can also take into account weights, in the sense that it is safe to use a "lighter" function where a "heavier" function is expected ($t\ w{\multimap}\ s \leq t\ w'{\multimap}\ s$ if $w \leq w'$). It is also easy to equip endpoint types with the dual constructs $T \oplus S$ and $T + S$ for denoting internal and external choices driven by boolean values.

The finite-weight restriction on the type of messages prevents the formation of cycles in the heap. In the context of Singularity OS, this restriction seems

to be reasonable since objects allocated in the exchange heap are managed by means of reference counting which cannot handle cyclic structures.

The type system we have presented (Table 5) is not syntax-directed and therefore leaves room for a fair amount of "guessing", in particular with respect to the introduction of type variables in types and weights. An open question is whether it is feasible to devise a fully automated type and weight inference algorithm that is capable of inferring the most general type of arbitrary expressions.

Acknowledgments. This work has been partially supported by MIUR PRIN 2010-2011 CINA. The authors are grateful to the anonymous referees for their comments.

References

1. V. Bono, C. Messa, and L. Padovani. Typing Copyless Message Passing. In *Proceedings of ESOP'11*, LNCS 6602, pages 57–76. Springer, 2011.
2. V. Bono and L. Padovani. Typing Copyless Message Passing. *Logical Methods in Computer Science*, 8:1–50, 2012.
3. V. Bono, L. Padovani, and A. Tosatto. Polymorphic Types for Leak Detection in a Session-Oriented Functional Language, 2013. Available at `http://www.di.unito.it/~padovani/Papers/BonoPadovaniTosatto13.pdf`.
4. S. Gay. Bounded Polymorphism in Session Types. *Mathematical Structures in Computer Science*, 18(5):895–930, 2008.
5. S. Gay and M. Hole. Subtyping for Session Types in the π-calculus. *Acta Informatica*, 42(2-3):191–225, 2005.
6. S. Gay and V. T. Vasconcelos. Linear Type Theory for Asynchronous Session Types. *Journal of Functional Programming*, 20(01):19–50, 2010.
7. K. Honda. Types for Dyadic Interaction. In *Proceedings of CONCUR'93*, LNCS 715, pages 509–523. Springer, 1993.
8. K. Honda, V. T. Vasconcelos, and M. Kubo. Language Primitives and Type Disciplines for Structured Communication-based Programming. In *Proceedings of ESOP'98*, LNCS 1381, pages 122–138. Springer, 1998.
9. G. Hunt, J. Larus, M. Abadi, M. Aiken, P. Barham, M. Fähndrich, C. Hawblitzel, O. Hodson, S. Levi, N. Murphy, B. Steensgaard, D. Tarditi, T. Wobber, and B. Zill. An Overview of the Singularity Project. Technical Report MSR-TR-2005-135, Microsoft Research, 2005.
10. G. C. Hunt and J. R. Larus. Singularity: Rethinking the Software Stack. *SIGOPS Operating Systems Review*, 41:37–49, April 2007.
11. J. Villard. *Heaps and Hops*. PhD thesis, Laboratoire Spécification et Vérification, ENS Cachan, France, 2011.
12. A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.