TYPING COPYLESS MESSAGE PASSING

VIVIANA BONO AND LUCA PADOVANI

Dipartimento di Informatica, Università degli Studi di Torino, Torino, Italy
e-mail address: {bono,padovani}@di.unito.it

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 memory faults, memory leaks, and communication errors. The type system is essentially linear, but we show that linearity alone is inadequate, because it leaves room for scenarios where well-typed processes leak significant amounts of memory. We address these problems basing the type system upon an original variant of session types.

1. INTRODUCTION

Communicating systems pervade every modern computing environment ranging from light-weight threads in multi-core architectures to Web services deployed over wide area networks. *Message passing* is a widespread communication paradigm adopted in many such systems. In this paradigm, it is usually the case that a message traveling on a channel is *copied* from the source to the destination. This is inevitable in a distributed setting, where the communicating parties are loosely coupled, but some small-scale systems grant access to a shared address space. In these cases it is possible to conceive a different communication paradigm – *copyless message passing* – where only *pointers* to messages are copied from the source to the destination. The Singularity Operating System (Singularity OS for short) [15, 16] is a notable example of system that adopts the copyless paradigm. In Singularity OS, processes have access to their own local memory as well as to a region called *exchange heap* that is shared by all processes in the system and that is explicitly managed (objects on the exchange heap are not garbage collected, but are explicitly allocated and deallocated by processes). Inter-process communication solely occurs by means of message passing over *channels* allocated on the exchange heap and messages are themselves pointers to the exchange heap.

The copyless paradigm has obvious performance advantages, because it may dramatically decrease the overhead caused by copying (possibly large) messages. At the same time, it fosters the proliferation of subtle programming errors due to the explicit handling

of pointers and the sharing of data. For this reason, Singularity processes must respect an *ownership invariant*: at any given point in time, each object allocated on the exchange heap is owned by exactly one process. In addition, inter-process communication is regulated by so-called *channel contracts* which specify, for each channel, the sequences of interactions that are expected to occur. Overall, these features are meant to prevent *memory faults* (the access to non-owned/deallocated/uninitialized objects on the exchange heap), *memory leaks* (the accumulation of unreachable allocated objects on the exchange heap), and communication errors which could cause the abnormal termination of processes and trigger the previous kinds of errors.

In this paper we attempt at providing a formal foundation to the copyless paradigm from a type-theoretic point of view, along the following lines:

- We develop a process calculus that captures the essential features of Singularity OS and formalizes a substantial fragment of $\mathsf{Sing}^\#$, the programming language specifically designed for the development of programs that run in Singularity OS. We provide a formal characterization of *well-behaved systems*, those that are free from memory faults, memory leaks, and communication errors.
- We develop a type system ensuring that well-typed systems are well behaved. The type system is fundamentally based on the *linear usage* of pointers and on *endpoint types*, a variant of session types [13, 14, 22] tailored to the communication model of Singularity OS. We provide evidence that session types are a natural and expressive formalization of channel contracts.
- We show that the combination of linearity and endpoint types is insufficient for preserving the ownership invariant, but also that endpoint types convey enough information to tighten the type system so as to guarantee its soundness. This allows us to give an indirect soundness proof of the current Singularity OS implementation.

The rest of the paper is organized as follows. In Section 2 we take a quick tour of $\mathsf{Sing}^\#$ and we focus on its peculiar features in the context of Singularity OS that we are going to study more formally in the subsequent sections. In Section 3 we define syntax and semantics (in terms of subtyping) of the type language for our type system. We also give a number of examples showing how to represent the $\mathsf{Sing}^\#$ types and channel contracts encountered in Section 2 into our type language. Section 4 presents the syntax and reduction semantics of the process calculus and ends with the formal definition of well-behaved systems. Since we want to model the copyless paradigm, our calculus includes an explicit representation of the exchange heap and of the objects allocated therein. Names in the language represent pointers to the exchange heap rather than abstract communication channels. Section 5 begins showing that a traditionally conceived type system based on linearity and behavioral types may leave room for violations of the ownership invariant. We then devise a type-theoretic approach to solve the problem, we present the type rules for the exchange heap and the process calculus and the soundness results of the type system. In Section 6 we define algorithms for deciding the subtyping relation and for implementing the type checking rules presented in the previous section. We relate our work with relevant literature in Section 7, where we also detail similarities and differences between this paper and two earlier versions [2, 3] that have appeared in conference and workshop proceedings. We conclude in Section 8 with a summary of our work. For the sake of readability, proofs and additional technical material relative to Sections 3, 5, and 6 have been moved into Appendixes A, B, and C respectively.

```
1   void map<α,β>(imp<Mapper<α,β>:WAIT_ARG> in ExHeap mapper,
2                 [Claims] imp<Stream<α>:START> in ExHeap source,
3                 [Claims] exp<Stream<β>:START> in ExHeap target) {
4     switch receive {
5       case source.Data(α in ExHeap x):
6         mapper.Arg(x);
7         switch receive {
8           case mapper.Res(β in ExHeap y):
9             target.Data(y);
10            map<α,β>(mapper, source, target);
11        }
12
13      case source.Eos():
14        target.Eos();
15        source.Close();
16        target.Close();
17    }
18  }
```

Figure 1: An example of Sing$^{\#}$ code.

2. A TASTE OF SING$^{\#}$

In this section we take a closer look at Sing$^{\#}$, the programming language specifically designed for the development of programs that run in Singularity OS. We do so by means of a simple, yet rather comprehensive example that shows the main features of the language and of its type system. In the discussion that follows it is useful to keep in mind that Singularity channels consist of pairs of related *endpoints*, called the *peers* of the channel. Messages sent over one peer are received from the other peer, and vice versa. Each peer is associated with a FIFO buffer containing the messages sent to that peer that have not been received yet. Therefore, communication is asynchronous (send operations are non-blocking) and process synchronization must be explicitly implemented by means of suitable handshaking protocols.

The code snippet in Figure 1 defines a polymorphic function map that transforms a stream of data of type α into a stream of data of type β through a provided mapper.[1] The function accepts two type arguments α and β and three proper arguments: a mapper endpoint that allows communication with a process that performs the actual processing of data; a source endpoint from which data to be processed is read; a target endpoint to which processed data is forwarded. For the time being, we postpone the discussion of the type annotations of these arguments and focus instead on the operational semantics of the function. We will come back to types shortly, when we discuss static analysis. The switch receive construct (lines 4–17) is used to receive messages from an endpoint, and to dispatch the control flow to various cases depending on the kind of message that is received. Each case block specifies the endpoint from which a message is expected and the

---

[1]This function can be thought of as the communication-oriented counterpart of the higher-order, list-processing map function defined in the standard library of virtually all functional programming languages.

tag of the message. In this example, two kinds of messages can be received from the `source` endpoint: either a `Data`-tagged message (lines 5–11) or a `Eos`-tagged message (lines 13–16). A `Data`-tagged message contains a chunk of data to be processed, which is bound to the local variable `x` (line 5). The data is sent in an `Arg`-tagged message on the `mapper` endpoint for processing (line 6), the result is received from the same endpoint as a `Res`-tagged message, stored in the local variable `y` (line 8) and forwarded on the `target` endpoint as another outgoing `Data`-tagged message (line 9). Finally, the `map` function is invoked recursively so that further data can be processed (line 10). An `Eos`-tagged message flags the fact that the incoming stream of data is finished (line 13). When this happens, the same kind of message is sent on the `target` endpoint (line 14) and both the `source` and the `target` endpoints are closed (lines 15 and 16).

We now illustrate the meaning of the type annotations and their relevance with respect to static analysis. The `in ExHeap` annotations state that all the names in this example denote pointers to objects allocated on the exchange heap. Some of these objects (like those pointed to by `source` and `target`) represent communication endpoints, others (those pointed to by `x` and `y`) represent data contained in messages. Static analysis of $\mathsf{Sing}^{\#}$ programs aims at providing strong guarantees on the absence of errors deriving from communications and the usage of heap-allocated objects.

Regarding communications, the correctness of this code fragment relies on the assumption that the process(es) using the peer endpoints of `mapper`, `source`, and `target` are able to deal with the message types as they are received/sent from within `map`. For instance, `map` assumes to receive a `Res`-tagged message *after* it has sent an `Arg`-tagged message on `mapper`. It also assumes that only `Data`-tagged and `Eos`-tagged messages can be received from `source` and sent to `target`, and that after an `Eos`-tagged message is received no further message can be received from it. No classical type associated with `mapper` or `source` or `target` is able to capture these temporal dependencies between such different usages of the same object at different times. The designers of $\mathsf{Sing}^{\#}$ have consequently devised *channel contracts* describing the allowed communication patterns on a given endpoint. Consider, for example, the polymorphic contracts `Mapper<`α,β`>` and `Stream<`α`>` below:

```
contract Mapper<α,β> {                contract Stream<α> {
  message Arg(α in ExHeap);             message Data(α in ExHeap);
  message Res(β in ExHeap);             message Eos();
  state WAIT_ARG                        state START
  { Arg? → SEND_RES; }                  { Data! → START;
  state SEND_RES                          Eos! → END; }
  { Res! → WAIT_ARG; }                  state END { }
}                                     }
```

A contract is made of a finite set of *message specifications* and a finite set of *states* connected by *transitions*. Each message specification begins with the `message` keyword and is followed by the *tag* of the message and the type of its arguments. For instance, the `Stream<`α`>` contract defines the `Data`-tagged message with an argument of type α and the `Eos`-tagged message with no arguments. The state of the contract determines the state in which the endpoint associated with the contract is and this, in turn, determines which messages can be sent/received. The same contract can have multiple states, each with a possibly different set of messages that can be sent/received, therefore capturing the behavioral

nature of endpoints. In Stream<α> we have a START state from which two kinds of message can be sent: if a Data-tagged message is sent, the contract remains in the START state; if a Eos-tagged message is sent, the contract transits to the END state from which no further transitions are possible. Communication errors are avoided by associating the two peers of a channel with types that are complementary, in that they specify complementary actions. This is achieved in Sing$^\#$ with the exp<C:s> and imp<C:s> type constructors that, given a contract C and a state s of C, respectively denote the so-called *exporting* and *importing* views of C when in state s. For the sake of hindsight, it is useful to think of the exporting view as of the type of the *provider* of the behavior specified in the contract, and of the importing view as of the type of the *consumer* of the behavior specified in the contract. On the one hand, the map function in Figure 1 accepts a mapper argument of type imp<Map<α,β>:WAIT_ARG> since it consumes the mapping service accessible through the mapper endpoint and a source argument of type imp<Stream<α>:START> since it consumes the source stream of data to be processed. On the other hand, the function accepts a target argument of type exp<Stream<β>:START> since it produces a new stream of data on the target endpoint. In the code fragment in Figure 1, the endpoint target has type exp<Stream<β>:START> on line 9, the output of a Data-tagged message is allowed by the exporting view of Stream<β> in this state, and the new type of target on line 10 is again exp<Stream<β>:START>. Its type turns to exp<Stream<α>:END> from line 14 to line 15, when the Eos-tagged message is received. The endpoint mapper has type imp<Mapper<α,β>:WAIT_ARG> on line 6. The importing view of Mapper<α,β> allows sending a Arg-tagged message in this state, hence the type of mapper turns to imp<Mapper<α,β>:SEND_RES> in lines 7 and back to type imp<Mapper<α,β>:WAIT_ARG> from line 8 to line 9.

A major complication of the copyless paradigm derives from the fact that communicated objects are not copied from the sender to the receiver, but rather pointers to allocated objects are passed around. This can easily invalidate the ownership invariant if special attention is not payed to whom is entitled to access which objects. Given these premises, it is natural to think of a type discipline controlling the *ownership* of allocated objects, whereby at any given time every allocated object is owned by one (and only one) process. Whenever (the pointer to) an allocated object is sent as a message, its ownership is also transferred from the sender to the receiver. In the example of Figure 1, the function map becomes the owner of data x in line 5. When x is sent on endpoint mapper, the ownership of x is transferred from map to whichever process is receiving messages on mapper's peer endpoint. Similarly, map acquires the ownership of y on line 8, and ceases it in the subsequent line. Overall it seems like map is well balanced, in the sense that everything it acquires it also released. In fact, as mapper, source, and target are also allocated on the exchange heap, we should care also for map's arguments. Upon invocation of map, the ownership of these three arguments transfers from the caller to map, but when map terminates, only the ownership of mapper returns to the caller, since source and target are closed (and deallocated) within map on lines 15 and 16. This is the reason why the types of source and target in the header of map are annotated with a [Claims] clause indicating that map retains the ownership of these two arguments even after it has returned.

From the previous discussion it would seem plausible to formalize Sing$^\#$ using a process calculus equipped with a suitable session type system. Session types capture very well the sort of protocols described by Sing$^\#$ contracts and one could hope that, by imposing a *linear* usage on entities, the problems regarding the ownership of heap-allocated objects would be easily solved. In practice, things are a little more involved than this because, somewhat

surprisingly, linearity alone is *too weak* to guarantee the absence of *memory leaks*, which occur when every reference to an heap-allocated object is lost. We devote the rest of this section to illustrating this issue through a couple of simple examples. Consider the function:

```
void foo([Claims] imp<C:START> in ExHeap e,
         [Claims] exp<C:START> in ExHeap f)
{ e.Arg(f); e.Close(); }
```

which accepts two endpoints `e` and `f` allocated in the exchange heap, sends endpoint `f` as an `Arg`-tagged message on `e`, and closes `e`. The `[Claims]` annotations in the function header are motivated by the fact that one of the two arguments is sent away in a message, while the other is properly deallocated within the function. Yet, this function may produce a leak if `e` and `f` are the peer endpoints of the same channel. If this is the case, only the `e` endpoint is properly deallocated while every reference to `f` is lost. Note that the `foo` function behaves correctly with respect to the $\mathsf{Sing}^\#$ contract

```
contract C {
  message Arg(exp<C:START> in ExHeap);
  state START { Arg? → END; }
  state END { }
}
```

whose only apparent anomaly is the implicit recursion in the type of the argument of the `Arg` message, which refers to the contract `C` being defined. A simple variation of `foo` and `C`, however, is equally dangerous and does not even need this form of implicit recursion:

```
void bar([Claims] imp<D:START> in ExHeap e,
         [Claims] exp<D:START> in ExHeap f)
{ e.Arg<exp<D:START>>(f); e.Close(); }
```

In this case, the `Arg`-tagged message is polymorphic (it accepts a linear argument of *any* type) and the contract `D` is defined as:

```
contract D {
  message Arg<α>(α in ExHeap);
  state START { Arg? → END; }
  state END { }
}
```

These examples show that, although it makes sense to allow the types `exp<C:START>` and `exp<D:START>` in general, their specific occurrences in the definition of `C` and in the body of `bar` are problematic. We will see why this is the case in Section 5 and we shall devise a purely type-theoretic framework that avoids these problems. Remarkably, the `foo` function is ill typed also in $\mathsf{Sing}^\#$ [8], although the motivations for considering `foo` dangerous come from the implementation details of ownership transfer rather than from the memory leaks that `foo` can produce (see Section 7 for a more detailed discussion).

Table 1: Syntax of types.

Type	t	$::=$	$q\ T$	(qualified endpoint type)
Qualifier	q	$::=$	lin	(linear)
		$\mid$	un	(unrestricted)
Endpoint Type	T	$::=$	end	(termination)
		$\mid$	α	(type variable)
		$\mid$	$\{!\mathtt{m}_i\langle\alpha_i\rangle(t_i).T_i\}_{i\in I}$	(internal choice)
		$\mid$	$\{?\mathtt{m}_i\langle\alpha_i\rangle(t_i).T_i\}_{i\in I}$	(external choice)
		$\mid$	$\mathtt{rec}\ \alpha.T$	(recursive type)

3. Types

We introduce some notation for the type language: we assume an infinite set of *type variables* ranged over by α, β, $\dots$; we use t, s, $\dots$ to range over types, q to range over qualifiers, and T, S, $\dots$ to range over endpoint types. The syntax of types and endpoint types is defined in Table 1. An endpoint type describes the allowed behavior of a process with respect to a particular endpoint. The process may send messages over the endpoint, receive messages from the endpoint, and deallocate the endpoint. The endpoint type end denotes an endpoint on which no input/output operation is possible and that can only be deallocated. An internal choice $\{!\mathtt{m}_i\langle\alpha_i\rangle(t_i).T_i\}_{i\in I}$ denotes an endpoint on which a process may send any message with tag $\mathtt{m}_i$ for $i \in I$. The message has a *type parameter* α_i, which the process can instantiate with any endpoint type (but we will impose some restrictions in Section 5), and an argument of type t_i. Depending on the tag $\mathtt{m}_i$ of the message, the endpoint can be used thereafter according to the endpoint type T_i. In a dual manner, an external choice $\{?\mathtt{m}_i\langle\alpha_i\rangle(t_i).T_i\}_{i\in I}$ denotes and endpoint from which a process must be ready to receive any message with tag $\mathtt{m}_i$ for $i \in I$. Again, α_i is the type parameter of the message and t_i denotes the type of the message's argument. Depending on the tag $\mathtt{m}_i$ of the received message, the endpoint is to be used according to T_i. The duality between internal and external choices regards not only the dual send/receive behaviors of processes obeying these types, but also the quantification of type parameters in messages, which we can think universally quantified in internal choices (the sender chooses how to instantiate the type variable) and existentially quantified in external choices (the receiver does not know the type with which the type variable has been instantiated). In endpoint types $\{!\mathtt{m}_i\langle\alpha_i\rangle(t_i).T_i\}_{i\in I}$ and $\{?\mathtt{m}_i\langle\alpha_i\rangle(t_i).T_i\}_{i\in I}$ we assume that $\mathtt{m}_i = \mathtt{m}_j$ implies $i = j$. That is, the tag $\mathtt{m}_i$ of the message that is sent or received identifies a unique continuation T_i. Terms $\mathtt{rec}\ \alpha.T$ can be used to specify recursive behaviors, as usual. The role of type variables α is twofold, depending on whether they are bound by a recursion $\mathtt{rec}\ \alpha.T$ or by a prefix $\mathtt{m}\langle\alpha\rangle(t)$ in a choice: they either represent recursion points, like α in $\mathtt{rec}\ \alpha.!\mathtt{m}\langle\beta\rangle(t).\alpha$, or abstracted endpoint types, like α in $!\mathtt{m}\langle\alpha\rangle(\mathsf{lin}\ ?\mathtt{m}'\langle\beta\rangle(t).\alpha).\mathsf{end}$. We will see plenty of examples of both usages in the following.

Even though the type system focuses on linear objects allocated on the exchange heap, the type language must be expressive enough to describe Singularity OS entities like system-wide services or $\mathsf{Sing}^{\#}$ functions and procedures. For this reason, we distinguish *linear* resources from *unrestricted* ones and, along the lines of [22, 12], we define types as qualified

Table 2: Well-formedness rules for endpoint types.

$$\text{(WF-End)} \qquad \frac{\text{(WF-Var)}}{\alpha \in \Delta_O \setminus \Delta_I} \qquad \frac{\text{(WF-Rec)}}{\Delta_O, \alpha; \Delta_I \setminus \{\alpha\} \Vdash T}$$

$$\Delta_O; \Delta_I \Vdash \mathsf{end} \qquad \frac{\alpha \in \Delta_O \setminus \Delta_I}{\Delta_O; \Delta_I \Vdash \alpha} \qquad \frac{\Delta_O, \alpha; \Delta_I \setminus \{\alpha\} \Vdash T}{\Delta_O; \Delta_I \Vdash \mathtt{rec}\ \alpha.T}$$

$$\text{(WF-Prefix)}$$

$$\frac{\dagger \in \{!, ?\} \qquad (\Delta_O \cup \Delta_I), \alpha_i; \emptyset \Vdash S_i \ ^{(i \in I)} \qquad \Delta_O; \Delta_I, \alpha_i \Vdash T_i \ ^{(i \in I)}}{\Delta_O; \Delta_I \Vdash \dagger\{\mathtt{m}_i\langle\alpha_i\rangle(q_i\ S_i).T_i\}_{i \in I}}$$

endpoint types. A qualifier is either 'lin', denoting a *linear endpoint type* or 'un', denoting an *unrestricted endpoint type*. Endpoints with a linear type must be owned by exactly one process at any given time, whereas endpoints with an unrestricted type can be owned by several (possibly zero) processes at the same time. Clearly, not every endpoint type can be qualified as unrestricted, for the type system relies fundamentally on linearity in order to enforce its properties. In the following we limit the use of the 'un' qualifier to endpoint types of the form $\mathtt{rec}\ \alpha.\{!\mathtt{m}_i\langle\alpha_i\rangle(t_i).\alpha\}_{i \in I}$, whose main characteristic is that they do not change over time (each continuation after an output action is α, that is the whole endpoint type itself). In a sense, they are not behavioral types, which intuitively explains why they can be safely qualified as unrestricted.

Here are some conventions regarding types and endpoint types:

- we sometimes use an infix notation for internal and external choices and write

$$!\mathtt{m}_1\langle\alpha_1\rangle(t_1).T_1 \oplus \cdots \oplus !\mathtt{m}_n\langle\alpha_n\rangle(t_n).T_n \qquad \text{instead of} \qquad \{!\mathtt{m}_i\langle\alpha_i\rangle(t_i).T_i\}_{i \in \{1,\dots,n\}}$$

and

$$?\mathtt{m}_1\langle\alpha_1\rangle(t_1).T_1 + \cdots + ?\mathtt{m}_n\langle\alpha_n\rangle(t_n).T_n \qquad \text{instead of} \qquad \{?\mathtt{m}_i\langle\alpha_i\rangle(t_i).T_i\}_{i \in \{1,\dots,n\}}$$

- we omit the type variable specification $\langle\alpha\rangle$ when useless (if the type variable occurs nowhere else) and write, for example, $!\mathtt{m}(t).T$;
- for the sake of simplicity, we formally study (endpoint) types where messages carry exactly one type/value argument, but we will be more liberal in the examples;
- we write $\mathsf{lin}(t)$ and $\mathsf{un}(t)$ to mean that t is respectively linear and unrestricted.

We have standard notions of free and bound type variables for (endpoint) types. The binders are $\mathtt{rec}$ and $\mathtt{m}\langle\alpha\rangle(t)$. In particular, $\mathtt{rec}\ \alpha.T$ binds α in T and $\dagger\mathtt{m}\langle\alpha\rangle(t).T$ where $\dagger \in \{!, ?\}$ binds α in t and in T. We will write $\mathtt{ftv}(T)$ and $\mathtt{btv}(T)$ for the set of free and bound type variables of T. We require that type variables bound by a recursion $\mathtt{rec}$ must be guarded by a prefix (therefore a non-contractive endpoint type such as $\mathtt{rec}\ \alpha.\alpha$ is forbidden) and that type variables bound in $\langle\alpha\rangle$ as in $!\mathtt{m}\langle\alpha\rangle(t).T$ can only occur in t and within the prefixes of T. We formalize this last requirement as a well-formedness predicate for types denoted by a judgment $\Delta_O; \Delta_I \Vdash T$ and inductively defined by the axioms and rules in Table 2. The set Δ_O contains so-called *outer variables* (those that can occur everywhere) while the set Δ_I contains so-called *inner variables* (those that can occur only within prefixes). Here and in the following we adopt the convention that Δ, Δ' denotes $\Delta \cup \Delta'$ when $\Delta \cap \Delta' = \emptyset$ and is undefined otherwise; we also write Δ, α instead of $\Delta, \{\alpha\}$. We say that T is well formed with respect to Δ, written $\Delta \Vdash T$, if $\Delta; \emptyset \Vdash t$ is derivable. Well formedness restricts the expressiveness of types, in particular endpoint types such as

$!\mathtt{m}\langle\alpha\rangle(t).\alpha$ and $?\mathtt{m}\langle\alpha\rangle(t).\alpha$ are not admitted because ill formed. We claim that ill-formed endpoint types have little practical utility: a process using an endpoint with type $!\mathtt{m}\langle\alpha\rangle(t).\alpha$ knows the type with which α is instantiated while no process is capable of using an endpoint with type $?\mathtt{m}\langle\alpha\rangle(t).\alpha$ since nothing can be assumed about the endpoint type with which α is instantiated.

In what follows we consider endpoint types modulo renaming of bound variables and the law $\mathtt{rec}\ \alpha.T = T\{\mathtt{rec}\ \alpha.T/\alpha\}$ where $T\{\mathtt{rec}\ \alpha.T/\alpha\}$ is the capture-avoiding substitution of $\mathtt{rec}\ \alpha.T$ in place of every free occurrence of α in T. Whenever we want to reason on the structure of endpoint types, we will use a syntactic equality operator $\equiv$. Therefore we have $\mathtt{rec}\ \alpha.T \not\equiv T\{\mathtt{rec}\ \alpha.T/\alpha\}$ (recall that T cannot be α for contractivity).

Example 3.1. Consider the contracts $\mathtt{Mapper}\texttt{<}\alpha\texttt{,}\beta\texttt{>}$ and $\mathtt{Stream}\texttt{<}\alpha\texttt{>}$ presented in Section 2. We use the endpoint types

$$\begin{aligned} T_{\mathtt{Mapper}}(\alpha,\beta) &= \mathtt{rec}\ \gamma.?\mathsf{Arg}(\mathsf{lin}\ \alpha).!\mathsf{Res}(\mathsf{lin}\ \beta).\gamma \\ T_{\mathtt{Stream}}(\alpha) &= \mathtt{rec}\ \gamma.(!\mathsf{Data}(\mathsf{lin}\ \alpha).\gamma \oplus !\mathsf{Eos}().\mathsf{end}) \end{aligned}$$

to denote the $\mathsf{Sing}^{\#}$ types $\texttt{exp<Mapper<}\alpha\texttt{,}\beta\texttt{>:WAIT\_ARG>}$ and $\texttt{exp<Stream<}\alpha\texttt{>:START>}$ respectively. Recursion models loops in the contracts and each state of a contract corresponds to a particular subterm of $T_{\mathtt{Mapper}}(\alpha,\beta)$ and $T_{\mathtt{Stream}}(\alpha)$. For instance, the $\mathsf{Sing}^{\#}$ type $\texttt{exp<Mapper<}\alpha\texttt{,}\beta\texttt{>:SEND\_RES>}$ is denoted by the endpoint type $!\mathsf{Res}(\mathsf{lin}\ \beta).T_{\mathtt{Mapper}}(\alpha,\beta)$. The type of message arguments are embedded within the endpoint types, like in session types but unlike $\mathsf{Sing}^{\#}$ where they are specified in separate $\texttt{message}$ directives. The lin qualifiers correspond to the $\texttt{in ExHeap}$ annotations and indicate that these message arguments are *linear* values.

Observe that both endpoint types are open, as the type variables α and β occur free in them. We will see how to embed these endpoint types into a properly closed type for $\texttt{map}$ in Example 3.3. ∎

Duality is a binary relation between endpoint types that describe complementary actions. Peer endpoints will be given dual endpoint types, so that processes accessing peer endpoints will interact without errors: if one of the two processes sends a message of some kind, the other process is able to receive a message of that kind; if one process has finished using an endpoint, the other process has finished too.

Definition 3.1 (duality). We say that $\mathscr{D}$ is a *duality relation* if $(T, S) \in \mathscr{D}$ implies either

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

We write $\bowtie$ for the largest duality relation and we say that T and S are *dual* if $T \bowtie S$.

We will see that every well-formed endpoint type T has a dual – that we denote by $\overline{T}$ – which is intuitively obtained from T by swapping ?'s with !'s. The formal definition of $\overline{T}$, however, is complicated by the possible occurrence of recursion variables within prefixes. As an example, the dual of the endpoint type $T = \mathtt{rec}\ \alpha.!\mathtt{m}\langle\beta\rangle(\alpha).\mathsf{end}$ is *not* $S = \mathtt{rec}\ \alpha.?\mathtt{m}\langle\beta\rangle(\alpha).\mathsf{end}$ but rather $\mathtt{rec}\ \alpha.?\mathtt{m}\langle\beta\rangle(T).\mathsf{end}$. This is because, by unfolding the recursion in T, we obtain $T = !\mathtt{m}\langle\beta\rangle(T).\mathsf{end}$ whose dual, $?\mathtt{m}\langle\beta\rangle(T).\mathsf{end}$, is clearly different from $S = ?\mathtt{m}\langle\beta\rangle(S).\mathsf{end}$ (duality does not change the type of message arguments).

To provide a syntactic definition of dual endpoint type, we use an *inner substitution operator* $\{\{\cdot/\cdot\}\}$ such that $T\{\{S/\alpha\}\}$ denotes T where every free occurrence of α *within the*

prefixes of T has been replaced by S. Free occurrences of α that do not occur within a prefix of T are not substituted. For example, we have $(!\mathtt{m}\langle\beta\rangle(\alpha).\alpha)\{\{S/\alpha\}\} = !\mathtt{m}\langle\beta\rangle(S).\alpha$. Then, the *dual* of an endpoint type T is defined inductively on the structure of T, thus:

$$
\begin{aligned}
\overline{\mathsf{end}} &= \mathsf{end} \\
\overline{\alpha} &= \alpha \\
\overline{\mathtt{rec}\ \alpha.T} &= \mathtt{rec}\ \alpha.\overline{T\{\{\mathtt{rec}\ \alpha.T/\alpha\}\}} \\
\overline{\{!\mathtt{m}_i\langle\alpha_i\rangle(t_i).T_i\}_{i\in I}} &= \{?\mathtt{m}_i\langle\alpha_i\rangle(t_i).\overline{T_i}\}_{i\in I} \\
\overline{\{?\mathtt{m}_i\langle\alpha_i\rangle(t_i).T_i\}_{i\in I}} &= \{!\mathtt{m}_i\langle\alpha_i\rangle(t_i).\overline{T_i}\}_{i\in I}
\end{aligned}
$$

Here are some important facts about well-formed endpoint types and duality:

Proposition 3.1. *The following properties hold:*
1. $\overline{\overline{T}} = T$.
2. $\emptyset \Vdash T$ *implies that* $T \bowtie \overline{T}$ *and* $\emptyset \Vdash \overline{T}$.
3. $\Delta; \{\alpha\} \Vdash T$ *and* $\Delta \Vdash S$ *imply* $\Delta \Vdash T\{S/\alpha\}$.
4. $\emptyset; \{\alpha\} \Vdash T$ *and* $\emptyset \Vdash S$ *imply* $\overline{T\{S/\alpha\}} = \overline{T}\{S/\alpha\}$.

Item (1) states that $\overline{\ \cdot\ }$ is an involution. Item (2) states that $\overline{T}$ is well formed and dual of T when T is well formed. Item (3) states the expected property of well-formedness preservation under substitution of well-formed endpoint types. Finally, item (4) shows that duality does not affect the inner variables of an endpoint type and that, in fact, duality and substitution commute.

Example 3.2. In Example 3.1 we have defined the endpoint types $T_{\mathtt{Mapper}}(\alpha, \beta)$ and $T_{\mathtt{Stream}}(\alpha)$ denoting the `exp<Mapper<`α`,`β`>:WAIT_ARG>` and `exp<Stream<`α`>:START>` types in $\mathsf{Sing}^{\#}$. The dual endpoint types of $T_{\mathtt{Mapper}}(\alpha, \beta)$ and $T_{\mathtt{Stream}}(\alpha)$ are

$$
\begin{aligned}
\overline{T_{\mathtt{Mapper}}(\alpha, \beta)} &= \mathtt{rec}\ \gamma.!\mathsf{Arg}(\mathsf{lin}\ \alpha).?\mathsf{Res}(\mathsf{lin}\ \beta).\gamma \\
\overline{T_{\mathtt{Stream}}(\alpha)} &= \mathtt{rec}\ \gamma.(?\mathsf{Data}(\mathsf{lin}\ \alpha).\gamma + ?\mathsf{Eos}().\mathsf{end})
\end{aligned}
$$

and they denote the `imp<Mapper<`α`,`β`>:WAIT_ARG>` and `imp<Stream<`α`>:START>` types in $\mathsf{Sing}^{\#}$. ∎

Example 3.3 (function types)**.** While $\mathsf{Sing}^{\#}$ is a procedural language, our formalization is based on a process algebra. Therefore, some $\mathsf{Sing}^{\#}$ entities like functions and function types that are not directly representable must be encoded. A function can be encoded as a process that waits for the arguments and sends the result of the computation. Callers of the function will therefore send the arguments and receive the result. Following this intuition, the type

$$
T_{\mathtt{map}}(\alpha, \beta) = !\mathsf{Arg}(\mathsf{lin}\ \overline{T_{\mathtt{Mapper}}(\alpha, \beta)}).!\mathsf{Arg}(\mathsf{lin}\ \overline{T_{\mathtt{Stream}}(\alpha)}).!\mathsf{Arg}(\mathsf{lin}\ T_{\mathtt{Stream}}(\beta)).?\mathsf{Res}().\mathsf{end}
$$

seems like a good candidate for denoting the type of `map` in Figure 1. This type allows a caller of the function to supply (send) three arguments having type $\overline{T_{\mathtt{Mapper}}(\alpha, \beta)}$, $\overline{T_{\mathtt{Stream}}(\alpha)}$, and $T_{\mathtt{Stream}}(\beta)$ in this order. The lin qualifiers indicates that all the arguments are linear. Since `map` returns nothing, the `Res`-tagged message does not carry any useful value, but it models the synchronous semantics of function invocation.

This encoding of the type of `map` does not distinguish arguments that are *claimed* by `map` from others that are not. The use of the lin qualifier in the encoding is mandated by the fact that the arguments are allocated in the exchange heap, but in this way the caller process permanently loses the ownership of the `mapper` argument, and this is not the

intended semantics of `map`. We can model the temporary ownership transfer as a pair of linear communications, by letting the (encoded) `map` function return any argument that is not claimed. Therefore, we patch the above endpoint type as follows:

$$T_{\text{map}}(\alpha, \beta) = !\text{Arg}(\text{lin } \overline{T_{\text{Mapper}}(\alpha, \beta)}).[\cdots].?\text{Arg}(\text{lin } \overline{T_{\text{Mapper}}(\alpha, \beta)}).?\text{Res}().\text{end}$$

The endpoint type $T_{\text{map}}(\alpha, \beta)$ describes the protocol for one particular invocation of the `map` function. A proper encoding of the type of `map`, which allows for multiple invocations and avoids interferences between independent invocations, is the following:

$$t_{\text{map}} = \text{un rec } \gamma.!\text{Invoke}\langle \alpha, \beta \rangle(\text{lin } \overline{T_{\text{map}}(\alpha, \beta)}).\gamma$$

Prior to invocation, a caller is supposed to create a fresh channel which is used for communicating with the process modeling the function. One endpoint, of type $T_{\text{map}}(\alpha, \beta)$, is retained by the caller, the other one, of type $\overline{T_{\text{map}}(\alpha, \beta)}$, is sent upon invocation to the process modeling `map`. The recursion in t_{map} permits multiple invocation of `map`, and the `un` qualifier indicates that `map` is *unrestricted* and can be invoked simultaneously and independently by multiple processes in the system. ∎

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 have been put forward [10, 9, 5, 22, 20]. We define subtyping in pretty much the same way as in [10, 9].

Definition 3.2 (subtyping). Let $\leq$ be the least preorder on qualifiers such that $\text{un} \leq \text{lin}$. We say that $\mathscr{S}$ is a *coinductive subtyping* if:

- $(q\,T, q'\,S) \in \mathscr{S}$ implies $q \leq q'$ and $(T, S) \in \mathscr{S}$, and
- $(T, S) \in \mathscr{S}$ implies either:
  (1) $T = S = \text{end}$, or
  (2) $T = S = \alpha$, or
  (3) $T = \{?\text{m}_i\langle\alpha_i\rangle(t_i).T_i\}_{i \in I}$ and $S = \{?\text{m}_i\langle\alpha_i\rangle(s_i).S_i\}_{i \in J}$ with $I \subseteq J$ and $(t_i, s_i) \in \mathscr{S}$ and $(T_i, S_i) \in \mathscr{S}$ for every $i \in I$, or
  (4) $T = \{!\text{m}_i\langle\alpha_i\rangle(t_i).T_i\}_{i \in I}$ and $S = \{!\text{m}_i\langle\alpha_i\rangle(s_i).S_i\}_{i \in J}$ with $J \subseteq I$ and $(s_i, t_i) \in \mathscr{S}$ and $(T_i, S_i) \in \mathscr{S}$ for every $i \in J$.

We write $\leqslant$ for the largest coinductive subtyping.

Items (1) and (2) account for reflexivity of subtyping when T and S are both `end` or the same type variable; items (3) and (4) are the usual covariant and contravariant rules for inputs and outputs respectively. Observe that subtyping is always covariant with respect to the continuations. Two types $q_1\,T$ and $q_2\,S$ are related by subtyping if so are T and S and if q_1 is no more stringent than q_2. In particular, it is safe to use an unrestricted value where a linear one is expected.

The reader may verify that subtyping is a pre-order:

Proposition 3.2. $\leqslant$ *is reflexive and transitive.*

Proof sketch. The proofs of both properties are easy exercises. In the case of transitivity it suffices to show that

$$\mathscr{S} \overset{\text{def}}{=} \{(t_1, t_2) \mid \exists s : t_1 \leqslant s \,\&\, s \leqslant t_2\} \cup \{(T_1, T_2) \mid \exists S : T_1 \leqslant S \,\&\, S \leqslant T_2\}$$

is a coinductive subtyping. □

The following property shows that duality is contravariant with respect to subtyping. It is a standard property of session type theories, except that in our case it holds only when the two endpoint types being related have no free type variables occurring at the top level (outside any prefix), for otherwise their duals are undefined (Proposition 3.1).

Proposition 3.3. *Let $\emptyset \Vdash T$ and $\emptyset \Vdash S$. Then $T \leqslant S$ if and only if $\overline{S} \leqslant \overline{T}$.*

Example 3.4. In Example 3.3 we have suggested a representation for the function type $s \to t$ as the type $[\![s \to t]\!]$ defined thus:

$$[\![s \to t]\!] = \text{un rec } \alpha.!\texttt{Invoke}(\text{lin }?\texttt{Arg}(s).!\texttt{Res}(t).\texttt{end}).\alpha$$

It is easy to verify that $[\![s_1 \to t_1]\!] \leqslant [\![s_2 \to t_2]\!]$ if and only if $s_2 \leqslant s_1$ and $t_1 \leqslant t_2$. That is, the subtyping relation between encoded function types is consistent with the standard subtyping between function types, which is contravariant in the domain and covariant in the co-domain.

Another way to interpret an endpoint having type

$$\text{rec } \alpha.!\texttt{Invoke}(t).\alpha$$

is as an object with one method $\texttt{Invoke}$. Sending a $\texttt{Invoke}$-tagged message on the endpoint means invoking the method (incidentally, this is the terminology adopted in SmallTalk), and after the invocation the object is available again with the same interface. We can generalize the type above to

$$\text{rec } \alpha.\{!\texttt{m}_i(t_i).\alpha\}_{i \in I}$$

for representing objects with multiple methods $\texttt{m}_i$. According to the definition of subtyping we have

$$\text{rec } \alpha.\{!\texttt{m}_i(t_i).\alpha\}_{i \in I} \leqslant \text{rec } \alpha.\{!\texttt{m}_j(t_j).\alpha\}_{j \in J}$$

whenever $J \subseteq I$, which corresponds the same notion of subtyping used in object-oriented language (it is safe to use an object offering more methods where one offering fewer methods is expected). ∎

4. Syntax and Semantics of Processes

We assume the existence of an infinite set $\texttt{Pointers}$ of *linear pointers* (or simply *pointers*) ranged over by $a, b, \ldots$, of an infinite set $\texttt{Variables}$ of *variables* ranged over by $x, y, \ldots$, and of an infinite set of *process variables* ranged over by $X, Y, \ldots$. We define the set $\overline{\texttt{Pointers}}$ of *unrestricted pointers* as $\overline{\texttt{Pointers}} = \{\overline{a} \mid a \in \texttt{Pointers}\}$. We assume $\texttt{Pointers}$, $\overline{\texttt{Pointers}}$, and $\texttt{Variables}$ be pairwise disjoint, we let $u, v, \ldots$ range over *names*, which are elements of $\texttt{Pointers} \cup \overline{\texttt{Pointers}} \cup \texttt{Variables}$, and we let $\texttt{v}, \texttt{w}, \ldots$ range over *values*, which are elements of $\texttt{Pointers} \cup \overline{\texttt{Pointers}}$.

Processes, ranged over by $P, Q, \ldots$, are defined by the grammar in Table 3. The calculus of processes is basically a monadic pi calculus equipped with tag-based message dispatching and primitives for handling heap-allocated endpoints. The crucial aspect of the calculus is that names are pointers to the heap and channels are concretely represented as structures allocated on the heap. Pointers can be either linear or unrestricted: a linear pointer must be owned by exactly one process at any given point in time; an unrestricted pointer can be owned by several (possibly zero) processes at any time. In practice the two kinds of pointers are indistinguishable and range over the same address space, but in the calculus we decorate unrestricted pointers with a bar to reason formally on the different

Table 3: Syntax of processes.

Process	P	::=	**0**	(idle)
		$\mid$	$\mathtt{close}(u)$	(close endpoint)
		$\mid$	$\mathtt{open}(a : T, a : T).P$	(open linear channel)
		$\mid$	$\mathtt{open}(a : T).P$	(open unrestricted channel)
		$\mid$	$u!\mathtt{m}\langle T\rangle(u).P$	(send)
		$\mid$	$\sum_{i \in I} u?\mathtt{m}_i\langle\alpha_i\rangle(x_i : t_i).P_i$	(receive)
		$\mid$	$P \oplus P$	(conditional process)
		$\mid$	$P \mid P$	(parallel composition)
		$\mid$	X	(process variable)
		$\mid$	$\mathtt{rec}\ X.P$	(recursive process)

ownership invariants. The term **0** denotes the idle process that performs no action. The term $\mathtt{open}(a : T, b : S).P$ denotes a process that creates a *linear channel*, represented as a pair of endpoints a of type T and b of type S, and continues as P. We will say that b is the peer endpoint of a and vice-versa. The term $\mathtt{open}(a : T).P$ denotes a process that creates an *unrestricted channel*, represented as an endpoint a of type T along with an unrestricted pointer $\overline{a}$ of type $\overline{T}$, and continues as P. The term $\mathtt{close}(u)$ denotes a process closing and deallocating the endpoint u. The term $u!\mathtt{m}\langle T\rangle(v).P$ denotes a process that sends a message $\mathtt{m}\langle T\rangle(v)$ on the endpoint u and continues as P. The message is made of a *tag* $\mathtt{m}$ along with its *parameter v*. The endpoint type T instantiates the type variable in the type of u. For consistency with the type language we only consider monadic communications where every message has exactly one type/value parameter. The generalization to polyadic communications, which we will occasionally use in the examples, does not pose substantial problems. The term $\sum_{i \in I} u?\mathtt{m}_i\langle\alpha_i\rangle(x_i : t_i).P_i$ denotes a process that waits for a message from the endpoint u. The tag $\mathtt{m}_i$ of the received message determines the continuation P_i where the variable x_i is instantiated with the parameter of the message. Sometimes we will write $u?\mathtt{m}_1\langle\alpha_1\rangle(x_1 : t_1).P_1 + \cdots + u?\mathtt{m}_n\langle\alpha_n\rangle(x_n : t_n).P_n$ in place of $\sum_{i=1}^{n} u?\mathtt{m}_i\langle\alpha_i\rangle(x_i : t_i).P_i$.[2] The term $P \oplus Q$ denotes a process that internally decides whether to behave as P or as Q. We do not specify the actual condition that determines the decision, as this is irrelevant for our purposes. To improve readability, in some of the examples we will use a more concrete syntax. As usual, terms $\mathtt{rec}\ X.P$ and X serve to denote recursive processes, while $P \mid Q$ denotes the parallel composition of P and Q.

Table 4 collects the definitions of *free names* $\mathtt{fn}(\cdot)$ and *bound names* $\mathtt{bn}(\cdot)$ for processes. Beware that a process $\mathtt{open}(a : T).P$ implicitly binds $\overline{a}$ in addition to a in P. In the same table we also define the sets of *free type variables* $\mathtt{ftv}(\cdot)$ and of *bound type variables* of a process. Note that the set of bound type variables only includes those variables occurring in input prefixes of the process, not the type variables bound within endpoint types occurring in the process. The construct $\mathtt{rec}\ X.P$ is the only binder for process variables. The sets of *free process variables* $\mathtt{fpv}(\cdot)$ and of *bound process variables* $\mathtt{bpv}(\cdot)$ are standard. We identify processes up to renaming of bound names/type variables/process variables and

---

[2]We require the endpoint u to be the same in all branches of the receive construct, while `switch receive` in $\mathsf{Sing}^{\#}$ allows waiting for messages coming from *different* endpoints. This generalization would not affect our formalization in any substantial way, save for slightly more complicated typing rules.

Table 4: Free and bound names/type variables in processes.

$$\mathtt{fn}(\mathbf{0}) = \mathtt{fn}(X) = \emptyset$$
$$\mathtt{fn}(\mathtt{close}(u)) = \{u\}$$
$$\mathtt{fn}(\mathtt{open}(a:T,b:S).P) = \mathtt{fn}(P) \setminus \{a,b\}$$
$$\mathtt{fn}(\mathtt{open}(a:T).P) = \mathtt{fn}(P) \setminus \{a,\overline{a}\}$$
$$\mathtt{fn}(u!\mathtt{m}\langle T\rangle(v).P) = \{u,v\} \cup \mathtt{fn}(P)$$
$$\mathtt{fn}(\textstyle\sum_{i\in I} u?\mathtt{m}_i\langle\alpha_i\rangle(x_i:t_i).P_i) = \{u\} \cup \textstyle\bigcup_{i\in I}(\mathtt{fn}(P_i) \setminus \{x_i\})$$
$$\mathtt{fn}(P \oplus Q) = \mathtt{fn}(P \mid Q) = \mathtt{fn}(P) \cup \mathtt{fn}(Q)$$
$$\mathtt{fn}(\mathtt{rec}\ X.P) = \mathtt{fn}(P)$$

$$\mathtt{bn}(\mathbf{0}) = \mathtt{bn}(\mathtt{close}(u)) = \mathtt{bn}(X) = \emptyset$$
$$\mathtt{bn}(\mathtt{open}(a:T,b:S).P) = \{a,b\} \cup \mathtt{bn}(P)$$
$$\mathtt{bn}(\mathtt{open}(a:T).P) = \{a,\overline{a}\} \cup \mathtt{bn}(P)$$
$$\mathtt{bn}(u!\mathtt{m}\langle T\rangle(v).P) = \mathtt{bn}(\mathtt{rec}\ X.P) = \mathtt{bn}(P)$$
$$\mathtt{bn}(\textstyle\sum_{i\in I} u?\mathtt{m}_i\langle\alpha_i\rangle(x_i:t_i).P_i) = \textstyle\bigcup_{i\in I}(\{x_i\} \cup \mathtt{bn}(P_i))$$
$$\mathtt{bn}(P \oplus Q) = \mathtt{bn}(P \mid Q) = \mathtt{bn}(P) \cup \mathtt{bn}(Q)$$

$$\mathtt{ftv}(\mathbf{0}) = \mathtt{ftv}(\mathtt{close}(u)) = \mathtt{ftv}(X) = \emptyset$$
$$\mathtt{ftv}(\mathtt{open}(a:T,b:S).P) = \mathtt{ftv}(T) \cup \mathtt{ftv}(S) \cup \mathtt{ftv}(P)$$
$$\mathtt{ftv}(\mathtt{open}(a:T).P) = \mathtt{ftv}(u!\mathtt{m}\langle T\rangle(v).P) = \mathtt{ftv}(T) \cup \mathtt{ftv}(P)$$
$$\mathtt{ftv}(\textstyle\sum_{i\in I} u?\mathtt{m}_i\langle\alpha_i\rangle(x_i:t_i).P_i) = \textstyle\bigcup_{i\in I}((\mathtt{ftv}(t_i) \cup \mathtt{ftv}(P_i)) \setminus \{\alpha_i\})$$
$$\mathtt{ftv}(P \oplus Q) = \mathtt{ftv}(P \mid Q) = \mathtt{ftv}(P) \cup \mathtt{ftv}(Q)$$
$$\mathtt{ftv}(\mathtt{rec}\ X.P) = \mathtt{ftv}(P)$$

$$\mathtt{btv}(\mathbf{0}) = \mathtt{btv}(\mathtt{close}(u)) = \mathtt{btv}(X) = \emptyset$$
$$\mathtt{btv}(\mathtt{open}(a:T,b:S).P) = \mathtt{btv}(\mathtt{open}(a:T).P) = \mathtt{btv}(P)$$
$$\mathtt{btv}(u!\mathtt{m}\langle T\rangle(v).P) = \mathtt{btv}(\mathtt{rec}\ X.P) = \mathtt{btv}(P)$$
$$\mathtt{btv}(\textstyle\sum_{i\in I} u?\mathtt{m}_i\langle\alpha_i\rangle(x_i:t_i).P_i) = \textstyle\bigcup_{i\in I}(\{\alpha_i\} \cup \mathtt{btv}(P_i))$$
$$\mathtt{btv}(P \oplus Q) = \mathtt{btv}(P \mid Q) = \mathtt{btv}(P) \cup \mathtt{btv}(Q)$$

let $P\{v/x\}$, $P\{T/\alpha\}$, and $P\{Q/X\}$ denote the standard capture-avoiding substitutions of variables/type variables/process variables with values/endpoint types/processes.

Example 4.1. Let us encode the map function in Figure 1 using the syntax of our process calculus. As anticipated in Example 3.3, the idea is to represent map as a process that permanently accepts invocations and handles them. For this reason we need an endpoint,

Table 5: Syntax of heaps and queues.

Heap	μ	$::=$	$\emptyset$	(empty)	**Queue**	$\mathfrak{Q}$	$::=$	ε	(empty)
		$\mid$	$a \mapsto [a, \mathfrak{Q}]$	(endpoint)			$\mid$	$\mathtt{m}\langle T\rangle(\mathsf{v})$	(message)
		$\mid$	μ, μ	(composition)			$\mid$	$\mathfrak{Q} :: \mathfrak{Q}$	(composition)

say c, to which invocation requests are sent and we define the $\text{MAP}(c)$ process thus:

$$\text{MAP}(c) = \texttt{rec } X.c?\texttt{Invoke}\langle\alpha,\beta\rangle(z : \mathsf{lin}\ \overline{T_{\texttt{map}}(\alpha,\beta)}).(X \mid \text{BODY}(\alpha,\beta,z))$$

$$\begin{aligned}\text{BODY}(\alpha,\beta,z) = \ &z?\texttt{Arg}(\textit{mapper} : \mathsf{lin}\ \overline{T_{\texttt{Mapper}}(\alpha,\beta)}).\\ &z?\texttt{Arg}(\textit{source} : \mathsf{lin}\ \overline{T_{\texttt{Stream}}(\alpha)}).\\ &z?\texttt{Arg}(\textit{target} : \mathsf{lin}\ T_{\texttt{Stream}}(\beta)).\\ &\texttt{rec } Y.(\ \ \textit{source}?\texttt{Data}(x : \mathsf{lin}\ \alpha).\textit{mapper}!\texttt{Arg}(x).\\ &\qquad\qquad \textit{mapper}?\texttt{Res}(y : \mathsf{lin}\ \beta).\textit{target}!\texttt{Data}(y).Y\\ &\qquad + \textit{source}?\texttt{Eos}().\textit{target}!\texttt{Eos}().\\ &\qquad\quad z!\texttt{Arg}(\textit{mapper}).z!\texttt{Res}().\\ &\qquad\quad (\texttt{close}(z) \mid \texttt{close}(\textit{source}) \mid \texttt{close}(\textit{target})))\end{aligned}$$

The process $\text{MAP}(c)$ repeatedly reads $\texttt{Invoke}$-tagged messages from c. Each message carries another endpoint z that represents a private session established between the caller and the callee, whose purpose is to make sure that no interference occurs between independent invocations of the service. Note that z has type $\overline{T_{\texttt{map}}(\alpha,\beta)}$, the dual of $T_{\texttt{map}}(\alpha,\beta)$, since it is the endpoint handed over by the caller from which the callee will *receive* the arguments and *send* the result. The body of the $\texttt{map}$ function is encoded by the $\text{BODY}(\alpha,\beta,z)$ process, which begins by reading the three arguments *mapper*, *source*, and *target*. Then, the process enters its main loop where messages are received from *source*, processed through *mapper*, and finally sent on *target*. Overall the structure of the process closely follows that of the code in Figure 1, where the branch operator is used for modeling the `switch receive` construct. The only remarkable difference occurs after the input of a $\texttt{Eos}$-tagged message, where the *mapper* argument is returned to the caller so as to model the temporary ownership transfer that was implicitly indicated by the lack of the `[Claims]` annotation in $\texttt{map}$. At this point the z endpoint serves no other purpose and is closed along with *source* and *target*. ∎

To state the operational semantics of processes we need a formal definition of the *exchange heap* (or simply *heap*), which is given in Table 5. *Heaps*, ranged over by μ, are term representations of finite maps from pointers to heap objects: the term $\emptyset$ denotes the empty heap, in which no object is allocated; the term $a \mapsto [b, \mathfrak{Q}]$ denotes a heap made of an endpoint located at a. The endpoint is a structure containing another pointer b and a *queue* $\mathfrak{Q}$ 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 $\texttt{dom}(\mu)$ and $\texttt{dom}(\mu')$, 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 $\mathfrak{Q}$, are finite ordered sequences of messages $\mathtt{m}_1\langle T_1\rangle(\mathsf{v}_1) :: \cdots :: \mathtt{m}_n\langle T_n\rangle(\mathsf{v}_n)$, where a message $\mathtt{m}\langle T\rangle(\mathsf{v})$ is identified by its tag $\mathtt{m}$, the endpoint type T with which its type argument has been instantiated, and its value argument v. 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 $::$. The T component in the enqueued messages must be understood as a technical annotation that helps reasoning on the formal properties of the model. In particular, it

Table 6: Structural congruence.

(S-Idle)	(S-Comm)	(S-Assoc)
$P \mid \mathbf{0} \equiv P$	$P \mid Q \equiv Q \mid P$	$P \mid (Q \mid R) \equiv (P \mid Q) \mid R$

Table 7: Operational semantics of processes.

(R-Open Linear Channel)
$$(\mu; \mathtt{open}(a : T, b : S).P) \rightarrow (\mu, a \mapsto [b, \varepsilon], b \mapsto [a, \varepsilon]; P)$$

(R-Open Unrestricted Channel)   (R-Choice Left)   (R-Choice Right)
$(\mu; \mathtt{open}(a : T).P) \rightarrow (\mu, a \mapsto [a, \varepsilon]; P)$ $(\mu; P \oplus Q) \rightarrow (\mu; P)$ $(\mu; P \oplus Q) \rightarrow (\mu; Q)$

(R-Send Linear)
$$(\mu, a \mapsto [b, \mathfrak{Q}], b \mapsto [a, \mathfrak{Q}']; a!\mathtt{m}\langle T \rangle(\mathsf{v}).P) \rightarrow (\mu, a \mapsto [b, \mathfrak{Q}], b \mapsto [a, \mathfrak{Q}' :: \mathtt{m}\langle T \rangle(\mathsf{v})]; P)$$

(R-Send Unrestricted)
$$(\mu, a \mapsto [a, \mathfrak{Q}]; \overline{a}!\mathtt{m}\langle T \rangle(\mathsf{v}).P) \rightarrow (\mu, a \mapsto [a, \mathfrak{Q} :: \mathtt{m}\langle T \rangle(\mathsf{v})]; P)$$

(R-Receive)
$$\frac{k \in I}{(\mu, a \mapsto [b, \mathtt{m}_k\langle T \rangle(\mathsf{v}) :: \mathfrak{Q}]; \sum_{i \in I} a?\mathtt{m}_i\langle \alpha_i \rangle(x_i : t_i).P_i) \rightarrow (\mu, a \mapsto [b, \mathfrak{Q}]; P_k\{T/\alpha_k\}\{\mathsf{v}/x_k\})}$$

(R-Rec)
$$(\mu; \mathtt{rec}\ X.P) \rightarrow (\mu; P\{\mathtt{rec}\ X.P/X\})$$

(R-Par)   (R-Struct)
$$\frac{(\mu; P) \rightarrow (\mu'; P')}{(\mu; P \mid Q) \rightarrow (\mu'; P' \mid Q)} \qquad \frac{P \equiv P' \quad (\mu; P') \rightarrow (\mu'; Q') \quad Q' \equiv Q}{(\mu; P) \rightarrow (\mu'; Q)}$$

does not imply that a practical implementation of the calculus must necessarily provide a runtime representation of endpoint types.[3]

We define the operational semantics of processes as the combination of a structural congruence relation, which equates processes we do not want to distinguish, and a reduction relation. Structural congruence, denoted by $\equiv$, is the least congruence relation defined by the axioms in Table 6 and closed under parallel composition. Essentially, the axioms state that $\mid$ is commutative, associative, and has $\mathbf{0}$ as neutral element.

Processes communicate by means of endpoints that are allocated on the heap. Consequently, the reduction relation defines the transitions of *systems* rather than of processes, where a system is a pair $(\mu; P)$ of a heap μ and a process P. The reduction relation $\rightarrow$ is inductively defined in Table 7; we comment on the rules in the following paragraphs. Rule (R-Open Linear Channel) creates a new linear channel, which consists of two fresh endpoints with empty queues and mutually referring to each other. The mutual references are needed since the messages sent using one of the endpoints will be enqueued into the

---

[3] Sing$^\#$ *does* require a runtime representation of endpoint types because its expression language is equipped with a dynamic cast operator.

other peer. Rule (R-Open Unrestricted Channel) creates a new unrestricted channel, which consists of a single endpoint with empty queue. The reference in the endpoint is initialized with a pointer to itself. This way, by inspecting the b component of an endpoint $a \mapsto [b, \mathfrak{Q}]$ it is possible to understand whether the endpoint belongs to a linear or to an unrestricted channel, as we respectively have either $a \neq b$ or $a = b$. This distinction is necessary in the reductions defining the semantics of outputs, as we will see shortly. In both (R-Open Linear Channel) and (R-Open Unrestricted Channel) we implicitly rename bound names to make sure that the newly introduced pointers do not already occur in $\mathtt{dom}(\mu)$, for otherwise the heap in the resulting system would be undefined. Rules (R-Choice Left) and (R-Choice Right) describe the standard reduction of conditional processes. Rules (R-Send Linear) and (R-Send Unrestricted) describe the output of a message $\mathtt{m}\langle T \rangle(\mathtt{v})$ on the endpoint a of a linear channel and on the endpoint $\overline{a}$ of an unrestricted channel, respectively. In the former case, the message is enqueued at the end of a's peer endpoint queue. In the latter case, the message is enqueued in the only available queue. Rule (R-Receive) describes the input of 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 type and value arguments instantiate the type variable α_k and variable 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. Rule (R-Rec) describes the usual unfolding of a recursive process. Rule (R-Par) closes reductions under parallel composition. Observe that the heap is treated globally, even when it is only a sub-process to reduce. Finally, rule (R-Struct) describes reductions modulo structural congruence. There is no reduction for `close`(a) processes. In principle, `close`(a) should deallocate the endpoint located at a and remove the association for a from the heap. In the formal model it is technically convenient to treat `close`(a) processes as persistent because, in this way, we keep track of the pointers that have been properly deallocated. We will see that this information is crucial in the definition of well-behaved processes (Definition 4.2). A process willing to deallocate a pointer a and to continue as P afterwards can be modeled as `close`$(a) \mid P$. 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 characterize well-behaved systems as those that are free from faults, leaks, and communication errors: a *fault* is an attempt to use a pointer not corresponding to an allocated object or to use a pointer in some way which is not allowed by the object it refers to; a *leak* is a region of the heap that some process allocates and that becomes unreachable because no reference to it is directly or indirectly available to the processes in the system; a *communication error* occurs if some process receives a message of unexpected type. We conclude this section formalizing these properties. To do so, we need to define the reachability of a heap object with respect to a set of *root* pointers. Intuitively, a process P may directly reach any object located at some pointer in the set $\mathtt{fn}(P)$ (we can think of the pointers in $\mathtt{fn}(P)$ as of the local variables of the process stored on its stack); from these pointers, the process may reach other heap objects by reading messages from the endpoints it can reach, and so forth.

Definition 4.1 (reachable pointers)**.** We say that c is *reachable* from a in μ, notation $c \prec_\mu a$, if $a \mapsto [b, \mathfrak{Q} :: \mathtt{m}\langle T \rangle(c) :: \mathfrak{Q}'] \in \mu$. We write $\preccurlyeq_\mu$ for the reflexive, transitive closure of $\prec_\mu$. Let $\mathtt{reach}(A, \mu) = \{c \in \mathtt{Pointers} \mid \exists a \in A : c \preccurlyeq_\mu a\}$.

Observe that $\mathtt{reach}(A, \mu) \subseteq \mathtt{Pointers}$ for every $A \subseteq \mathtt{Pointers} \cup \overline{\mathtt{Pointers}}$ and μ. Also, according to this definition nothing is reachable from an unrestricted pointer. The rationale is that we will use $\mathtt{reach}(\cdot, \cdot)$ only to define the ownership invariant, for which the only pointers that matter are the linear ones. We now define well-behaved systems formally.

Definition 4.2 (well-behaved process)**.** We say that P is *well behaved* if $(\emptyset; P) \Rightarrow (\mu; Q)$ implies:

(1) $\mathtt{dom}(\mu) = \mathtt{reach}(\mathtt{fn}(Q), \mu)$;
(2) $Q \equiv P_1 \mid P_2$ implies $\mathtt{reach}(\mathtt{fn}(P_1), \mu) \cap \mathtt{reach}(\mathtt{fn}(P_2), \mu) = \emptyset$;
(3) $Q \equiv P_1 \mid P_2$ and $(\mu; P_1) \not\rightarrow$ where P_1 does not have unguarded parallel compositions imply either $P_1 = \mathbf{0}$ or $P_1 = \mathtt{close}(a)$ or $P_1 = \sum_{i \in I} a?\mathtt{m}_i\langle\alpha_i\rangle(x_i : t_i).P_i$ and, in the last two cases, $a \mapsto [b, \varepsilon] \in \mu$.

In words, a process P is well behaved if every residual of P reachable from a configuration where the heap is empty satisfies a number of conditions. Conditions (1) and (2) require the absence of faults and leaks. Indeed, 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 linear pointer is reachable from two or more distinct processes. Because of the definition of reachable pointers, though, it may be possible that two or more processes share the same unrestricted pointer. Since processes of the form $\mathtt{close}(a)$ are persistent, this condition also requires the absence of faults deriving from multiple deallocations of the same endpoint or from the use of deallocated endpoints. Condition (3) requires the absence of communication errors, namely that if $(\mu; Q)$ is stuck (no reduction is possible), then it is because every non-terminated process in Q is waiting for a message on an endpoint having an empty queue. This configuration corresponds to a genuine deadlock where every process in some set is waiting for a message that is to be sent by another process in the same set. Condition (3) also ensures the absence of so-called *orphan messages*: no message accumulates in the queue of closed endpoints. We only consider initial configurations with an empty heap for two reasons: first, we take the point of view that initially there are no allocated objects; second, since we will need a well-typed predicate for heaps and we do not want to verify heap well-typedness at runtime, we will make sure that the empty heap is trivially well typed.

We conclude this section with a few examples of ill-behaved processes to illustrate the sort of errors we aim to avoid with our static type system:

- The process $\mathtt{open}(a : T, b : S).\mathbf{0}$ violates condition (1), since it allocates two endpoints a and b and forgets them, thus generating a leak.
- The process $\mathtt{open}(a : T, b : S).(\mathtt{close}(a) \mid \mathtt{close}(a) \mid \mathtt{close}(b))$ violates condition (2), since it deallocates the same endpoint a twice. This is an example of fault.
- The process $\mathtt{open}(a : T, b : S).(a!\mathtt{m}().\mathtt{close}(a) \mid b?\mathtt{m}'().\mathtt{close}(b))$ violates condition (3), since it reduces to a parallel composition of subprocesses where one has sent an $\mathtt{m}$-tagged message, but the other one was expecting an $\mathtt{m}'$-tagged message.
- The process $\mathtt{open}(a : T, b : S).\overline{a}!\mathtt{m}().b?\mathtt{m}().(\mathtt{close}(a) \mid \mathtt{close}(b))$ violates condition (3), since it reduces to a stuck process that attempts at sending an $\mathtt{m}$-tagged message using the unrestricted pointer $\overline{a}$, while in fact a is a linear pointer.

5. Type System

5.1. **Weighing Types.** We aim at defining a type system such that well-typed processes are well behaved. In session type systems, from which we draw inspiration, each action performed by a process using a certain endpoint must be matched by a corresponding action in the type associated with the endpoint, and the continuation process after that action must behave according the continuation in the endpoint type. Following this intuition, the reader may verify that the process BODY (Example 4.1) uses the endpoint z correctly with respect to the endpoint type $\overline{T_{\mathtt{map}}(\alpha, \beta)}$ (Example 3.3). Analogous observations can be made for the other endpoints (*mapper*, *source*, *target*) received from z and subsequently used in BODY. Linearity makes sure that a process owning an endpoint *must* use the endpoint (according to its type), or it must delegate it to another process. Endpoints cannot be simply forgotten and this is essential in guaranteeing the absence of leaks. In Example 4.1 there is a number of endpoints involved: c is owned permanently by MAP; z is owned by BODY until an Eos-tagged message is received, at which point it is deallocated; *source* and *target* are acquired by BODY and deallocated when no longer in use; finally, *mapper* is acquired by BODY from the caller and returned to the caller when BODY ends. Overall, MAP is evenly balanced as far as the ownership of linear endpoints is concerned.

Nonetheless, as we have anticipated in Section 2, there are apparently well-typed processes that lead to a violation of the ownership invariant. A first example is the process

$$P = \mathtt{open}(a : T_1, b : T_2).a!\mathtt{m}(b).\mathtt{close}(a) \tag{5.1}$$

where

$$T_1 = !\mathtt{m}(\mathsf{lin}\ T_2).\mathsf{end} \qquad \text{and} \qquad T_2 = \mathtt{rec}\ \alpha.?\mathtt{m}(\mathsf{lin}\ \alpha).\mathsf{end}\ .$$

The process P begins by creating two endpoints a and b with dual endpoint types. The fact that $T_1 = \overline{T_2}$ ensures the absence of communication errors, as each action performed on one endpoint is matched by a corresponding co-action performed on the corresponding peer. After its creation, endpoint b is sent over endpoint a. Observe that, according to T_1, the process is entitled to send an m-tagged message with argument of type T_2 on a and b has precisely that type. After the output operation, the process no longer owns endpoint b and endpoint a is deallocated. Apparently, P behaves correctly while in fact it generates a leak, as we can see from its reduction:

$$(\emptyset; P) \to (a \mapsto [b, \varepsilon], b \mapsto [a, \varepsilon]; a!\mathtt{m}(b).\mathtt{close}(a)) \to (a \mapsto [b, \varepsilon], b \mapsto [a, \mathtt{m}(b)]; \mathtt{close}(a))$$

In the final, stable configuration we have $\mathtt{reach}(\mathtt{fn}(\mathtt{close}(a)), \mu) = \mathtt{reach}(\{a\}, \mu) = \{a\}$ (recall that b is not reachable from a even though its peer is) while $\mathtt{dom}(\mu) = \{a, b\}$. In particular, the endpoint b is no longer reachable and this configuration violates condition (1) of Definition 4.2. Additionally, if there were some 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 compromise the typing of b: the endpoint type associated with b is T_2, but as we remove the message from its queue it turns to end. The b in the message, however, would retain the now obsolete type T_2, 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 $b \mapsto [a, \mathtt{m}(b)]$ region of the heap needs not be owned by any process because it "owns itself". With respect to other type systems for session types, we must tighten our typing rules 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, a and b might have been acquired in previous communications (think of the `foo` and `bar` functions in Section 2, where nothing is known about the arguments `e` and `f` save for their type) and they may not even be peers. For example, the process

$$\texttt{open}(a : T_1, c : T_2).\texttt{open}(b : T_1, d : T_2).a!\texttt{m}(d).b!\texttt{m}(c).(\texttt{close}(a) \mid \texttt{close}(b))$$

creates a leak with a cycle of length 2 even though no endpoint is ever sent over its own peer.

Our approach for attacking the problem stems from the observation that infinite values (once the leak configuration has been reached the endpoint b above fits well in this category) usually inhabit recursive types and the endpoint type T_2 indeed exhibits an odd form of recursion, as the recursion variable α occurs within the only prefix of T_2. Forbidding this form of recursion in general, however, would (1) unnecessarily restrict our language and (2) it would not protect us completely against leaks. Regarding (1), we can argue that an endpoint type $T_2' = \texttt{rec}\ \alpha.!\texttt{m}(\alpha).\texttt{end}$ (which begins with an output action) would never allow the creation of cycles in the heap despite its odd recursion. The reason is that, if we are sending an endpoint $b : T_2'$ over $a : T_2'$, then the peer of a must have the dual type $\overline{T_2'} = ?\texttt{m}(T_2').\texttt{end}$ (which begins with an input action) and therefore must be different from b. Regarding (2), consider the following variation of the process P above

$$Q = \texttt{open}(a : S_1, b : S_2).a!\texttt{m}\langle S_2 \rangle(b).\texttt{close}(a) \tag{5.2}$$

where

$$S_1 = !\texttt{m}\langle\alpha\rangle(\textsf{lin}\ \alpha).\texttt{end} \qquad \text{and} \qquad S_2 = ?\texttt{m}\langle\alpha\rangle(\textsf{lin}\ \alpha).\texttt{end}\ .$$

Once again, S_1 and S_2 are dual endpoint types and process Q behaves correctly with respect to them. Notice that neither S_1 nor S_2 is recursive, and yet Q yields the same kind of leak that we have observed in the reduction of P.

What do T_2 and S_2 have in common that T_2' and S_1 do not and that makes them dangerous? First of all, both T_2 and S_2 begin with an input action so they denote endpoints in a *receive state*, and only endpoints in a receive state can have a non-empty queue. Second, the type of the arguments in T_2 and S_2 *may* denote other endpoints with a non-empty queue: in T_2 this is evident as the type of the argument is T_2 itself; in S_2 the type of the argument is the existentially quantified type variable α, which can be instantiated with *any* endpoint type and, in particular, with an endpoint type beginning with an input action. If we think of the chain of pointers originating from the queue of an endpoint, we see that both T_2 and S_2 allow for chains of arbitrary length and the leak originates when this chain becomes in fact infinite, meaning that a cycle has formed in the heap. Our idea to avoid these cycles uses the fact that it is possible to compute, for each endpoint type, a value in the set $\mathbb{N} \cup \{\infty\}$, that we call *weight*, representing the upper bound of the length of any chain of pointers originating from the queue of the endpoints it denotes. A weight equal to ∞ means that there is no such upper bound. Then, the idea is to restrict the type system so that:

> Only endpoints having a finite-weight type can be sent as messages.

A major issue in defining the weight of types is how to deal with type variables. If type variables can be instantiated with arbitrary endpoint types, hence with endpoint types having arbitrary weight, the weight of type variables cannot be estimated to be finite. At the same time, assigning an infinite weight to *every* type variable can be overly restrictive.

To see why, consider the following fragment of the MAP process defined in Example 4.1:

$$[\cdots].source?\mathtt{Data}(x : \mathsf{lin}\ \alpha).mapper!\mathtt{Arg}(x).[\cdots]$$

The process performs an output operation $mapper!\mathtt{Arg}(x)$ which, according to our idea, would be allowed only if the type of argument x had a finite weight. It turns out that x has type $\mathsf{lin}\ \alpha$ and is bound by the preceding input action $source?\mathtt{Data}(x : \mathsf{lin}\ \alpha)$. If we estimate the weight to α to be infinite, a simple process like MAP would be rejected by our type system. By looking at the process more carefully one realizes that, since x has been received from a message, its actual type *must be* finite-weight, for otherwise the sender (the process using $source$'s peer endpoint) would have been rejected by the type system. In general, since type variables denote values that can only be passed around and these must have a finite-weight type, it makes sense to impose a further restriction:

> Only finite-weight endpoint types can instantiate type variables.

Then, in computing the weight of a type, we should treat its free and bound type variables differently: free type variables are placeholders for a finite-weight endpoint type and are given a finite weight; bound type variables are yet to be instantiated with some unknown endpoint type of arbitrary weight and therefore their weight cannot be estimated to be finite. We will thus define the weight $\|t\|_\Delta$ of a type t with respect to a set Δ of free type variables:

Definition 5.1 (type weight). We say that $\mathscr{W}$ is a *coinductive weight bound* if $(\Delta, T, n) \in \mathscr{W}$ implies either:

- $T = \mathsf{end}$, or
- $T = \alpha \in \Delta$, or
- $T = \{!\mathtt{m}_i\langle\alpha_i\rangle(t_i).T_i\}_{i\in I}$, or
- $T = \{?\mathtt{m}_i\langle\alpha_i\rangle(q_i\ S_i).T_i\}_{i\in I}$ and $n > 0$ and $\alpha_i \notin \Delta$ and $(\Delta, S_i, n-1) \in \mathscr{W}$ and $(\Delta, T_i, n) \in \mathscr{W}$ for every $i \in I$.

We write $\Delta \vdash T :: n$ if $(\Delta, T, n) \in \mathscr{W}$ for some coinductive weight bound $\mathscr{W}$. The *weight* of an endpoint type T with respect to Δ, denoted by $\|T\|_\Delta$, is defined by $\|T\|_\Delta = \min\{n \in \mathbb{N} \mid \Delta \vdash T :: n\}$ where we let $\min \emptyset = \infty$. We simply write $\|T\|$ in place of $\|T\|_\emptyset$ and we extend weights to types so that $\|q\ T\| = \|T\|$. When comparing weights we extend the usual total orders $<$ and $\leq$ over natural numbers so that $n < \infty$ for every $n \in \mathbb{N}$ and $\infty \leq \infty$.

The weight of t is defined as the least of its weight bounds, or ∞ if there is no such weight bound. A few weights are straightforward to compute, for example we have $\|\mathsf{end}\| = \|\{!\mathtt{m}_i\langle\alpha_i\rangle(t_i).T_i\}_{i\in I}\| = 0$. Indeed, the queues of endpoints with type end and those in a send state are empty and therefore the chains of pointers originating from them has zero length. A type variable α can have a finite or infinite weight depending on whether it occurs free or bound. So we have $\|\alpha\|_{\{\alpha\}} = 0$ and $\|\alpha\| = \infty$. Note that $\|\alpha\|_{\{\alpha\}} = 0$ although α may be actually instantiated with a type that has a strictly positive, but finite weight. Endpoint types in a receive state have a strictly positive weight. For instance we have $\|?\mathtt{m}(\mathsf{end}).\mathsf{end}\| = 1$ and $\|?\mathtt{m}(?\mathtt{m}(\mathsf{end}).\mathsf{end}).\mathsf{end}\| = 2$. If we go back to the examples of endpoint types that we used to motivate this discussion, we have $\|T_2'\| = \|S_1\| = 0$ and $\|T_2\| = \|S_2\| = \infty$, from which we deduce that endpoints with type T_2' or S_1 are safe to be sent as messages, while endpoints with type T_2 or S_2 are not.

Before we move on to illustrating the type system, we must discuss one last issue that has to do with subtyping. Any type system with subtyping normally allows to use a value

having type t where a value having type s with $t \leqslant s$ is expected. For example, in the MAP process we have silently made the assumption that the value x received with the Data-tagged message had *exactly* the (finite-weight) type with which α has been instantiated while in fact x might have a *smaller* type. Therefore, the restrictions we have designed work provided that, if $t \leqslant s$ and s is finite-weight, then t is finite-weight as well. This is indeed the case, and in fact we can express an even stronger correspondence between weights and subtyping:

Proposition 5.1. $t \leqslant s$ *implies* $\|t\|_\Delta \leq \|s\|_\Delta$.

Proof. It is easy to show that $\mathscr{W} = \{(\Delta, T, n) \mid \exists S : \Delta \vdash T \leqslant S \,\&\, \Delta \vdash S :: n\} \cup \{(\Delta, t, n) \mid \exists s : \Delta \vdash t \leqslant s \,\&\, \Delta \vdash s :: n\}$ is a coinductive weight bound. $\square$

5.2. **Typing the Heap.** The heap plays a primary role because inter-process communication utterly relies on heap-allocated structures; also, most properties of well-behaved processes are direct consequences of related properties of the heap. Therefore, just as we will check well typedness of a process P with respect to a type environment that associates the pointers occurring in P with the corresponding types, we will also need to check that the heap is consistent with respect to the same environment. This leads to a notion of well-typed heap that we develop in this section. The mere fact that we have this notion does not mean that we need to type-check the heap at runtime, because well-typed processes will only create well-typed heaps and the empty heap will be trivially well typed. We shall express well-typedness of a heap μ with respect to a pair $\Gamma_0; \Gamma$ of type environments where Γ contains the type of unrestricted pointers and the type of the *roots* of μ (the pointers that are not referenced by any other structure allocated on the heap), while Γ_0 contains the type of the pointers to allocated structures that are reachable from the roots of μ.

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

$$\mathtt{tail}(T, \mathtt{m}\langle S\rangle(s)) = T'$$

indicates that a message with tag $\mathtt{m}$, type argument S, and argument of type s can be received from an endpoint with type T which can be used according to type T' thereafter. The function is defined by the rule:

$$\frac{k \in I \qquad s \leqslant t_k\{S/\alpha_k\}}{\mathtt{tail}(\{?\mathtt{m}_i\langle\alpha_i\rangle(t_i).T_i\}_{i\in I}, \mathtt{m}_k\langle S\rangle(s)) = T_k\{S/\alpha_k\}}$$

Note that $\mathtt{tail}(T, \mathtt{m}\langle S\rangle(s))$ is undefined when $T = \mathtt{end}$ or T is an internal choice. This is consistent with the observation that it is not possible to receive messages from endpoints having these types. We extend $\mathtt{tail}(\cdot, \cdot)$ to possibly empty sequences of message specifications thus:

$$\mathtt{tail}(T, \varepsilon) = T$$
$$\mathtt{tail}(T, \mathtt{m}_1\langle S_1\rangle(s_1) \cdots \mathtt{m}_n\langle S_n\rangle(s_n)) = \mathtt{tail}(\mathtt{tail}(T, \mathtt{m}_1\langle S_1\rangle(s_1)), \mathtt{m}_2\langle S_2\rangle(s_2) \cdots \mathtt{m}_n\langle S_n\rangle(s_n))$$

We now have all the notions to express the well-typedness of a heap μ with respect to a pair $\Gamma_0; \Gamma$ of type environments. A *type environment* is a finite map $\Gamma = \{u_i : q_i\ T_i\}_{i \in I}$ from names to types. We adopt the following notation regarding type environments:

- We write $\mathtt{dom}(\Gamma)$ for the domain of Γ, namely the set $\{u_i \mid i \in I\}$;
- we write Γ, Γ' for the union of Γ and Γ' when $\mathtt{dom}(\Gamma) \cap \mathtt{dom}(\Gamma') = \emptyset$;
- we write $q(\Gamma)$ if $q = q_i$ for every $i \in I$ and we say that Γ is *linear* if $\mathsf{lin}(\Gamma)$ and *unrestricted* if $\mathsf{un}(\Gamma)$;
- we define the *q-restriction* of Γ as $\Gamma|_q = \{u_i : q\ T_i \mid i \in I\ \&\ q_i = q\}$;
- finally, we write $\Gamma \vdash u : t$ if $\Gamma(u) = t$.

Definition 5.2 (well-typed heap). Let $\mathsf{lin}(\Gamma_0)$ and $\mathtt{dom}(\Gamma_0) \cap \mathtt{dom}(\Gamma) = \emptyset$ where every endpoint type in Γ_0, Γ is well formed. We write $\Gamma_0; \Gamma \vdash \mu$ if all of the following conditions hold:

(1) For every $a \mapsto [b, \mathfrak{Q}] \in \mu$ we have $b \mapsto [a, \mathfrak{Q}'] \in \mu$ and either $a = b$ or $\mathfrak{Q} = \varepsilon$ or $\mathfrak{Q}' = \varepsilon$.
(2) For every $a \mapsto [b, \varepsilon] \in \mu$ and $b \mapsto [a, \mathtt{m}_1\langle S_1\rangle(\mathsf{v}_1) :: \cdots :: \mathtt{m}_n\langle S_n\rangle(\mathsf{v}_n)] \in \mu$ with $a \neq b$ we have
$$\overline{T} = \mathtt{tail}(S, \mathtt{m}_1\langle S_1\rangle(s_1) \cdots \mathtt{m}_n\langle S_n\rangle(s_n))$$
where $\Gamma_0, \Gamma \vdash a : \mathsf{lin}\ T$ and $\Gamma_0, \Gamma \vdash b : \mathsf{lin}\ S$ and $\Gamma_0, \Gamma \vdash \mathsf{v}_i : s_i$ and $\max\{\|S_i\|, \|s_i\|\} < \infty$ for $1 \leq i \leq n$.
(3) For every $a \mapsto [a, \mathtt{m}_1\langle S_1\rangle(\mathsf{v}_1) :: \cdots :: \mathtt{m}_n\langle S_n\rangle(\mathsf{v}_n)] \in \mu$ we have
$$\overline{T} = \mathtt{tail}(S, \mathtt{m}_1\langle S_1\rangle(s_1) \cdots \mathtt{m}_n\langle S_n\rangle(s_n))$$
where $\Gamma_0, \Gamma \vdash \overline{a} : \mathsf{un}\ T$ and $\Gamma_0, \Gamma \vdash a : \mathsf{lin}\ S$ and $\Gamma_0, \Gamma \vdash \mathsf{v}_i : s_i$ and $\max\{\|S_i\|, \|s_i\|\} < \infty$ for $1 \leq i \leq n$.
(4) $\mathtt{dom}(\mu) = \mathtt{dom}(\Gamma_0, \Gamma|_{\mathsf{lin}}) = \mathtt{reach}(\mathtt{dom}(\Gamma), \mu)$;
(5) $\mathtt{reach}(\{a\}, \mu) \cap \mathtt{reach}(\{b\}, \mu) = \emptyset$ for every $a, b \in \mathtt{dom}(\Gamma)$ with $a \neq b$.

Condition (1) requires that in a well-typed heap every endpoint comes along with its peer and that at least one of the queues of peer endpoints be empty. This invariant is ensured by duality, since a well-typed process cannot send messages on an endpoint until it has read all the pending messages from the corresponding queue. Condition (2) requires that the endpoint types of peer endpoints are dual. More precisely, for every endpoint a with an empty queue, the dual $\overline{T}$ of its type coincides with the residual $\mathtt{tail}(S, \mathtt{m}_1\langle S_1\rangle(s_1) \cdots \mathtt{m}_n\langle S_n\rangle(s_n))$ of the peer's type S. Additionally, every S_i and s_i has finite weight. Condition (3) is similar to condition (2), but deals with unrestricted endpoints. The only difference is that a has no peer endpoint, and the (unrestricted) dual endpoint type is associated instead with $\overline{a}$. Condition (4) states that the type environment Γ_0, Γ must specify a type for all of the allocated objects in the heap and, in addition, every object (located at) a in the heap must be reachable from a root $b \in \mathtt{dom}(\Gamma)$. Finally, condition (5) requires the uniqueness of the root for every allocated object. Overall, since the roots will be distributed linearly to the processes of the system, conditions (4) and (5) guarantee the ownership invariant, namely that every allocated object belongs to one and only one process.

5.3. **Typing Processes.** First of all we define an operation on type environments to add new associations:

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

Table 8: Typing rules for processes.

$$
\begin{array}{cc}
\text{(T-Idle)} & \text{(T-Close)} \\[4pt]
\dfrac{\mathsf{un}(\Gamma)}{\Sigma;\Delta;\Gamma \vdash \mathbf{0}} & \dfrac{\mathsf{un}(\Gamma)}{\Sigma;\Delta;\Gamma, u : \mathsf{lin}\ \mathsf{end} \vdash \mathtt{close}(u)}
\end{array}
$$

$$
\begin{array}{cc}
\text{(T-Open Linear Channel)} & \text{(T-Open Unrestricted Channel)} \\[4pt]
\dfrac{\Delta \Vdash T \qquad \Sigma;\Delta;\Gamma, a : \mathsf{lin}\ T, b : \mathsf{lin}\ \overline{T} \vdash P}{\Sigma;\Delta;\Gamma \vdash \mathtt{open}(a : T, b : \overline{T}).P} & \dfrac{\Delta \Vdash T \qquad \Sigma;\Delta;\Gamma, a : \mathsf{lin}\ T, \overline{a} : \mathsf{un}\ \overline{T} \vdash P}{\Sigma;\Delta;\Gamma \vdash \mathtt{open}(a : T).P}
\end{array}
$$

$$
\text{(T-Send)}
$$
$$
\dfrac{\Delta \Vdash S}{k \in I \qquad s \leqslant t_k\{S/\alpha_k\} \qquad \max\{\|S\|_\Delta, \|s\|_\Delta\} < \infty \qquad \Sigma;\Delta;\Gamma, u : q\ T_k\{S/\alpha_k\} \vdash P}{\Sigma;\Delta;(\Gamma, u : q\ \{!\mathtt{m}_i\langle\alpha_i\rangle(t_i).T_i\}_{i\in I}) + v : s \vdash u!\mathtt{m}_k\langle S\rangle(v).P}
$$

$$
\text{(T-Receive)}
$$
$$
\dfrac{\Delta \Vdash t_i\ ^{(i\in I)} \qquad s_i \leqslant t_i\ ^{(i\in I)} \qquad \Sigma;\Delta,\alpha_i;\Gamma, u : \mathsf{lin}\ T_i, x_i : t_i \vdash P_i\ ^{(i\in I)}}{\Sigma;\Delta;\Gamma, u : \mathsf{lin}\ \{?\mathtt{m}_i\langle\alpha_i\rangle(s_i).T_i\}_{i\in I} \vdash \sum_{i\in I\cup J} u?\mathtt{m}_i\langle\alpha_i\rangle(x_i : t_i).P_i}
$$

$$
\begin{array}{cc}
\text{(T-Choice)} & \text{(T-Par)} \\[4pt]
\dfrac{\Sigma;\Delta;\Gamma \vdash P \qquad \Sigma;\Delta;\Gamma \vdash Q}{\Sigma;\Delta;\Gamma \vdash P \oplus Q} & \dfrac{\Sigma;\Delta;\Gamma_1 \vdash P \qquad \Sigma;\Delta;\Gamma_2 \vdash Q}{\Sigma;\Delta;\Gamma_1 + \Gamma_2 \vdash P \mid Q}
\end{array}
$$

$$
\begin{array}{cc}
\text{(T-Rec)} & \text{(T-Var)} \\[4pt]
\dfrac{\Sigma, \{X \mapsto (\Delta;\Gamma)\};\Delta;\Gamma \vdash P \qquad \mathtt{dom}(\Gamma|_{\mathsf{lin}}) \subseteq \mathtt{fn}(P)}{\Sigma;\Delta;\Gamma \vdash \mathtt{rec}\ X.P} & \dfrac{\mathsf{un}(\Gamma')}{\Sigma, \{X \mapsto (\Delta;\Gamma)\};\Delta,\Delta';\Gamma,\Gamma' \vdash X}
\end{array}
$$

In plain words, an association $u : t$ where t is linear can be added to Γ only if u does not already occur in Γ. An association $u : t$ where t is unrestricted can be added to Γ in two cases: either u does not occur in Γ, in which case the association is simply added, or the *same* association already occurs in Γ, in which case the operation has no effect on the environment. In all the other cases the result is undefined. We generalize $+$ to pairs of arbitrary environments $\Gamma + \Gamma'$ in the natural way.

The typing rules for processes are inductively defined in Table 8. Judgments have the form $\Sigma;\Delta;\Gamma \vdash P$ and state that process P is well typed under the specified environments. The additional environment Σ is a map from process variables to pairs $(\Delta;\Gamma)$ and is used for typing recursive processes. We describe the typing rules in the following paragraphs:

- Rule (T-Idle) states that the idle process is well typed in every unrestricted type environment. Since we impose a correspondence between the free names of a process and the roots of the heap, this rule states that the terminated process has no leaks.
- Rule (T-Close) states that a process $\mathtt{close}(u)$ is well typed provided that u corresponds to an endpoint with type end, on which no further interaction is possible. Also, the remaining type environment must be unrestricted.
- Rule (T-Open Linear Channel) deals with the creation of a new linear channel, which is visible in the continuation process as two peer endpoints typed by dual endpoint types.

The premise $\Delta \Vdash T$ requires T to be well formed with respect to the type variables in Δ. In addition, the rule implicitly requires that no type variable, not even those in Δ, can occur at the top level in T, for otherwise its dual $\overline{T}$ would be undefined.

- Rule (T-OPEN UNRESTRICTED CHANNEL) deals with the creation of a new unrestricted channel, which is accessible in the continuation process by means of two names: a is the linear pointer used for receiving messages while $\overline{a}$ is the unrestricted pointer used for sending messages. Note that $\overline{T}$ is qualified by 'un', therefore it must be $\overline{T} = \{!\mathtt{m}_i \langle \alpha_i \rangle (t_i).\overline{T}\}_{i \in I}$ and $T = \{?\mathtt{m}_i \langle \alpha_i \rangle (t_i).T\}_{i \in I}$.

- Rule (T-SEND) states that a process $u!\mathtt{m}\langle S \rangle(v).P$ is well typed if u (which can be either linear or unrestricted according to q) is associated with an endpoint type T that permits the output of $\mathtt{m}$-tagged messages (second premise). The endpoint type S instantiates the type argument of the message, while the type of the argument v must be a subtype of the expected type in the endpoint type where α has been instantiated with S (third premise). Both S and s must be finite-weight (fourth premise). Since the peer of u must be able to accept a message with an argument of type s, its weight will be strictly larger than that of s. This is to make sure that the the output operation does not create any cycle in the heap. Observe that the weights of S and s are computed with respect to the environment Δ, containing all the free type variables that can possibly occur in S and s. Finally, the continuation P must be well typed in a suitable type environment where the endpoint u is typed according to a properly instantiated continuation of T (fifth premise). Beware of the use of $+$ in the type environments of the rule: if s is linear, then v is no longer accessible in the continuation P; if s is unrestricted, then v may or may not be available in P depending on whether P uses v again or not. Note also that every endpoint type occurring in the process is verified to be well formed with respect to Δ (first premise).

- Rule (T-RECEIVE) deals with inputs: a process waiting for a message from an endpoint $u : q\ T$ is well typed if it can deal with at least all of the message tags in the topmost inputs of T. The continuation processes may use the endpoint u according to the endpoint type T_i and can access the message argument x_i. The context Δ is enriched with the type variable α_i denoting the fact that P_i does not know the exact type with which α_i has been instantiated. Like for the previous typing rule, there is an explicit premise demanding well-formedness of the types occurring in the process.

- Rules (T-CHOICE) and (T-PAR) are standard. In the latter, the type environment is split into two environments to type the processes being composed. According to the definition of $+$, Γ_1 and Γ_2 can only share associations with unrestricted types and, if they do, the associations in Γ_1 and in Γ_2 for the same name must be equal.

- Rule (T-REC) is a nearly standard rule for recursive processes, except for the premise $\mathtt{dom}(\Gamma|_{\mathsf{lin}}) \subseteq \mathtt{fn}(P)$ that enforces a weak form of contractivity in processes. It states that $\mathtt{rec}\ X.P$ is well typed under Γ only if P actually uses the linear names in $\mathtt{dom}(\Gamma)$. Normally, divergent processes such as $\mathtt{rec}\ X.X$ are well typed in every type environment. If this were the case, however, the process $\mathtt{open}(a : T, b : \overline{T}).\mathtt{rec}\ X.X$, which leaks a and b, would be well typed.

- We conclude with the familiar rule (T-VAR) that deals with recursion variables. The rule takes into account the possibility that new type variables and (unrestricted) associations have accumulated in Δ and Γ since the binding of X.

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

Definition 5.3 (well-typed system)**.** We write $\Gamma_0; \Gamma \vdash (\mu; P)$ if $\Gamma_0; \Gamma \vdash \mu$ and $\Gamma \vdash P$.

Let us present the two main results about our framework: well-typedness is preserved by reduction, and well-typed processes are well behaved. Subject reduction takes into account the possibility that types in the environment may change as the process reduces, which is common in behavioral type theories.

Theorem 5.1 (subject reduction). *Let* $\Gamma_0; \Gamma \vdash (\mu; P)$ *and* $(\mu; P) \rightarrow (\mu'; P')$. *Then* $\Gamma_0'; \Gamma' \vdash (\mu'; P')$ *for some* Γ_0' *and* Γ'.

Theorem 5.2 (safety). *Let* $\vdash P$. *Then* P *is well behaved.*

5.4. **Examples.** We conclude this section with a few extended examples: the first one is meant to show a typing derivation; the second one presents a scenario in which it would be natural to send around endpoints with infinite weight, and shows a safe workaround to circumvent the finite-weight restriction; the last example demonstrates the expressiveness of our calculus in modeling some advanced features of $\mathsf{Sing}^\#$, namely the ability to safely share linear pointers between several processes.

Example 5.1 (forwarder). We illustrate a type derivation for a simple forwarder process that receives two endpoints with dual types and forwards the stream of m-tagged messages coming from the first endpoint to the second one. We have at least two ways to implement the forwarder, depending on whether the stream is homogeneous (all the m-tagged messages carry an argument of the same type) or heterogeneous (different m-tagged messages may carry arguments of possibly different types). Considering the latter possibility we have:

$$\mathrm{FWD}(a) = a?\mathtt{Src}(x : \mathsf{lin}\ T).a?\mathtt{Dest}(y : \mathsf{lin}\ \overline{T}).$$
$$(\mathtt{close}(a) \,|\, \mathtt{rec}\ X.x?\mathtt{m}\langle\alpha\rangle(z : \mathsf{lin}\ \alpha).y!\mathtt{m}\langle\alpha\rangle(z).X)$$

where

$$T = \mathtt{rec}\ \beta.?\mathtt{m}\langle\alpha\rangle(\mathsf{lin}\ \alpha).\beta\ .$$

Below we show the derivation proving that FWD is well typed. To keep the derivation's size manageable, we elide some subprocesses with $[\cdots]$ and we define $\Gamma = x : \mathsf{lin}\ T, y : \mathsf{lin}\ \overline{T}$.

$$\cfrac{\cfrac{\cfrac{\cfrac{\overline{\{X \mapsto (\emptyset; \Gamma)\}; \alpha; \Gamma \vdash X}}\ \text{(T-Var)}}{\{X \mapsto (\emptyset; \Gamma)\}; \alpha; \Gamma, z : \mathsf{lin}\ \alpha \vdash y!\mathtt{m}\langle\alpha\rangle(z).X}\ \text{(T-Send)}}{\cfrac{\{X \mapsto (\emptyset; \Gamma)\}; \emptyset; \Gamma \vdash x?\mathtt{m}\langle\alpha\rangle(z : \mathsf{lin}\ \alpha).[\cdots]}{x : \mathsf{lin}\ T, y : \mathsf{lin}\ \overline{T} \vdash \mathtt{rec}\ X.[\cdots]}\ \text{(T-Rec)}}\ \text{(T-Receive)}}{\cfrac{\cfrac{\overline{a : \mathsf{lin}\ \mathsf{end} \vdash \mathtt{close}(a)}\ \text{(T-Close)} \qquad }{a : \mathsf{lin}\ \mathsf{end}, x : \mathsf{lin}\ T, y : \mathsf{lin}\ \overline{T} \vdash \mathtt{close}(a) \,|\, \mathtt{rec}\ X.[\cdots]}\ \text{(T-Par)}}{\cfrac{a : \mathsf{lin}\ ?\mathtt{Dest}(\mathsf{lin}\ \overline{T}).\mathsf{end}, x : \mathsf{lin}\ T \vdash a?\mathtt{Dest}(y : \mathsf{lin}\ \overline{T}).[\cdots]}{a : \mathsf{lin}\ ?\mathtt{Src}(\mathsf{lin}\ T).?\mathtt{Dest}(\mathsf{lin}\ \overline{T}).\mathsf{end} \vdash \mathrm{FWD}(a)}\ \text{(T-Receive)}}\ \text{(T-Receive)}}$$

Observe that, by the time rule (T-Var) is applied for the process variable X, a type variable α has accumulated into the bound type variables context which was empty when X was introduced in (T-Rec). Therefore, it is essential for rule (T-Var) to discharge extra type variables in the bound type variable context for declaring this process well typed. ∎

Example 5.2 (linear lists). In most of the examples we have presented so far the type of services begins with an output action, suggesting that it is the consumers of these services that play the first move and *invoke* them by sending a message. There are cases, in particular with the modeling of datatypes, where it is more natural to adopt the dual point of view,

in which the reception of a message indicates the consumption of the data type. In this example we 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. Along these lines, the type of lists with elements of type α would be encoded as the endpoint type

$$\texttt{List}(\alpha) = \texttt{rec } \beta.(?\texttt{Nil}().\texttt{end} + ?\texttt{Cons}(\textsf{lin } \alpha, \textsf{lin } \beta).\texttt{end})$$

Note that, just as this type denotes lists of arbitrary length, the encoding of lists in terms of messages within endpoints may yield chains of pointers of arbitrary length because of the recursion of β through an input prefix. As a consequence we have $\|\texttt{List}(\alpha)\|_{\{\alpha\}} = \infty$, meaning that our type system would reject any output operation sending a list over an endpoint. Incidentally, since a non-empty list is encoded as a `Cons`-tagged message containing another list, the finite-weight restriction on the type of message arguments would in fact prevent the construction of any non-trivial list, rendering the type $\texttt{List}(\alpha)$ useless.

It is possible to fix this by requiring the consumers of the list to signal the imminent deconstruction via a "prompt" message. This corresponds to defining

$$\texttt{List}(\alpha) = \texttt{rec } \beta.!\texttt{Prompt}().(?\texttt{Nil}().\texttt{end} + ?\texttt{Cons}(\textsf{lin } \alpha, \textsf{lin } \beta).\texttt{end})$$

The insertion of an output action between the binding of β and its occurrence among the arguments of `Cons` nullifies the weight of $\texttt{List}(\alpha)$, that is $\|\texttt{List}(\alpha)\|_{\{\alpha\}} = 0$. To see why this is sufficient for preventing the creation of cycles in the heap consider a process

$$b!\texttt{Cons}(x, a).P$$

where we assume that $a : \texttt{List}(\alpha)$ and $b : !\texttt{Nil}().\texttt{end} \oplus !\texttt{Cons}(\textsf{lin } \alpha, \texttt{List}(\alpha)))$. The intention here is to yield a leak like the one generated by the process in (5.1). Note however that the peer endpoint of b must have already been used for sending the `Prompt`-tagged message, while a has type $\texttt{List}(\alpha)$ and therefore no `Prompt`-tagged message has been sent on a yet. We conclude that a cannot be b's peer.

As an example of list-manipulating function we can now define the polymorphic consing service on channel c, that creates a list from a head and a tail, thus:

$$\begin{aligned}
\text{CONS}(c) = &\texttt{rec } X.c?\texttt{Invoke}\langle\alpha\rangle(x : \textsf{lin } T). \\
&x?\texttt{Arg}(y : \textsf{lin } \alpha).x?\texttt{Arg}(z : \textsf{lin } \texttt{List}(\alpha)). \\
&\texttt{open}(a : \texttt{List}(\alpha), b : \overline{\texttt{List}(\alpha)}). \\
&(b?\texttt{Prompt}().b!\texttt{Cons}(y, z).\texttt{close}(b) \mid x!\texttt{res}(a).X)
\end{aligned}$$

where

$$T = ?\texttt{Arg}(\textsf{lin } \alpha).?\texttt{Arg}(\textsf{lin } \texttt{List}(\alpha)).!\texttt{Res}(\textsf{lin } \texttt{List}(\alpha)).\texttt{end}$$

The interested reader can verify that

$$c : \texttt{rec } \beta.?\texttt{Invoke}\langle\alpha\rangle(\textsf{lin } T).\beta \vdash \text{CONS}(c)$$

is derivable. ∎

Example 5.3. Development of the Singularity OS prototype has suggested that there are many scenarios in which the ownership invariant, requiring that a given object – an endpoint – can be owned exclusively by one sole process at any given time, easily leads to convoluted code. For this reason, $\textsf{Sing}^{\#}$ provides a `TCell<`α`>` class that permits the

Table 9: Modeling of a shared mutable cell.

$$
\begin{aligned}
\mathrm{MKCELL}(a) ={}& a?\mathtt{Invoke}\langle\alpha\rangle(x : \mathsf{lin}\ !\mathtt{Res}(\mathsf{un}\ \overline{T_{\mathtt{TCell}}(\alpha)}).\mathtt{end}).\\
& \mathtt{open}(c : T_{\mathtt{TCell}}(\alpha)).\mathtt{open}(\mathit{buffer} : T_{\mathtt{Buffer}}(\alpha)).\\
& \mathtt{open}(\mathit{acquire} : T_{\mathtt{Acquire}}(\alpha)).\mathtt{open}(\mathit{release} : T_{\mathtt{Release}}(\alpha)).\\
& x!\mathtt{Res}(\overline{c}).(\mathrm{EMPTY}(\alpha, c) \mid \mathtt{close}(x) \mid \mathrm{MKCELL}(a))\\
\mathrm{EMPTY}(\alpha, c) ={}& c?\mathtt{Invoke}(x : \mathsf{lin}\ T).\\
& (\quad x?\mathtt{Acquire}().\overline{\mathit{acquire}}!\mathtt{In}(x).\mathrm{EMPTY}(\alpha, c)\\
& + x?\mathtt{Release}().x!\mathtt{Ok}().x?\mathtt{Arg}(y : \mathsf{lin}\ \alpha).\\
& \quad \mathtt{if}\ \mathtt{empty}(\mathit{acquire})\ \mathtt{then}\ (\overline{\mathit{buffer}}!\mathtt{In}(y) \mid \mathtt{close}(x) \mid \mathrm{FULL}(\alpha, c))\\
& \qquad\qquad\qquad\qquad\quad \mathtt{else}\ \mathit{acquire}?\mathtt{In}(z : \mathsf{lin}\ !\mathtt{Res}(\mathsf{lin}\ \alpha).\mathtt{end}).z!\mathtt{Res}(y).\\
& \qquad\qquad\qquad\qquad\qquad\qquad (\mathtt{close}(x) \mid \mathtt{close}(z) \mid \mathrm{EMPTY}(\alpha, c)))\\
\mathrm{FULL}(\alpha, c) ={}& c?\mathtt{Invoke}(x : \mathsf{lin}\ T).\\
& (\quad x?\mathtt{Acquire}().\mathit{buffer}?\mathtt{In}(y : \mathsf{lin}\ \alpha).x!\mathtt{Res}(y).\\
& \quad \mathtt{if}\ \mathtt{empty}(\mathit{release})\ \mathtt{then}\ (\mathtt{close}(x) \mid \mathrm{EMPTY}(\alpha, c))\\
& \qquad\qquad\qquad\qquad\quad \mathtt{else}\ \mathit{release}?\mathtt{In}(z : \mathsf{lin}\ !\mathtt{Ok}().?\mathtt{Res}(\mathsf{lin}\ \alpha).\mathtt{end}).\\
& \qquad\qquad\qquad\qquad\qquad\quad z!\mathtt{Ok}().z?\mathtt{Arg}(y : \mathsf{lin}\ \alpha).\overline{\mathit{buffer}}!\mathtt{In}(y).\\
& \qquad\qquad\qquad\qquad\qquad\quad (\mathtt{close}(x) \mid \mathtt{close}(z) \mid \mathrm{FULL}(\alpha, c)))\\
& + x?\mathtt{Release}().\overline{\mathit{release}}!\mathtt{In}(x).\mathrm{FULL}(\alpha, c))\\[6pt]
T(\alpha) ={}& ?\mathtt{Acquire}().!\mathtt{Res}(\mathsf{lin}\ \alpha).\mathtt{end} + ?\mathtt{Release}().!\mathtt{Ok}().?\mathtt{Arg}(\mathsf{lin}\ \alpha).\mathtt{end}\\
T_{\mathtt{TCell}}(\alpha) ={}& \mathtt{rec}\ \beta.?\mathtt{Invoke}(\mathsf{lin}\ T(\alpha)).\beta\\
T_{\mathtt{Buffer}}(\alpha) ={}& \mathtt{rec}\ \beta.?\mathtt{In}(\mathsf{lin}\ \alpha).\beta\\
T_{\mathtt{Acquire}}(\alpha) ={}& \mathtt{rec}\ \beta.?\mathtt{In}(\mathsf{lin}\ !\mathtt{Res}(\mathsf{lin}\ \alpha).\mathtt{end}).\beta\\
T_{\mathtt{Release}}(\alpha) ={}& \mathtt{rec}\ \beta.?\mathtt{In}(\mathsf{lin}\ !\mathtt{Ok}().?\mathtt{Res}(\mathsf{lin}\ \alpha).\mathtt{end}).\beta
\end{aligned}
$$

unrestricted sharing of exchange heap pointers at the expense of some runtime checks. In practice, an instance of `TCell<`α`>` acts like a 1-place buffer for a linear pointer of type α and can be shared non-linearly among different processes. A process willing to use the pointer must explicitly *acquire* it, while a process that has finished using the pointer must *release* it. The internal implementation of `TCell<`α`>` makes sure that, once the pointer has been acquired, all subsequent acquisition requests will be blocked until a release is performed. The interface of `TCell<`α`>` is as follows:

```
class TCell<α> {
  TCell([Claims] α in ExHeap);
  α in ExHeap Acquire();
  void Release([Claims] α in ExHeap);
}
```

Table 9 presents an implementation of the $\mathsf{Sing}^{\#}$ class `TCell<`α`>` in our process calculus. For readability, we have defined the MKCELL(a) process in terms of (mutually) recursive equations that can be folded into a proper term as by [7]. Below we describe the process from a bird's eye point of view and expect the reader to fill in the missing details.

MKCELL(a) waits for invocations on endpoint a. Each invocation creates a new cell represented as a linear endpoint c (retained by the implementation) and an unrestricted pointer $\overline{c}$ (that can be shared by the users of the cell). The cell consists of three unrestricted

endpoints: *buffer* is the actual buffer that contains the pointer to be shared, while *acquire* and *release* are used to enqueue pending requests for acquisition and release of the cell content. The implementation ensures that *buffer* always contains at most one message (of type α), that *acquire* can have pending requests only when *buffer* is empty, and that *release* can have pending requests only when *buffer* is full. Users of the cell send invocation requests on endpoint c. When the cell is empty, any acquisition request is enqueued into *acquire* while a release request checks whether there are pending acquisition requests by means of the $\texttt{empty}(acquire)$ primitive: if there is no pending request, the released pointer y is stored within *buffer* and the cell becomes full; if there are pending requests, the first one (z) is dequeued and served, and the cell stays empty. When the cell is full, any release request is enqueued into *release* while the first acquisition request is served immediately. Then, the cell may become empty or stay full depending on whether there are pending release requests.

One aspect of this particular implementation which is highlighted by the endpoint type $T(\alpha)$ is the handling of multiple release requests. In principle, it could be reasonable for the $\texttt{Release}$-tagged message to carry an argument of type α, the pointer being released. However, if this were the case a process releasing a pointer would *immediately* transfer the ownership of the pointer to the cell, even in case the cell is in a full state. This is because communication is asynchronous and send operations are non-blocking, so the message with the pointer would be enqueued into *release*, which is permanently owned by the cell, regardless of whether the cell is already full. In our modeling, the $\texttt{Release}$-tagged message carries no argument, its only purpose being to signal the *intention* for a process to release a pointer. If the cell is empty, then the cell answers the requester with an $\texttt{Ok}$-tagged message, and only at that point the pointer (and its ownership) is transferred from the requester to the cell with an $\texttt{Arg}$-tagged message. If however the cell is full when the release request is made, the $\texttt{Ok}$-tagged message is deferred and the requester remains the formal owner of the pointer being released until the cell becomes empty again. ∎

6. Algorithms

In this section we define algorithms for deciding subtyping and for computing the weight of (endpoint) types. We also argue how the typing rules in Table 8 can be easily turned into a type checking algorithm using a technique explained elsewhere.

6.1. **Subtyping.** The algorithm for deciding the subtyping relation $T \leqslant S$ is more easily formulated if we make a few assumptions on the variables occurring in T and S. The reason is that $\leqslant$ (Definition 3.2) implicitly uses alpha renaming in order to match the bound type variables occurring in one endpoint type with the bound type variables occurring in the other endpoint type. However, termination of the subtyping algorithm can be guaranteed only if we perform these renamings in a rather controlled way, and the assumptions we are going to make are aimed at this.

Definition 6.1 (independent endpoint types)**.** We say that T and S are *independent* if:
(1) $\texttt{ftv}(T) \cap \texttt{btv}(T) = \emptyset$;
(2) $\texttt{ftv}(S) \cap \texttt{btv}(S) = \emptyset$;
(3) no type variable in T or in S is bound more than once;
(4) $\texttt{btv}(T) \cap \texttt{btv}(S) = \emptyset$.

Table 10: Algorithmic subtyping rules.

$$
\begin{array}{cc}
\text{(S-Var)} & \text{(S-End)} \\
\mathscr{S} \vdash_{\mathsf{m}} \alpha \leqslant_{\mathsf{a}} \alpha & \mathscr{S} \vdash_{\mathsf{m}} \mathsf{end} \leqslant_{\mathsf{a}} \mathsf{end}
\end{array}
$$

$$
\begin{array}{cc}
\text{(S-Axiom)} & \text{(S-Rec Left)} \\
\dfrac{(T, S) \in \mathscr{S}}{\mathscr{S} \vdash_{\mathsf{m}} T \leqslant_{\mathsf{a}} S} & \dfrac{\mathscr{S} \cup \{(\mathtt{rec}\ \alpha.T, S)\} \vdash_{\mathsf{m}} T\{\mathtt{rec}\ \alpha.T/\alpha\} \leqslant_{\mathsf{a}} S}{\mathscr{S} \vdash_{\mathsf{m}} \mathtt{rec}\ \alpha.T \leqslant_{\mathsf{a}} S}
\end{array}
$$

$$
\begin{array}{cc}
\text{(S-Rec Right)} & \text{(S-Type)} \\
\dfrac{\mathscr{S} \cup \{(T, \mathtt{rec}\ \alpha.S)\} \vdash_{\mathsf{m}} T \leqslant_{\mathsf{a}} S\{\mathtt{rec}\ \alpha.S/\alpha\}}{\mathscr{S} \vdash_{\mathsf{m}} T \leqslant_{\mathsf{a}} \mathtt{rec}\ \alpha.S} & \dfrac{q \leq q' \qquad \mathscr{S} \vdash_{\mathsf{m}} T \leqslant_{\mathsf{a}} S}{\mathscr{S} \vdash_{\mathsf{m}} q\ T \leqslant_{\mathsf{a}} q'\ S}
\end{array}
$$

$$
\text{(S-Input)}
$$
$$
\dfrac{\mathscr{S}' = \mathscr{S} \cup \{(T, S)\} \qquad I \subseteq J \qquad \mathscr{S}' \vdash_{\mathsf{m}} t_i\{\mathsf{m}(\alpha_i, \beta_i)/\alpha_i\} \leqslant_{\mathsf{a}} s_i\{\mathsf{m}(\alpha_i, \beta_i)/\beta_i\}\ ^{(i \in I)}}{\mathscr{S}' \vdash_{\mathsf{m}} T_i\{\mathsf{m}(\alpha_i, \beta_i)/\alpha_i\} \leqslant_{\mathsf{a}} S_i\{\mathsf{m}(\alpha_i, \beta_i)/\beta_i\}\ ^{(i \in I)}}{\mathscr{S} \vdash_{\mathsf{m}} T \equiv \{?\mathtt{m}_i\langle\alpha_i\rangle(t_i).T_i\}_{i \in I} \leqslant_{\mathsf{a}} \{?\mathtt{m}_j\langle\beta_j\rangle(s_j).S_j\}_{j \in J} \equiv S}
$$

$$
\text{(S-Output)}
$$
$$
\dfrac{\mathscr{S}' = \mathscr{S} \cup \{(T, S)\} \qquad J \subseteq I \qquad \mathscr{S}' \vdash_{\mathsf{m}} s_j\{\mathsf{m}(\alpha_j, \beta_j)/\beta_j\} \leqslant_{\mathsf{a}} t_j\{\mathsf{m}(\alpha_j, \beta_j)/\alpha_j\}\ ^{(j \in J)}}{\mathscr{S}' \vdash_{\mathsf{m}} T_j\{\mathsf{m}(\alpha_j, \beta_j)/\alpha_j\} \leqslant_{\mathsf{a}} S_j\{\mathsf{m}(\alpha_j, \beta_j)/\beta_j\}\ ^{(j \in J)}}{\mathscr{S} \vdash_{\mathsf{m}} T \equiv \{!\mathtt{m}_i\langle\alpha_i\rangle(t_i).T_i\}_{i \in I} \leqslant_{\mathsf{a}} \{!\mathtt{m}_j\langle\beta_j\rangle(s_j).S_j\}_{j \in J} \equiv S}
$$

Informally, conditions (1–3) state that T and S obey the so-called Barendregt convention for type variables, by stating that free and bound type variables are disjoint and that every type variable is bound at most once. Condition (4) makes sure that there is no shared bound type variable between T and S.

We will restrict the subtyping algorithm to independent endpoint types. This is bearable as every pair of endpoint types can be easily rewritten into an equivalent pair of independent endpoint types:

Proposition 6.1. *For every T and S there exist independent T' and S' such that $T = T'$ and $S = S'$.*

Proof sketch. A structural induction on T followed by a structural induction on S, in both cases renaming bound variables with fresh ones. □

The subtyping algorithm is defined using the rules in Table 10, thus:

Definition 6.2 (subtyping algorithm). Let T and S be independent endpoint types. Let m be a map from unordered pairs of type variables to type variables such that $\mathsf{m}(\alpha, \beta) \notin \mathtt{ftv}(T) \cup \mathtt{btv}(T) \cup \mathtt{ftv}(S) \cup \mathtt{btv}(S)$ for every $\alpha \in \mathtt{btv}(T)$ and $\beta \in \mathtt{btv}(S)$. We write $\vdash_{\mathsf{m}} T \leqslant_{\mathsf{a}} S$ if and only if $\emptyset \vdash_{\mathsf{m}} T \leqslant_{\mathsf{a}} S$ is derivable with the axioms and rules in Table 10, where we give rule (S-Axiom) the highest priority, followed by rule (S-Rec Left), followed by rule (S-Rec Right), followed by all the remaining rules which are syntax-directed.[4]

---

[4]The relative priority of rules (S-Rec Left) and (S-Rec Right) is irrelevant since they are confluent.

The algorithm derives judgments of the form $\mathscr{S} \vdash_{\mathsf{m}} T \leqslant_{\mathsf{a}} S$, where $\mathscr{S}$ is a memoization context that records pairs of endpoint types that are assumed to be related by subtyping. The map m is used for unifying consistently the bound type variables of the endpoint types being related. The same (unordered) pair of bound type variables (α, β) is always unified to the same fresh type variable $\mathsf{m}(\alpha, \beta)$, which is essential for guaranteeing that the memoization context $\mathscr{S}$ does not grow unwieldy. The fact that we work with unordered pairs simply means that $\mathsf{m}(\alpha, \beta) = \mathsf{m}(\beta, \alpha)$ for every $\alpha \in \mathtt{btv}(T)$ and $\beta \in \mathtt{btv}(S)$. The axioms and rules in Table 10 are mostly unremarkable, since they closely mimic the coinductive definition of subtyping (Definition 3.2), therefore we only comment on the peculiar features of this deduction system: Axiom (S-Axiom) allows one to immediately deduce $\mathscr{S} \vdash_{\mathsf{m}} T \leqslant_{\mathsf{a}} S$ whenever the pair (T, S) occurs in $\mathscr{S}$. This prevents the algorithm to loop forever when comparing recursive endpoint types. A pair (T, S) is added to $\mathscr{S}$ whenever a constructor is crossed, which happens in rules (S-Rec Left), (S-Rec Right), (S-Input), and (S-Output). Rules (S-Rec Left) and (S-Rec Right) unfold recursive endpoint types in order to expose their outermost proper constructor (an internal/external choice or end). Contractivity of endpoint types guarantees that a finite number of applications of these rules is always enough to achieve this exposure. In Definition 3.2 recursive endpoint types are not treated explicitly since equality '$=$' is defined modulo folding/unfolding of recursions. Rules (S-Input) and (S-Output) deal with inputs and outputs. Note that the pairs of endpoint types being compared in the conclusions of the rules have distinct sets $\{\alpha_i\}_{i \in I}$ and $\{\beta_j\}_{j \in J}$ of bound type variables that are unified in the premises by means of the map m.

The following result establishes the correctness and completeness of the subtyping algorithm with respect to $\leqslant$ for independent endpoint types.

Theorem 6.1 (correctness and completeness). *Let T_0 and S_0 be independent endpoint types and m be a map as by Definition 6.2. Then $\vdash_{\mathsf{m}} T_0 \leqslant_{\mathsf{a}} S_0$ if and only if $T_0 \leqslant S_0$.*

Example 6.1. Consider the endpoint types
$$
\begin{aligned}
T &\equiv \mathtt{rec}\ \alpha.!\mathsf{a}\langle\alpha_1\rangle(!\mathsf{b}\langle\alpha_2\rangle(\alpha_2).\alpha_1).\alpha \\
S &\equiv \mathtt{rec}\ \beta.!\mathsf{a}\langle\beta_1\rangle(!\mathsf{b}\langle\beta_2\rangle(\beta_2).\beta_1 \oplus !\mathsf{c}\langle\beta_3\rangle(\beta_3).\mathsf{end}).\beta
\end{aligned}
$$
and observe that they are independent. The following derivation, together with Theorem 6.1, shows that $T \leqslant S$:

$$
\cfrac{
\cfrac{
\cfrac{\cfrac{}{\mathscr{S}_3 \vdash_{\mathsf{m}} \gamma_2 \leqslant_{\mathsf{a}} \gamma_2}\ {\scriptstyle(\text{S-Var})}\quad \cfrac{}{\mathscr{S}_3 \vdash_{\mathsf{m}} \gamma_1 \leqslant_{\mathsf{a}} \gamma_1}\ {\scriptstyle(\text{S-Var})}}
{\mathscr{S}_2 \vdash_{\mathsf{m}} !\mathsf{b}\langle\beta_2\rangle(\beta_2).\gamma_1 \oplus !\mathsf{c}\langle\beta_3\rangle(\beta_3).\mathsf{end} \leqslant_{\mathsf{a}} !\mathsf{b}\langle\alpha_2\rangle(\alpha_2).\gamma_1}\ {\scriptstyle(\text{S-Output})}
\quad
\cfrac{\cfrac{}{\mathscr{S}_2 \vdash_{\mathsf{m}} T \leqslant_{\mathsf{a}} S}\ {\scriptstyle(\text{S-Axiom})}}{}\ {\scriptstyle(\text{S-Output})}
}
{
\cfrac{
\cfrac{\mathscr{S}_1 \vdash_{\mathsf{m}} !\mathsf{a}\langle\alpha_1\rangle(!\mathsf{b}\langle\alpha_2\rangle(\alpha_2).\alpha_1).T \leqslant_{\mathsf{a}} !\mathsf{a}\langle\beta_1\rangle(!\mathsf{b}\langle\beta_2\rangle(\beta_2).\beta_1 \oplus !\mathsf{c}\langle\beta_3\rangle(\beta_3).\mathsf{end}).S}
{\{(T,S)\} \vdash_{\mathsf{m}} !\mathsf{a}\langle\alpha_1\rangle(!\mathsf{b}\langle\alpha_2\rangle(\alpha_2).\alpha_1).T \leqslant_{\mathsf{a}} S}\ {\scriptstyle(\text{S-Rec Right})}
}
{\emptyset \vdash_{\mathsf{m}} T \leqslant_{\mathsf{a}} S}\ {\scriptstyle(\text{S-Rec Left})}
}
$$

where we have used the abbreviations:
- $\gamma_1 = \mathsf{m}(\alpha_1, \beta_1)$ and $\gamma_2 = \mathsf{m}(\alpha_2, \beta_2)$;
- $\mathscr{S}_1 = \{(T, S), (!\mathsf{a}\langle\alpha_1\rangle(!\mathsf{b}\langle\alpha_2\rangle(\alpha_2).\alpha_1).T, S)\}$;
- $\mathscr{S}_2 = \mathscr{S}_1 \cup \{(!\mathsf{a}\langle\alpha_1\rangle(!\mathsf{b}\langle\alpha_2\rangle(\alpha_2).\alpha_1).T, !\mathsf{a}\langle\beta_1\rangle(!\mathsf{b}\langle\beta_2\rangle(\beta_2).\beta_1 \oplus !\mathsf{c}\langle\beta_3\rangle(\beta_3).\mathsf{end}).S)\}$;
- $\mathscr{S}_3 = \mathscr{S}_2 \cup \{(!\mathsf{b}\langle\beta_2\rangle(\beta_2).\gamma_1 \oplus !\mathsf{c}\langle\beta_3\rangle(\beta_3).\mathsf{end}, !\mathsf{b}\langle\alpha_2\rangle(\alpha_2).\gamma_1)\}$. ∎

6.2. **Type Weight.** We now address the computation of the weight of an (endpoint) type, which is the least of its weight bounds or ∞ if it has no weight bound. Unlike the definition of weight bound (Definition 5.1), the algorithm avoids unfoldings of recursive endpoint types in order to terminate. This imposes a refinement in the strategy we use for weighing type variables. Recall that, according to Definition 5.1, when determining $\|T\|_{\Delta_0}$ type variables are weighed either 0 or ∞ according to whether they occur in the context Δ_0 or in $\mathtt{btv}(T)$ when they are bound in an input or output prefix. If we avoid unfoldings of recursions, we must also deal with type variables that are bound by recursive terms $\mathtt{rec}\ \alpha.T$. The idea is that these variables must be weighed differently, depending on whether they occur *within an input prefix* of T or not. For this reason, we use another context Δ that contains the subset of type variables bound by a recursive term and that can be weighed 0.

Ultimately, we define a function $\mathtt{W}(\Delta_0, \Delta, T)$ by induction on the structure of T, thus:

$$\mathtt{W}(\Delta_0, \Delta, \mathsf{end}) = 0$$
$$\mathtt{W}(\Delta_0, \Delta, \alpha) = \begin{cases} 0 & \text{if } \alpha \in \Delta_0 \cup \Delta \\ \infty & \text{otherwise} \end{cases}$$
$$\mathtt{W}(\Delta_0, \Delta, \mathtt{rec}\ \alpha.T) = \mathtt{W}(\Delta_0, \Delta \cup \{\alpha\}, T)$$
$$\mathtt{W}(\Delta_0, \Delta, \{!\mathtt{m}_i\langle\alpha_i\rangle(t_i).T_i\}_{i\in I}) = 0$$
$$\mathtt{W}(\Delta_0, \Delta, \{?\mathtt{m}_i\langle\alpha_i\rangle(t_i).T_i\}_{i\in I}) = \max\{1 + \mathtt{W}(\Delta_0, \emptyset, t_i), \mathtt{W}(\Delta_0, \Delta \setminus \{\alpha_i\}, T_i)\}_{i\in I}$$
$$\mathtt{W}(\Delta_0, \Delta, q\ T) = \mathtt{W}(\Delta_0, \Delta, T)$$

The first and fourth equations give a null weight to end and endpoint types in a send state, as expected. The third equation weighs a recursive term $\mathtt{rec}\ \alpha.T$ by weighing the body T and recording the fact that α can be given a null weight, as long as α does not occur in a prefix of T. The second equation weighs a type variable α: if α occurs in $\Delta_0 \cup \Delta$, then it means that either α occurs free in the original endpoint type being weighed and therefore must be given a null weight, or α is bound in a recursive term $\mathtt{rec}\ \alpha.S$ but it does not occur within an input prefix of S; if α does not occur in $\Delta_0 \cup \Delta$, then it means that either α was bound in an prefix of an endpoint type in send/receive state, or it was bound in a recursive term $\mathtt{rec}\ \alpha.S$ and it occurs within an input prefix of S. The fifth equation determines the weight of an endpoint type in receive state. The rule essentially mimics the corresponding condition of Definition 5.1, but notice that when weighing the types t_i in the prefixes the context Δ is emptied, since if any of the type variables in it is encountered, then it must be given an infinite weight. The last equation simply determines the weight of a qualified endpoint type to be the weight of the endpoint type itself.

We work out a few simple examples to help clarifying the algorithm:

- $\mathtt{W}(\emptyset, \emptyset, ?\mathtt{m}\langle\alpha\rangle(\mathsf{end}).\mathsf{end}) = \max\{1 + \mathtt{W}(\emptyset, \emptyset, \mathsf{end}), \mathtt{W}(\emptyset, \emptyset, \mathsf{end})\} = 1$;
- $\mathtt{W}(\emptyset, \emptyset, ?\mathtt{m}\langle\alpha\rangle(\alpha).\mathsf{end}) = \max\{1 + \mathtt{W}(\emptyset, \emptyset, \alpha), \mathtt{W}(\emptyset, \emptyset, \mathsf{end})\} = \infty$;
- $\mathtt{W}(\emptyset, \emptyset, \mathtt{rec}\ \alpha.?\mathtt{m}(\alpha).\mathsf{end}) = \mathtt{W}(\emptyset, \{\alpha\}, ?\mathtt{m}(\alpha).\mathsf{end}) = \max\{1 + \mathtt{W}(\emptyset, \emptyset, \alpha), \mathtt{W}(\emptyset, \{\alpha\}, \mathsf{end})\} = \infty$;
- $\mathtt{W}(\emptyset, \emptyset, \mathtt{rec}\ \alpha.?\mathtt{m}(\mathsf{end}).\alpha) = \mathtt{W}(\emptyset, \{\alpha\}, ?\mathtt{m}(\mathsf{end}).\alpha) = \max\{1 + \mathtt{W}(\emptyset, \emptyset, \mathsf{end}), \mathtt{W}(\emptyset, \{\alpha\}, \alpha)\} = 1$.

In the last example, note that the type variable α that virtually represents the recursive term $\mathtt{rec}\ \alpha.T$ is weighed 0 even though the whole term turns out to have weight 1. The idea is that the proper weight of the whole term will be computed anyway according to the structure of the term in which α occurs, and therefore we can safely approximate the weight of α to 0. This property of the algorithm, which is also one of the key ingredients for proving its correctness, can be formalized as the fact that the weight of a recursive term and of its unfolding are the same:

Proposition 6.2. $\text{W}(\Delta_0, \emptyset, \texttt{rec } \alpha.T) = \text{W}(\Delta_0, \emptyset, T\{\texttt{rec } \alpha.T/\alpha\})$.

We conclude with the formal statement saying that the algorithm for computing weights is correct. Its termination is guaranteed as it works by structural induction over finite terms.

Theorem 6.2. $\|T\|_\Delta = \text{W}(\Delta, \emptyset, T)$.

6.3. **Type Checking.** In Sections 6.1 and 6.2 we have already presented algorithms for deciding whether two (endpoint) types are related by subtyping and for computing the weight of (endpoint) types. Therefore, there is just one aspect left that makes the type checking rules in Table 8 non-algorithmic, which is the decomposition of the type environment Γ into $\Gamma_1 + \Gamma_2$ when attempting to derive the judgment $\Sigma; \Delta; \Gamma \vdash P \mid Q$ by means of rule (T-PAR). The idea is to look at the free names of P and Q that have linear types in Γ and to split Γ in such a way that $\texttt{dom}(\Gamma_1|_{\text{lin}}) \subseteq \texttt{fn}(P)$ and $\texttt{dom}(\Gamma_2|_{\text{lin}}) \subseteq \texttt{fn}(Q)$ and $\texttt{dom}(\Gamma_1|_{\text{un}}) = \texttt{dom}(\Gamma_2|_{\text{un}}) = \texttt{dom}(\Gamma|_{\text{un}})$. Clearly, if P and Q share a free name that has a linear type in Γ there is no way to derive the judgment $\Sigma; \Delta; \Gamma \vdash P \mid Q$. We omit a formal definition of this splitting since it can be worked out precisely as explained in [10].

7. RELATED WORK

Singularity OS. Copyless message passing is one of the key features adopted by the Singularity OS [15] to compensate the overhead of communication-based interactions between isolated processes. Communication safety is enforced by checking processes against *channel contracts* that are *deterministic*, *autonomous*, and *synchronizing* [21, 24]. A contract is deterministic if there cannot be two transitions that differ only for the target state, autonomous if every two transitions departing from the same state are either two sends or two receives, and synchronizing if every loop that goes through a final state has at least one input and one output action. As argued in [8], session types can model channel contracts quite well because they always correspond by construction to contracts that are deterministic and autonomous. Session types like those adopted in this work have just one final state end and therefore are trivially synchronizing, but this implies that we are unable to model contracts where a final state has outgoing transitions. This is not an intrinsic limit of session types (it is possible to extend session types with more general "final states" as shown in [5]) and plausibly this restriction is quite natural in practice (for example, all the channel contracts in the source code of Singularity OS have final states without outgoing transitions).

Interestingly, already in [8] it was observed that special attention must be deserved to the type of endpoints that are sent as messages to avoid inconsistencies. In Singularity OS, endpoints (as well as any other memory block) allocated in the exchange heap are explicitly tagged with the identifier of their owner process, and when a block changes owner (because its pointer is sent in a message) it is the sender's responsibility to update the tag with the identifier of the receiver process. If this update is not performed atomically (and it cannot be, for efficiency reasons) the following can happen: a process sends a message m on an endpoint a whose peer b is owned by some process P_1; the sender therefore tags m with P_1; simultaneously, P_1 sends b away to some other process P_2; message m is now formally owned by P_1, while in fact it is enqueued in an endpoint that is owned by P_2. The authors of [8] argue that this inconsistency is avoided if only endpoint in a "send state" (those whose

type begins with an internal choice) can be sent as messages. The reason is that, if b is in a "send state", then a, which must have a dual type, is in a "receive state", and therefore it is not possible to send message m on it. In this respect, our work shows that the "send state" restriction has deeper motivations that go beyond the implementation details of ownership transfer, it gives formal evidence that the restriction devised in [8] is indeed safe, because endpoints in a "send state" always have a null weight, and it shows how to handle a more expressive type system with polymorphic endpoint types.

Early Type-Theoretic Formalizations of Singularity OS. This work improves previous formalizations of Singularity OS presented in [2, 3]. The main differences regard polymorphic and unrestricted endpoint types and the modeling of $\mathsf{Sing}^{\#}$'s expose.

Polymorphic endpoint types increase the flexibility of the type system and are one of the features of Singularity OS, in the form of polymorphic contracts, documented in the design note dedicated to channels [18]. The most interesting aspect of polymorphic endpoint types is their interaction with the ownership invariant (see the example (5.2)) and with the computation of type weights. Polymorphism was not considered in [2], and in [3] we have introduced a *bounded* form of polymorphism, along the lines of [9], but we did not impose any constraint on the instantiation of type variables without bound which were all estimated to have infinite weight. This proved to be quite restrictive (a simple forwarder process like the one in Example 5.1 would be ill-typed). The crucial observation of the present type system is that type variables denote "abstract" values that can only be passed around. So, just as values that are passed around must have a finite-weight type, it makes sense to impose the same restriction when instantiating type variables. For the sake of simplicity, in the present work we have dropped type bounds for type variables. This allowed us to define the subtyping algorithm as a relatively simple extension of the standard subtyping algorithm for session types [10]. It should be possible to work out a subtyping algorithm for bounded, polymorphic, recursive endpoint types, possibly adapting related algorithms defined for functional types [17, 6], although the details might be quite involved.

In [2, 3] only linear endpoint types were considered. However, as pointed out by some referees, a purely linear type system is quite selective on the sort of constructs that can be effectively modeled with the calculus. For this reason, in the present version we have introduced unrestricted endpoint types in addition to linear ones, with the understanding that other kinds of unrestricted data types (such as the primitive types of boolean or integer values) can be accommodated just as easily. We have shown that unrestricted endpoint types can be used for representing the type of non-linear resources such as permanent services and functions and we have also been able to implement the `TCell` type constructor of $\mathsf{Sing}^{\#}$ (Example 5.3). Interestingly, the introduction of unrestricted endpoint types required very little change to the process language (only a different open primitive) and no change at all to the heap model.

The remaining major difference between [2] and this work is the lack of any expose primitive in the process calculus, which is used in the $\mathsf{Sing}^{\#}$ compiler to keep track of memory ownership. To illustrate the construct, consider the code fragment

```
expose (a) {
  b.Arg(*a);
  *a = new[ExHeap] T();
}
```

which dereferences a cell `a` and sends its content on endpoint `b`. After the `b.Arg(*a)` operation the process no longer owns `*a` but it still owns `a`. Therefore, the ownership invariant could be easily violated if the process were allowed to access `*a` again. To prevent this, the $\mathsf{Sing}^\#$ compiler allows (linear) pointer dereferentiation only within `expose` blocks. The `expose (a)` block temporarily transfers the ownership of `*a` from `a` to the process exposing `a` and is well-typed if the process still owns `*a` at the end of block. In this example, the only way to regain ownership of `*a` is to assign it with the pointer to another object that the process owns. In [2] we showed that all we need to capture the static semantics of `expose` blocks is to distinguish cells with type $*t$ (whose content, of type t, is owned by the cell) from cells with type $*\bullet$ (whose content is owned directly by the process). At the beginning of the expose block, the type of `a` turns from $*t$ to $*\bullet$; within the block it is possible to (linearly) use `*a`; at the end of the block, `*a` is assigned with the pointer to a newly allocated object that the process owns, thus turning `a`'s type from $*\bullet$ back to some $*s$. In other words, cell types (and other object types) are simple behavioral types that can be easily modeled in terms of polymorphic endpoint types. In [3] we have shown that the endpoint type

$$\mathtt{CellT} = \mathtt{rec}\ \alpha.(!\mathtt{Set}\langle\beta\rangle(\mathsf{lin}\ \beta).?\mathtt{Get}(\mathsf{lin}\ \beta).\alpha \oplus !\mathtt{Free}().\mathsf{end})$$

corresponds to the open cell type $*\bullet$ that allows for setting a cell with a value of arbitrary type and for freeing the cell. Once the cell has been set, its type turns to some

$$?\mathtt{Get}(t).\mathtt{CellT}$$

corresponding to the cell type $*t$ that only allows for retrieving its content. The cell itself can be easily modeled as a process that behaves according to $\overline{\mathtt{CellT}}$, as shown in [3].

As a final note, in [2] we have shown how to accommodate the possibility of closing endpoints "in advance" (when their type is different from `end`), since this feature is available in $\mathsf{Sing}^\#$. Overall, it seems like the issues it poses exclusively concern the implementation details rather than the peculiar characteristics of the formal model. Consequently, we have decided to drop this feature in the present paper.

Type Weight. Other works [8, 11] introduce apparently similar, finite-size restrictions on session types. In these cases, 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. For example, in [11] the session type $T = \mathtt{rec}\ \alpha.?\mathtt{m}(\mathsf{lin}\ \alpha).\mathsf{end}$ has size 1 (there can be at most one message of type $\mathsf{lin}\ T$ in the queue of an endpoint with type T) and the session type $S = \mathtt{rec}\ \alpha.?\mathtt{m}(\mathsf{lin}\ \mathsf{end}).\alpha$ has size ∞ (there can be any number of messages, each of type $\mathsf{lin}\ \mathsf{end}$, in the queue of an endpoint with type S). In our theory we have just the opposite, that is $\|T\| = \infty$ and $\|S\| = 1$. Despite these differences, the workaround we have used to bound the weight of endpoint types (Example 5.2) can also be used to bound the size of session types as well, as pointed out in [11].

Logic-Based Analysis. A radically different approach for the static analysis of Singularity processes is given by [24, 25], where the authors develop a proof system based on a variant of *separation logic* [19]. The proof system 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 [24] manifest themselves only when both endpoints of any channel have been closed. In particular, it is possible to prove that the function foo in Section 2 is safe, although it may indeed leak some memory. This problem has been subsequently recognized and solved in [23]. Roughly, the solution consists in forbidding the output of a message unless it is possible to prove (in the logic) that the queue that is going to host the message is reachable from the content of the message itself. In principle this condition is optimal, in the sense that it should permit every safe output. However, it relies on the knowledge of the identity of endpoints, that is a very precise information that is not always available. For this reason, [23] also proposes an approximation of this condition, consisting in tagging endpoints of a channel with distinct *roles* (basically, what are called *importing* and *exporting* views in Singularity). Then, an endpoint can be safely sent as a message only if its role matches the one of the endpoint on which it is sent. This solution is incomparable to the one we advocate – restricting the output to endpoints with finite-weight type – suggesting that it may be possible to work out a combination of the two. In any case, neither [24] nor [23] take into account polymorphism.

Global Progress. There exist a few works on session types [1, 5] that guarantee a global progress property for well-typed systems where the basic idea is to impose an order on channels to prevent circular dependencies that could lead to a deadlock. Not surprisingly, the critical processes such as (5.1) that we rule out thanks to the finite-weight restriction on the type of messages are ill typed in these works. It turns out that a faithful encoding of (5.1) into the models proposed in these works is impossible, because the open$(\cdot, \cdot)$ primitive we adopt (and that mimics the corresponding primitive operation in Singularity OS) creates *both* endpoints of a channel within the same process, while the session initiation primitives in [1, 5] associate the fresh endpoints of a newly opened session to different processes running in parallel. This invariant – that the same process cannot own more than one endpoint of the same channel – is preserved in well-typed processes because of a severe restriction: whenever an endpoint c is received, the continuation process cannot use any endpoint other than c and the one from which c was received.

8. 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. Our type system has been inspired by session type theories. 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. We have shown that, in the copyless message passing paradigm, linearity alone is not enough for preventing memory leaks, but also that endpoint types convey enough information – their *weight* – to devise a manageable type system that detects potentially dangerous processes: it is enough to restrict send operations so that only endpoint with a finite-weight type can be sent as messages and only finite-weight endpoint types can instantiate type variables. This restriction can be circumvented in a fairly easy and general way at the cost of a few extra communications, still preserving all the nice properties of the type system (Example 5.2).

We claim that our calculus provides a fairly comprehensive formalization of the peculiar features of $\mathsf{Sing}^\#$, among which are the explicit memory management of the exchange heap, the controlled ownership of memory allocated on the exchange heap, and channel contracts. We have also shown how to accommodate some advanced features of the $\mathsf{Sing}^\#$ type system, namely (the lack of) `[Claims]` annotations, the `TCell` type constructor that allows for the sharing of linear pointers, and polymorphic channel contracts. In prior work [3] we had already shown how polymorphic endpoint types permit the encoding of `expose` blocks for accessing linear pointers stored within other objects allocated on the exchange heap. Interestingly, previous studies on Singularity channel contracts [8] had already introduced a restriction on send operations so that only endpoints in a *send-state*, those whose type begins with an internal choice, can be safely sent as messages. There the restriction was 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. We have shown that there are more reasons for being careful about which endpoints can be sent as messages and that the send-state restriction is a sound approximation of our finite-weight restriction, because endpoints in a send-state always have a null weight.

On a more technical side, we have also developed a decidable theory of polymorphic, recursive behavioral types. Our theory is incomparable with that developed in [9]: we handle recursive behavioral types, whereas [9] only considers finite ones; polymorphism in [9] is bounded, while it is unrestricted in our case. The subtyping relation that takes into account both recursive behaviors and bounds is in fact quite straightforward to define (see [3]), but its decision algorithm appears to be quite challenging. As observed in [9], bounded polymorphic session types share many properties with the type language in system $F_{<:}$ [4], and subtyping algorithms for extensions of $F_{<:}$ with recursive types are well known for their complexity [17, 6]. We leave the decision algorithm for subtyping of behavioral types with recursion and bounded polymorphism as future work.

References

[1] Lorenzo Bettini, Mario Coppo, Loris D'Antoni, Marco De Luca, Mariangiola Dezani-Ciancaglini, and Nobuko Yoshida. Global Progress in Dynamically Interleaved Multiparty Sessions. In *Proceedings of CONCUR'08*, LNCS 5201, pages 418–433. Springer, 2008.

[2] Viviana Bono, Chiara Messa, and Luca Padovani. Typing Copyless Message Passing. In *Proceedings of ESOP'11*, LNCS 6602, pages 57–76. Springer, 2011.

[3] Viviana Bono and Luca Padovani. Polymorphic Endpoint Types for Copyless Message Passing. In *Proceedings of ICE'11*, volume EPTCS 59, pages 52–67, 2011.

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

[5] Giuseppe Castagna, Mariangiola Dezani-Ciancaglini, Elena Giachino, and Luca Padovani. Foundations of Session Types. In *Proceedings of PPDP'09*, pages 219–230. ACM, 2009.

[6] Dario Colazzo and Giorgio Ghelli. Subtyping, Recursion, and Parametric Polymorphism in Kernel Fun. *Information and Computation*, 198(2):71–147, 2005.

[7] Bruno Courcelle. Fundamental Properties of Infinite Trees. *Theoretical Computer Science*, 25:95–169, 1983.

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

[9] Simon Gay. Bounded Polymorphism in Session Types. *Mathematical Structures in Computer Science*, 18(5):895–930, 2008.

[10] Simon Gay and Malcolm Hole. Subtyping for Session Types in the π-calculus. *Acta Informatica*, 42(2-3):191–225, 2005.

[11] Simon Gay and Vasco T. Vasconcelos. Linear Type Theory for Asynchronous Session Types. *Journal of Functional Programming*, 20(01):19–50, 2010.

[12] Marco Giunti and Vasco Thudichum Vasconcelos. A Linear Account of Session Types in the Pi Calculus. In *Proceedings of CONCUR'10*, volume LNCS 6269, pages 432–446, 2010.

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

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

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

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

[17] Alan Jeffrey. A Symbolic Labelled Transition System for Coinductive Subtyping of $F_{\mu\leq}$ Types. In *Proceedings of LICS'01*, pages 323–333. IEEE, 2001.

[18] Microsoft. Singularity Design Note 5: Channel Contracts. Technical report, Microsoft Research, 2004. Available at `http://www.codeplex.com/singularity`.

[19] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *Proceedings of CSL'01*, LNCS 2142, pages 1–19. Springer, 2001.

[20] Luca Padovani. Session Types at the Mirror. In *Proceedings of ICE'09*, volume EPTCS 12, pages 71–86, 2009.

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

[22] Vasco Thudichum Vasconcelos. Fundamentals of Session Types. In *Proceedings of SFM'09*, volume LNCS 5569, pages 158–186. Springer, 2009.

[23] Jules Villard. *Heaps and Hops*. PhD thesis, Laboratoire Spécification et Vérification, ENS Cachan, France, 2011.

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

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

Appendix A. Supplement to Section 3

Proposition A.1 (Proposition 3.1). *The following properties hold:*

(1) $\overline{\overline{T}} = T$.

(2) $\emptyset \Vdash T$ *implies that* $T \bowtie \overline{T}$ *and* $\emptyset \Vdash \overline{T}$.

(3) $\Delta; \{\alpha\} \Vdash T$ *and* $\Delta \Vdash S$ *imply* $\Delta \Vdash T\{S/\alpha\}$.

(4) $\emptyset; \{\alpha\} \Vdash T$ *and* $\emptyset \Vdash S$ *imply* $\overline{T\{S/\alpha\}} = \overline{T}\{S/\alpha\}$.

Proof sketch. Item (1) is proved by induction on T. The only interesting case is when $T \equiv \mathtt{rec}\ \alpha.S$. Then, by definition of dual, we have $\overline{T} \equiv \mathtt{rec}\ \alpha.\overline{S\{\!\{T/\alpha\}\!\}}$ and now:

$$
\begin{aligned}
\overline{\overline{T}} &\equiv \mathtt{rec}\ \alpha.\overline{\overline{S\{\!\{T/\alpha\}\!\}}\{\!\{\overline{T}/\alpha\}\!\}} && \text{(by definition of dual)} \\
&= \mathtt{rec}\ \alpha.\overline{\overline{S\{\!\{T/\alpha\}\!\}}} && \text{(by definition of inner substitution)} \\
&= \mathtt{rec}\ \alpha.(\overline{\overline{S}}\{\!\{T/\alpha\}\!\}) && \text{(because inner substitution and dual commute)} \\
&= \mathtt{rec}\ \alpha.(S\{\!\{T/\alpha\}\!\}) && \text{(by induction hypothesis)} \\
&= \mathtt{rec}\ \alpha.S \equiv T && \text{(by folding the recursion)}
\end{aligned}
$$

Item (2) relies on the fact that duality and unfolding commute. Indeed we have (*) $\overline{\mathtt{rec}\ \alpha.T} \equiv \mathtt{rec}\ \alpha.\overline{T\{\!\{\mathtt{rec}\ \alpha.T/\alpha\}\!\}}$ and now:

$$
\begin{aligned}
\overline{T\{\mathtt{rec}\ \alpha.T/\alpha\}} &= \overline{T\{\!\{\mathtt{rec}\ \alpha.T/\alpha\}\!\}\{\mathtt{rec}\ \alpha.T/\alpha\}} && \text{(def. of inner substitution)} \\
&= \overline{T\{\!\{\mathtt{rec}\ \alpha.T/\alpha\}\!\}}\{\overline{\mathtt{rec}\ \alpha.T}/\alpha\} && \text{(by def. of dual)} \\
&= \overline{T\{\!\{\mathtt{rec}\ \alpha.T/\alpha\}\!\}}\{\mathtt{rec}\ \alpha.\overline{T\{\!\{\mathtt{rec}\ \alpha.T/\alpha\}\!\}}/\alpha\} && \text{(by (*))} \\
&= \overline{\mathtt{rec}\ \alpha.T\{\!\{\mathtt{rec}\ \alpha.T/\alpha\}\!\}} && \text{(by folding the recursion)} \\
&\equiv \overline{\mathtt{rec}\ \alpha.T} && \text{(by (*))}
\end{aligned}
$$

In proving items (2–4) it is also needed the fact that $\Delta \Vdash T$ implies $\mathtt{ftv}(T) \subseteq \Delta$ and these free type variables can only occur within prefixes of T. We let the reader fill in the remaining details. $\square$

Proposition A.2 (Proposition 3.3). *Let $\emptyset \Vdash T$ and $\emptyset \Vdash S$. Then $T \leqslant S$ if and only if $\overline{S} \leqslant \overline{T}$.*

Proof. It is enough to show that

$$
\mathscr{S} \overset{\text{def}}{=} \leqslant \cup \{(\overline{S}, \overline{T}) \mid T \leqslant S \ \& \ \emptyset; \Delta \Vdash T \ \& \ \emptyset; \Delta \Vdash S\}
$$

is a coinductive subtyping. Suppose $(\overline{S}, \overline{T}) \in \mathscr{S}$ where (1) $T \leqslant S$ and (2) $\emptyset; \Delta \Vdash T$ and $\emptyset; \Delta \vdash S$. We reason by cases on the shape of T and S:

- $(T = S = \mathsf{end})$ We conclude immediately since $\overline{\mathsf{end}} = \mathsf{end}$.
- $(T = S = \alpha)$ This case is impossible because of the hypothesis (2).
- $(T = \{?\mathtt{m}_i\langle\alpha_i\rangle(t_i).T_i\}_{i \in I}$ and $S = \{?\mathtt{m}_j\langle\alpha_j\rangle(s_j).S_j\}_{j \in J})$ Then $\overline{T} = \{!\mathtt{m}_i\langle\alpha_i\rangle(t_i).\overline{T_i}\}_{i \in I}$ and $S = \{!\mathtt{m}_i\langle\alpha_i\rangle(s_i).\overline{S_i}\}_{i \in J}$. From (1) we deduce $I \subseteq J$ and $t_i \leqslant s_j$ and $T_i \leqslant S_i$ for every $i \in I$. From (2) we deduce $\emptyset; \Delta, \alpha_i \Vdash T_i$ and $\emptyset; \Delta, \alpha_i \Vdash S_i$. By definition of $\mathscr{S}$ we conclude $(\overline{S_i}, \overline{T_i}) \in \mathscr{S}$ for every $i \in I$.
- $(T = \{!\mathtt{m}_i\langle\alpha_i\rangle(t_i).T_i\}_{i \in I}$ and $S = \{!\mathtt{m}_j\langle\alpha_j\rangle(s_j).S_j\}_{j \in J})$ Dual of the previous case. $\square$

Appendix B. Supplement to Section 5

Before addressing subject reduction and soundness we prove a series of auxiliary results. The first one states an expected property of endpoint types, namely that the weight $\|T\|_{\{\alpha\}}$ where we take the free occurrences of α to have null weight remains finite if we replace the same occurrences of α with an arbitrary, but finite-weight endpoint type S (recall that $T\{S/\alpha\}$ is a capture-avoiding substitution).

Proposition B.1. *Let $\max\{\|T\|_{\{\alpha\}}, \|S\|\} < \infty$. Then $\|T\{S/\alpha\}\| < \infty$.*

Proof. We show that $\{\alpha\} \vdash T :: m$ and $S :: n$ imply $T\{S/\alpha\} :: m + n$. It is enough to show that

$$\mathscr{W} \stackrel{\text{def}}{=} \{(\emptyset, T'\{S/\alpha\}, m + n) \mid \exists m \in \mathbb{N} : \{\alpha\} \vdash T' :: m\}$$

is a coinductive weight bound. Observe that $T'' :: n$ implies $(\emptyset, T'', n) \in \mathscr{W}$. Let $(\emptyset, T'', k) \in \mathscr{W}$. Then there exist T' and m such that $T'' = T'\{S/\alpha\}$ and $k = m+n$ and (*) $\{\alpha\} \vdash T' :: m$. We reason by cases on T' assuming, without loss of generality, that $(\{\alpha\} \cup \mathtt{ftv}(S)) \cap \mathtt{btv}(T') = \emptyset$:

- ($T' = \mathsf{end}$) Trivial.
- ($T' = \alpha$) Then $T'\{S/\alpha\} = S$ and from the hypothesis $S :: n$ we conclude $S :: m + n$.
- ($T' = \beta \neq \alpha$) This case is impossible for it contradicts (*).
- ($T' = \{!\mathtt{m}_i\langle\alpha_i\rangle(t_i).T_i\}_{i\in I}$) Trivial.
- ($T' = \{?\mathtt{m}_i\langle\alpha_i\rangle(q_i \ S_i).T_i\}_{i\in I}$) From (*) we deduce $m > 0$ and $\{\alpha\} \vdash S_i :: m - 1$ and $\{\alpha\} \vdash T_i :: m$ for every $i \in I$. By definition of $\mathscr{W}$ we conclude $(\emptyset, S_i\{S/\alpha\}, (m-1)+n) \in \mathscr{W}$ and $(\emptyset, T_i\{S/\alpha\}, m + n) \in \mathscr{W}$ for every $i \in I$. $\square$

Type variable instantiation does not affect the subtyping relation:

Proposition B.2. *The following properties hold:*
(1) $T_1 \leqslant T_2$ *implies* $T_1\{S/\alpha\} \leqslant T_2\{S/\alpha\}$;
(2) $t_1 \leqslant t_2$ *implies* $t_1\{S/\alpha\} \leqslant t_2\{S/\alpha\}$.

Proof sketch. Follows from the fact that a free type variable α can only be related to itself. The details are left as an technical exercise. $\square$

We now turn to a series of standard auxiliary results of type preservation under structural congruence and various forms of substitutions.

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

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

Lemma B.2 (type substitution). *If $\Sigma; \Delta, \alpha; \Gamma \vdash P$ and $\emptyset \Vdash S$ and $\|S\| < \infty$, then $\Sigma; \Delta; \Gamma\{S/\alpha\} \vdash P\{S/\alpha\}$.*

Proof sketch. Straightforward induction on the derivation of $\Sigma; \Delta, \alpha; \Gamma \vdash P$, using Propositions B.1 and B.2 wherever necessary. $\square$

Lemma B.3 (value substitution). *If $\Sigma; \Delta; \Gamma, x : t \vdash P$ and $\Gamma + \mathsf{v} : s$ is defined and well formed and $s \leqslant t$, then $\Sigma; \Delta; \Gamma + \mathsf{v} : s \vdash P\{\mathsf{v}/x\}$.*

Proof. By induction on the derivation of $\Sigma; \Delta; \Gamma, x : t \vdash P$ and by cases on the last rule applied. We only show the proof of the (T-SEND) case, the others being simpler or trivial. In the (T-SEND) case we have:

- $P = u!\mathtt{m}\langle S\rangle(v).P'$;
- $\Gamma, x : t = (\Gamma'', u : q \ \{!\mathtt{m}_i\langle\alpha_i\rangle(t_i).T_i\}_{i\in I}) + v : s''$;
- $\Sigma; \Delta; \Gamma'', u : q \ T_k\{S/\alpha_k\} \vdash P'$.

We can assume $x \in \mathtt{dom}(\Gamma'') \cup \{u\}$ for otherwise $x \notin \mathtt{fn}(P')$ and there is nothing left to prove. Let $\Gamma'', u : q \ T_k\{S/\alpha_k\} = \Gamma', x : t'$ for some Γ' and t'. In order to apply the induction hypothesis and deduce $\Sigma; \Delta; \Gamma' + \mathsf{v} : s' \vdash P'\{\mathsf{v}/x\}$, we must find s' such that (a) $s' \leqslant t'$ and (b) $\Gamma' + \mathsf{v} : s'$ is defined and well formed. Observe that the type of x, t', may change from the conclusion to the premise of the rule if $x = u$. We distinguish the following sub-cases:

- $(\mathsf{v} \neq u, u \neq x)$ For (a), we deduce $t' = t$ and we conclude by taking $s' = s$. For (b), then either $\mathsf{v} \notin \mathsf{dom}(\Gamma'')$ or $\mathsf{un}(\Gamma''(\mathsf{v}))$. In both cases we conclude that $\Gamma' + \mathsf{v} : s'$ is defined and well formed.
- $(\mathsf{v} \neq u, u = x)$ For (a), we deduce $t = q \ \{!m_i\langle\alpha_i\rangle(t_i).T_i\}_{i \in I}$. From $s \leqslant t$, we deduce $s = q' \ \{!m_i\langle\alpha_i\rangle(s_i).S_i\}_{i \in I \cup J}$ and $q' \leq q$ and $S_i \leqslant T_i$ for $i \in I$. By Proposition B.2(1) we obtain $S_k\{S/\alpha_k\} \leqslant T_k\{S/\alpha_k\}$ and we conclude by taking $s' = S_k\{S/\alpha_k\}$. For (b) we can reason as for the previous case.
- $(\mathsf{v} = u)$ Since $u \in \mathsf{dom}(\Gamma)$, then $s = \Gamma(u)$ and $q = \mathsf{un}$. Since the un qualifier can only be applied to invariant types, it must be the case that $T_k\{S/\alpha_k\} = s$. We conclude (a) by taking $s' = s$ and (b) follows immediately. $\qquad \square$

Lemma B.4 (weakening). *If $\Sigma; \Delta; \Gamma \vdash P$ and $\mathsf{un}(\Gamma')$, then $\Sigma, \Sigma'; \Delta, \Delta'; \Gamma, \Gamma' \vdash P$.*

Proof. Straightforward induction on the derivation of $\Sigma; \Delta; \Gamma \vdash P$. $\qquad \square$

Lemma B.5 (process substitution). *Let (1) $\Sigma, \{X \mapsto (\Delta; \Gamma)\}; \Delta; \Gamma \vdash Q$. Then (2) $\Sigma, \{X \mapsto (\Delta; \Gamma)\}, \Sigma'; \Delta'; \Gamma' \vdash P$ implies $\Sigma, \Sigma'; \Delta'; \Gamma' \vdash P\{\mathtt{rec}\ X.Q/X\}$.*

Proof. By induction on P. Whenever we encounter some bound name/type variable/process variable in P we assume, without loss of generality, that it does not occur free in Q:

- $(P = \mathbf{0})$ Then $P\{\mathtt{rec}\ X.Q/X\} = \mathbf{0}$. From (2) and (T-IDLE) we deduce $\mathsf{un}(\Gamma')$. We conclude with an application of (T-IDLE).
- $(P = X)$ Then $P\{\mathtt{rec}\ X.Q/X\} = \mathtt{rec}\ X.Q$. From (2) and (T-VAR) we deduce:
  - $\Delta' = \Delta, \Delta''$;
  - $\Gamma' = \Gamma, \Gamma''$;
  - $\mathsf{un}(\Gamma'')$.

  From (1) and Lemma B.4 we obtain $\Sigma, \{X \mapsto (\Delta; \Gamma)\}, \Sigma'; \Delta'; \Gamma' \vdash Q$. We conclude with an application of (T-REC).
- $(P = Y \neq X)$ Then $P\{\mathtt{rec}\ X.Q/X\} = Y$ and we conclude immediately from (T-VAR).
- $(P = \mathtt{close}(u))$ Then $P\{\mathtt{rec}\ X.Q/X\} = \mathtt{close}(u)$ and we conclude immediately from (T-CLOSE).
- $(P = P_1 \oplus P_2)$ Then $P\{\mathtt{rec}\ X.Q/X\} = P_1\{\mathtt{rec}\ X.Q/X\} \oplus P_2\{\mathtt{rec}\ X.Q/X\}$. From (2) and (T-CHOICE) we deduce:
  - $\Sigma, \{X \mapsto (\Delta; \Gamma)\}, \Sigma'; \Delta'; \Gamma' \vdash P_i$ for $i = 1, 2$.

    By induction hypothesis we obtain:
  - $\Sigma, \Sigma'; \Delta'; \Gamma' \vdash P_i\{\mathtt{rec}\ X.Q/X\}$ for $i = 1, 2$.

    We conclude with an application of (T-CHOICE).
- $(P = P_1 \mid P_2)$ Then $P\{\mathtt{rec}\ X.Q/X\} = P_1\{\mathtt{rec}\ X.Q/X\} \mid P_2\{\mathtt{rec}\ X.Q/X\}$. From (2) and (T-PAR) we deduce
  - $\Gamma' = \Gamma_1 + \Gamma_2$ and
  - $\Sigma, \{X \mapsto (\Delta; \Gamma)\}, \Sigma'; \Delta'; \Gamma_i \vdash P_i$ for $i = 1, 2$.

    By induction hypothesis:
  - $\Sigma, \Sigma'; \Delta'; \Gamma_i \vdash P_i\{\mathtt{rec}\ X.Q/X\}$.

    We conclude with an application of (T-PAR).
- $(P = \mathtt{open}(a : T, b : S).P')$ Then $P\{\mathtt{rec}\ X.Q/X\} = \mathtt{open}(a : T, b : S).(P'\{\mathtt{rec}\ X.Q/X\})$. From (2) and (T-OPEN LINEAR CHANNEL) we deduce:
  - $\Delta' \Vdash T$;
  - $\Sigma, \{X \mapsto (\Delta; \Gamma)\}, \Sigma'; \Delta'; \Gamma', a : \mathsf{lin}\ T, b : \mathsf{lin}\ S \vdash P'$;
  - $S = \overline{T}$.

By induction hypothesis:

– $\Sigma, \Sigma'; \Delta'; \Gamma', a : \mathsf{lin}\ T, b : \mathsf{lin}\ S \vdash P'\{\texttt{rec}\ X.Q/X\}$.
  We conclude with an application of (T-Open Linear Channel).

- $(P = \mathsf{open}(a : T).P')$ Similar to the previous case.
- $(P = \sum_{i \in I} u?\mathtt{m}_i\langle\alpha_i\rangle(x_i : s_i).P_i)$ Then

$$P\{\texttt{rec}\ X.Q/X\} = \sum_{i \in I} u?\mathtt{m}_i\langle\alpha_i\rangle(x_i : t_i).(P_i\{\texttt{rec}\ X.Q/X\}).$$

From (2) and (T-Receive) we deduce:

– $\Gamma' = \Gamma'', u : \mathsf{lin}\ \{?\mathtt{m}_i\langle\alpha_i\rangle(s_i).T_i\}_{i\in J}$;
– $J \subseteq I$;
– $s_i \leqslant t_i$ for every $i \in J$;
– $\Sigma, \{X \mapsto (\Delta; \Gamma)\}, \Sigma'; \Delta', \alpha_i; \Gamma', u : \mathsf{lin}\ T_i, x_i : t_i \vdash P_i$ for every $i \in J$.
  By induction hypothesis:
– $\Sigma, \Sigma'; \Delta', \alpha_i; \Gamma', u : \mathsf{lin}\ T_i, x_i : t_i \vdash P_i\{\texttt{rec}\ X.Q/X\}$ for $i \in J$.
  We conclude with an application of (T-Receive).

- $(P = u!\mathtt{m}\langle S\rangle(v).P')$ Then $P\{\texttt{rec}\ X.Q/X\} = u!\mathtt{m}\langle S\rangle(v).(P'\{\texttt{rec}\ X.Q/X\})$. From (2) and (T-Send) we deduce:

– $\Gamma' = (\Gamma'', u : q\ \{!\mathtt{m}_i\langle\alpha_i\rangle(t_i).T_i\}_{i\in I}) + v : s$;
– $\Delta' \Vdash S$;
– $\mathtt{m} = \mathtt{m}_k$ for some $k \in I$;
– $s \leqslant t_k\{S/\alpha_k\}$;
– $\max\{\|S\|_\Delta, \|s\|_\Delta\} < \infty$;
– $\Sigma, \{X \mapsto (\Delta; \Gamma)\}, \Sigma'; \Delta'; \Gamma'', u : q\ T_k\{S/\alpha_k\} \vdash P'$.
  By induction hypothesis:
– $\Sigma, \Sigma'; \Delta'; \Gamma'', u : \mathsf{lin}\ T'\{S/\alpha\} \vdash P'\{\texttt{rec}\ X.Q/X\}$.
  We conclude with an application of (T-Send).

- $(P = \texttt{rec}\ Y.P')$ Then $P\{\texttt{rec}\ X.Q/X\} = \texttt{rec}\ Y.(P'\{\texttt{rec}\ X.Q/X\})$. From (2) and (T-Rec) we deduce:

– $\Sigma, \{X \mapsto (\Delta; \Gamma), Y \mapsto (\Delta'; \Gamma')\}, \Sigma'; \Delta'; \Gamma' \vdash P'$;
– (t2) $\mathsf{dom}(\Gamma'|_{\mathsf{lin}}) \subseteq \mathtt{fn}(P')$.
  By induction hypothesis:
– $\Sigma, \{Y \mapsto (\Delta'; \Gamma')\}, \Sigma'; \Delta'; \Gamma' \vdash P'\{\texttt{rec}\ X.Q/X\}$;
  From (t2) and by definition of process substitution:
– $\mathsf{dom}(\Gamma'|_{\mathsf{lin}}) \subseteq \mathtt{fn}(P') \subseteq \mathtt{fn}(P'\{\texttt{rec}\ X.Q/X\})$.
  We conclude with an application of (T-Rec). $\square$

The following lemma serves as a slight generalization of subject reduction (Theorem 5.1). Note that the last condition $\Gamma|_{\mathsf{un}} \subseteq \Gamma'|_{\mathsf{un}}$ implies that unrestricted values can only accumulate (they are never removed from the type environment) and furthermore their type does not change over time.

Lemma B.6. *Let (1) $\Gamma_0; \Gamma_R, \Gamma \vdash \mu$ where $\mathsf{lin}(\Gamma_R)$ and (2) $\Gamma \vdash P$ and $(\mu; P) \to (\mu'; P')$. Then $\Gamma_0'; \Gamma_R, \Gamma' \vdash \mu'$ and $\Gamma' \vdash P'$ for some Γ_0' and Γ' such that $\Gamma|_{\mathsf{un}} \subseteq \Gamma'|_{\mathsf{un}}$.*

Proof. By induction on the derivation of $(\mu; P) \to (\mu'; P')$ and by cases on the last rule applied.

- (R-Open Linear Channel) In this case:
– $P = \mathsf{open}(a : T, b : S).P'$;

- $\mu' = \mu, a \mapsto [b, \varepsilon], b \mapsto [a, \varepsilon]$.

From the hypothesis (2) and rule (T-Open Linear Channel) we obtain:

- $\emptyset \Vdash T$;
- $S = \overline{T}$;
- $\Gamma, a : \text{lin } T, b : \text{lin } \overline{T} \vdash P'$.

From Proposition 3.1(1) we deduce:

- $\emptyset \Vdash S$.

We conclude by taking $\Gamma_0' = \Gamma_0$ and $\Gamma' = \Gamma, a : \text{lin } T, b : \text{lin } \overline{T}$. The proof that $\Gamma_0'; \Gamma_R, \Gamma' \vdash \mu'$ is trivial and $\Gamma|_{\text{un}} = \Gamma'|_{\text{un}}$.

- (R-Open Unrestricted Channel) Similar to the previous case, except that a fresh unrestricted pointer is added to Γ'.
- (R-Choice Left/Right) Trivial.
- (R-Send Linear) In this case:
  - $P = a!\text{m}\langle S \rangle(\text{v}).P'$;
  - $\mu = \mu'', a \mapsto [b, \mathfrak{Q}], b \mapsto [a, \mathfrak{Q}']$;
  - $\mu' = \mu'', a \mapsto [b, \mathfrak{Q}], b \mapsto [a, \mathfrak{Q}' :: \text{m}\langle S \rangle(\text{v})]$.

  From the hypothesis (2) and rule (T-Send) we obtain:
  - (t1) $\Gamma = (\Gamma'', a : \text{lin } \{!\text{m}_i\langle \alpha_i \rangle(t_i).T_i\}_{i \in I}) + \text{v} : s$;
  - $\emptyset \Vdash S$;
  - $\text{m} = \text{m}_k$ for some $k \in I$;
  - $s \leqslant t_k\{S/\alpha_k\}$;
  - $\|S\| < \infty$ and $\|s\| < \infty$;
  - $\Gamma'', a : T_k\{S/\alpha_k\} \vdash P'$.

  Let $\Gamma_0' = \Gamma_0 + (\text{v} : s)|_{\text{lin}}$ and $\Gamma' = (\Gamma'', a : \text{lin } T_k\{S/\alpha_k\}) + (\text{v} : s)|_{\text{un}}$. Since $\Gamma|_{\text{un}} = \Gamma'|_{\text{un}}$ we only have to show that $\Gamma_0'; \Gamma_R, \Gamma' \vdash \mu'$.

  We prove the items of Definition 5.2 in order.

  (1) We only need to show that $\mathfrak{Q}$ is empty. Suppose by contradiction that this is not the case. Then the endpoint type associated with a before the reduction occurs must begin with an external choice, which contradicts (t1).

  (2) Let $\mathfrak{Q}' = \text{m}_1\langle T_1 \rangle(\text{v}_1) :: \cdots :: \text{m}_n\langle T_n \rangle(\text{v}_n)$. From hypothesis (1) and (t1) we deduce $\Gamma_0, \Gamma_R, \Gamma \vdash b : \text{lin } T_b$ and $\Gamma_0, \Gamma_R, \Gamma \vdash \text{v}_i : s_i$ where

  $$\{?\text{m}_i\langle \alpha_i \rangle(t_i).\overline{T_i}\}_{i \in I} = \overline{\{!\text{m}_i\langle \alpha_i \rangle(t_i).T_i\}_{i \in I}} = \text{tail}(T_b, \text{m}_1\langle T_1 \rangle(s_1) \cdots \text{m}_n\langle T_n \rangle(s_n))$$

  and by Proposition 3.1(3) we conclude

  $$\overline{T_k\{S/\alpha_k\}} = \overline{T_k}\{S/\alpha_k\} = \text{tail}(T_b, \text{m}_1\langle T_1 \rangle(s_1) \cdots \text{m}_n\langle T_n \rangle(s_n)\text{m}\langle S \rangle(s))\,.$$

  (3) Immediate from hypothesis (1).

  (4) From hypothesis (1) we have $\text{dom}(\mu) = \text{dom}(\Gamma_0, \Gamma_R, \Gamma|_{\text{lin}})$ and for every $a' \in \text{dom}(\mu)$ there exists $b' \in \text{dom}(\Gamma_R, \Gamma)$ such that $a' \preccurlyeq_\mu b'$. Clearly $\text{dom}(\mu') = \text{dom}(\Gamma_0', \Gamma_R, \Gamma'|_{\text{lin}})$ since $\text{dom}(\mu') = \text{dom}(\mu)$ and $\text{dom}(\Gamma_0') \cup \text{dom}(\Gamma') = \text{dom}(\Gamma_0) \cup \text{dom}(\Gamma)$. Let $b \preccurlyeq_\mu b_0$ and $\Gamma_R, \Gamma \vdash b_0 : T_0$ and assume $\text{v} \in \text{Pointers}$. We have $\text{v} \prec_{\mu'} b \preccurlyeq_{\mu'} b_0$, namely $\text{v} \preccurlyeq_{\mu'} b_0$. Now

  $$\|s\| < \|\text{tail}(T_b, \text{m}_1\langle T_1 \rangle(s_1) \cdots \text{m}_n\langle T_n \rangle(s_n))\| \leq \|T_b\| \leq \|T_0\|$$

  therefore $\text{v} \neq b_0$. We conclude $b_0 \in \text{dom}(\Gamma_R, \Gamma')$.

  (5) Immediate from hypothesis (1).

- (R-Send Unrestricted) In this case:
  - $P = \overline{a}!\text{m}\langle S \rangle(\text{v}).P'$;

- $\mu = \mu'', a \mapsto [a, \mathfrak{Q}];$
- $\mu' = \mu'', a \mapsto [a, \mathfrak{Q} :: \mathtt{m}\langle S \rangle(\mathsf{v})].$

From the hypothesis (2) and rule (T-SEND) we obtain:

- (t1) $\Gamma = (\Gamma'', \overline{a} : \mathsf{un}\ T) + \mathsf{v} : s$ where $T = \{!\mathtt{m}_i \langle \alpha_i \rangle (t_i).T\}_{i \in I};$
- $\emptyset \Vdash S;$
- $\mathtt{m} = \mathtt{m}_k$ for some $k \in I;$
- $s \leqslant t_k \{S / \alpha_k\};$
- $\|S\| < \infty$ and $\|s\| < \infty;$
- $\Gamma'', a : \mathsf{un}\ T \vdash Q.$

Let $\Gamma_0' = \Gamma_0 + (\mathsf{v} : s)|_{\mathsf{lin}}$ and $\Gamma' = (\Gamma'', \overline{a} : \mathsf{un}\ T) + (\mathsf{v} : s)|_{\mathsf{un}}.$ Since $\Gamma|_{\mathsf{un}} = \Gamma'|_{\mathsf{un}}$ we only have to show that $\Gamma_0'; \Gamma_R, \Gamma' \vdash \mu'.$

We prove the items of Definition 5.2 in order.

(1) Trivial since no queue of linear endpoint was affected by the reduction.

(2) Ditto.

(3) Let $\mathfrak{Q} = \mathtt{m}_1 \langle T_1 \rangle(\mathsf{v}_1) :: \cdots :: \mathtt{m}_n \langle T_n \rangle(\mathsf{v}_n)$ and $\Gamma_0, \Gamma_R, \Gamma \vdash a : \mathsf{lin}\ T_a$ and $\Gamma_0, \Gamma_R, \Gamma \vdash \mathsf{v}_i : s_i.$ We deduce

$$\overline{\{?\mathtt{m}_i \langle \alpha_i \rangle (t_i).\overline{T}\}_{i \in I}} = \overline{\{!\mathtt{m}_i \langle \alpha_i \rangle (t_i).T\}_{i \in I}} = \mathtt{tail}(T_a, \mathtt{m}_1 \langle T_1 \rangle(s_1) \cdots \mathtt{m}_n \langle T_n \rangle(s_n))$$

and we conclude

$$\overline{T} = \mathtt{tail}(T_a, \mathtt{m}_1 \langle T_1 \rangle(s_1) \cdots \mathtt{m}_n \langle T_n \rangle(s_n) \mathtt{m}\langle S \rangle(s)).$$

(4) Analogous to the case (R-SEND LINEAR) with T_a in place of $T_b.$

(5) Immediate from hypothesis (1).

- (R-RECEIVE) In this case:
  - $P = \sum_{i \in I} a?\mathtt{m}_i \langle \alpha_i \rangle (x_i : t_i).P_i;$
  - $\mu = \mu'', a \mapsto [b, \mathtt{m}\langle S \rangle(\mathsf{v}) :: \mathfrak{Q}]$ where $\mathfrak{Q} = \mathtt{m}_1 \langle S_1 \rangle(\mathsf{v}_1) :: \cdots :: \mathtt{m}_n \langle S_n \rangle(\mathsf{v}_n);$
  - $\mathtt{m} = \mathtt{m}_k$ for some $k \in I;$
  - $P' = P_k \{S / \alpha_k\}\{\mathsf{v} / x_k\};$
  - $\mu' = \mu'', a \mapsto [b, \mathfrak{Q}].$

From the hypothesis (2) and rule (T-RECEIVE) we obtain:

- $\Gamma = \Gamma'', a : \mathsf{lin}\ \{?\mathtt{m}_i \langle \alpha_i \rangle (s_i).T_i\}_{i \in J}$ with $J \subseteq I;$
- $s_k \leqslant t_k;$
- (t3) $\alpha_k; \Gamma'', a : \mathsf{lin}\ T_k, x_k : t_k \vdash P_k$

Let $\Gamma_0, \Gamma_R, \Gamma \vdash \mathsf{v} : s.$ From hypothesis (1) and Proposition B.2 we obtain:

- (c1) $\emptyset \Vdash S$ and $\|S\| < \infty;$
- (c2) $s \leqslant s_k \{S / \alpha_k\} \leqslant t_k \{S / \alpha_k\}.$

From hypothesis (1) we also deduce that:

- (f1) if $\mathsf{un}(s)$, then $\mathsf{v} \in \mathsf{dom}(\Gamma)$ and $\Gamma \vdash \mathsf{v} : s$, because all the unrestricted values are in $\Gamma;$
- (f2) if $\mathsf{lin}(s)$, then $\mathsf{v} \notin \mathsf{dom}(\Gamma)$, because $\mathsf{v} \prec_\mu a$ and therefore it must be $\mathsf{v} \in \mathsf{dom}(\Gamma_0)$ (process isolation prevents a from being reachable from any pointer in $\mathsf{dom}(\Gamma_R, \Gamma)$ and different from a).

From (t3), (c1), and Lemma B.2 we have:

- (t3') $\Gamma'' \{S / \alpha_k\}, a : q\ T_k \{S / \alpha_k\}, x_k : t_k \{S / \alpha_k\} \vdash P_k \{S / \alpha_k\}.$

From (f1) and (f2) we deduce that $\Gamma_0 = \Gamma_0', (\mathsf{v} : s)|_{\mathsf{lin}}$ for some $\Gamma_0'.$ Take $\Gamma' = (\Gamma'', a : q\ T_k \{S / \alpha_k\}) + \mathsf{v} : s$ and observe that Γ' is well defined by (f1) and (f2) and also $\Gamma|_{\mathsf{un}} \subseteq \Gamma'|_{\mathsf{un}}$ by construction of $\Gamma'.$ From (t3'), (c2), and Lemma B.3 we conclude:

$-\ \Gamma' \vdash P_k\{S/\alpha_k\}\{\mathsf{v}/x_k\}$

We have to show $\Gamma'_0, \Gamma_R, \Gamma' \vdash \mu'$ and we prove the items of Definition 5.2 in order.

(1) If $a = b$ there is nothing to prove. Suppose $a \neq b$. Since the queue associated with a is not empty in μ, the queue associated with its peer endpoint b must be empty. The reduction does not change the queue associated with b, therefore condition (1) of Definition 5.2 is satisfied.

(2) Suppose $a \neq b$ for otherwise there is nothing to prove. From hypothesis (1) we deduce $\Gamma_0, \Gamma_R, \Gamma \vdash b : \mathsf{lin}\ T_b$ and

$$\begin{aligned}
\overline{T_b} &= \mathtt{tail}(\{?\mathtt{m}_i\langle\alpha_i\rangle(s_i).T_i\}_{i\in J}, \mathtt{m}\langle S\rangle(s)\mathtt{m}_1\langle S_1\rangle(s'_1)\cdots\mathtt{m}_n\langle S_n\rangle(s'_n)) \\
&= \mathtt{tail}(T_k\{S/\alpha_k\}, \mathtt{m}_1\langle S_1\rangle(s'_1)\cdots\mathtt{m}_n\langle S_n\rangle(s'_n))
\end{aligned}$$

where $\Gamma_0, \Gamma_R, \Gamma \vdash \mathsf{v}_i : s'_i$ for $1 \leq i \leq n$.

(3) Similar to the previous item, where $a = b$.

(4) Straightforward by definition of Γ'_0 and Γ'.

(5) Immediate from hypothesis (1).

- (R-PAR) In this case:
  - $P = P_1 \mid P_2$;
  - $(\mu; P_1) \rightarrow (\mu'; P'_1)$;
  - $P' = P'_1 \mid P_2$.

  From the hypothesis (2) and rule (T-PAR) we obtain:
  - $\Gamma = \Gamma_1 + \Gamma_2$;
  - $\Gamma_i \vdash P_i$ for $i \in \{1, 2\}$.

  In particular, from Lemma B.4 we have:
  - $(\Gamma_0; \Gamma_R, \Gamma_2|_{\mathsf{lin}}, \Gamma_1) + \Gamma_2|_{\mathsf{un}} \vdash \mu$;
  - $\Gamma_1 + \Gamma_2|_{\mathsf{un}} \vdash P_1$.

  By induction hypothesis we deduce that there exist Γ'_0 and Γ'_1 such that:
  - $(\Gamma_1 + \Gamma_2|_{\mathsf{un}})|_{\mathsf{un}} = (\Gamma_1 + \Gamma_2)|_{\mathsf{un}} \subseteq \Gamma'_1|_{\mathsf{un}}$;
  - $\Gamma'_0; \Gamma_R, \Gamma_2|_{\mathsf{lin}}, \Gamma'_1 \vdash \mu'$;
  - $\Gamma'_1 \vdash P'_1$.

  Now $\Gamma_2|_{\mathsf{lin}}, \Gamma'_1 = \Gamma_2|_{\mathsf{lin}}, (\Gamma'_1 + \Gamma_2|_{\mathsf{un}}) = \Gamma'_1 + \Gamma_2$. Therefore, from rule (T-PAR) we obtain $\Gamma'_1 + \Gamma_2 \vdash P'$. We conclude by taking $\Gamma' = \Gamma'_1 + \Gamma_2$.

- (R-REC) In this case:
  - $P = \mathtt{rec}\ X.Q$;
  - $P' = Q\{P/X\}$;
  - $\mu' = \mu$.

  From the hypothesis (2) and rule (T-REC) we obtain:
  - (t3) $\{X \mapsto (\emptyset; \Gamma)\}; \emptyset; \Gamma \vdash Q$;
  - $\mathsf{dom}(\Gamma|_{\mathsf{lin}}) \subseteq \mathtt{fn}(Q)$.

  From (t3) and Lemma B.5 we obtain:
  - $\Gamma \vdash P'$.

  We conclude by taking $\Gamma'_0 = \Gamma_0$ and $\Gamma' = \Gamma$.

- (R-STRUCT) Follows from Lemma B.1 and induction. □

We conclude with the proofs of subject reduction and soundness.

Theorem B.1 (Theorem 5.1). *Let* $\Gamma_0; \Gamma \vdash (\mu; P)$ *and* $(\mu; P) \rightarrow (\mu'; P')$. *Then* $\Gamma'_0; \Gamma' \vdash (\mu'; P')$ *for some* Γ'_0 *and* Γ'.

Proof. Follows from Lemma B.6 by taking $\Gamma_R = \emptyset$. □

Proposition B.3. *Let* $\Gamma \vdash P$. *Then* $\mathtt{fn}(P) \subseteq \mathtt{dom}(\Gamma)$ *and* $\mathtt{dom}(\Gamma|_{\mathsf{lin}}) \subseteq \mathtt{fn}(P)$.

Proof. From the hypothesis $\Gamma \vdash P$ we deduce that P is closed with respect to process variables. The results follows by a straightforward induction on the derivation of $\Gamma \vdash P$ and by cases on the last rule applied, where the case for (T-Var) is impossible by hypothesis and (T-Rec) is a base case when proving the second inclusion. $\qquad\square$

Theorem B.2 (Theorem 5.2). *Let* $\vdash P$. *Then* P *is well behaved.*

Proof. Consider a derivation $(\emptyset; P) \Rightarrow (\mu; Q)$. From Theorem 5.1 we deduce (*) $\Gamma_0; \Gamma \vdash (\mu; Q)$ for some Γ_0 and Γ. We prove conditions (1–3) of Definition 4.2 in order:

(1) Using Proposition B.3, Definition 4.1, and Definition 5.2 we have $\mathtt{reach}(\mathtt{fn}(Q), \mu) \subseteq \mathtt{reach}(\mathtt{dom}(\Gamma), \mu) = \mathtt{dom}(\mu)$ and $\mathtt{dom}(\mu) = \mathtt{reach}(\mathtt{dom}(\Gamma), \mu) = \mathtt{reach}(\mathtt{dom}(\Gamma|_{\mathsf{lin}}), \mu) \subseteq \mathtt{reach}(\mathtt{fn}(Q), \mu)$.

(2) Suppose $Q \equiv P_1 \mid P_2$. By Lemma B.1 we deduce $\Gamma \vdash P_1 \mid P_2$, namely there exist Γ_1 and Γ_2 such that $\Gamma = \Gamma_1 + \Gamma_2$ and $\Gamma_i \vdash P_i$. From the definition of $\Gamma_1 + \Gamma_2$ we deduce $\mathtt{dom}(\Gamma_1|_{\mathsf{lin}}) \cap \mathtt{dom}(\Gamma_2|_{\mathsf{lin}}) = \emptyset$. From Proposition B.3 we have $\mathtt{fn}(P_i) \subseteq \mathtt{dom}(\Gamma_i)$ for $i \in \{1, 2\}$. From (*) we conclude $\mathtt{reach}(\mathtt{fn}(P_1), \mu) \cap \mathtt{reach}(\mathtt{fn}(P_2), \mu) \subseteq \mathtt{reach}(\mathtt{dom}(\Gamma_1), \mu) \cap \mathtt{reach}(\mathtt{dom}(\Gamma_2), \mu) = \mathtt{reach}(\mathtt{dom}(\Gamma_1|_{\mathsf{lin}}), \mu) \cap \mathtt{reach}(\mathtt{dom}(\Gamma_2|_{\mathsf{lin}}), \mu) = \emptyset$.

(3) Suppose $Q \equiv P' \mid Q'$ where P' has no unguarded parallel composition and $(\mu; Q) \not\rightarrow$. Then P' contains no unfolded recursion, choice, `open`, output prefix that is not guarded by an input prefix, for all these processes reduce. In the case of output prefixes, one uses $\Gamma_0; \Gamma \vdash \mu$ to deduce that either (R-Send Linear) or (R-Send Unrestricted) can be applied. Suppose $P' \neq \mathbf{0}$. Then either $P' = \mathtt{close}(a)$ or $P' = \sum_{i \in I} a?\mathtt{m}_i\langle\alpha_i\rangle(x_i : t_i).P_i$. Suppose by contradiction that the queue associated with a is not empty, namely that $a \mapsto [b, \mathtt{m}\langle S\rangle(\mathtt{v}) :: \mathfrak{Q}] \in \mu$. From the hypothesis $\Gamma \vdash \mu$ we deduce that the endpoint type associated with a cannot be `end`, and therefore $P' \neq \mathtt{close}(a)$. From the hypothesis $\Gamma \vdash Q$ and rule (T-Receive) we deduce $\Gamma \vdash a : \mathsf{lin}\ \{?\mathtt{m}_i\langle\alpha_i\rangle(s_i).T_i\}_{i \in J}$ and $J \subseteq I$. From the hypothesis $\Gamma \vdash \mu$ we deduce $\mathtt{m} = \mathtt{m}_k$ for some $k \in J$, namely $(\mu; P') \rightarrow$, which is absurd. $\qquad\square$

Appendix C. Supplement to Section 6

C.1. **Subtyping.** In order to prove correctness and completeness of the subtyping algorithm (Definition 6.2) with respect to the subtyping relation (Definition 3.2) we need a few more concepts. The first one is that of *trees* of an endpoint type T, which is the set of all subtrees of T where recursive terms have been infinitely unfolded. We build this set inductively, as follows:

Definition C.1 (endpoint type trees). We write $\mathtt{trees}(T)$ for the least set such that:

- $T \in \mathtt{trees}(T)$;
- $\mathtt{rec}\ \alpha.S \in \mathtt{trees}(T)$ implies $S\{\mathtt{rec}\ \alpha.S/\alpha\} \in \mathtt{trees}(T)$;
- $\{\dagger\mathtt{m}_i\langle\alpha_i\rangle(q_i\ S_i).T_i\}_{i \in I} \in \mathtt{trees}(T)$ where $\dagger \in \{?, !\}$ implies $S_i \in \mathtt{trees}(T)$ and $T_i \in \mathtt{trees}(T)$ for every $i \in I$.

Observe that $\mathtt{trees}(T)$ is *finite* for every T, because the infinite unfolding of an endpoint type is a regular tree [7]. Also, every free type variable in $S \in \mathtt{trees}(T)$ is either free in T or it is bound by a prefix of T. In particular, it cannot be bound by a recursion.

The next concept we need is that of *instance* of an endpoint type subtree. The idea is to generate the set of all instances of the (trees of the) endpoint types that the subtyping algorithm visits, and to make sure that this set is finite. Looking at the rules in Table 10 we see that only type variables that are bound in a prefix $\mathsf{m}\langle\alpha\rangle(t)$ are ever instantiated. Also, each variable α in one of the endpoint types can be instantiated with $\mathsf{m}(\alpha,\beta)$ where β is some type variable of the other endpoint type. These considerations lead to the following definition of endpoint type instances:

Definition C.2 (endpoint type instances). Let m be a map as by Definition 6.2. We define $\mathtt{instances}(\mathsf{m}, T, S)$ as the smallest set such that:

- if $T' \in \mathtt{trees}(T)$ and $\{\alpha_1, \dots, \alpha_n\} = \mathtt{ftv}(T') \cap \mathtt{btv}(T)$ and $\{\beta_1, \dots, \beta_n\} \subseteq \mathtt{btv}(S)$, then $T'\{\mathsf{m}(\alpha_1, \beta_1)/\alpha_1\} \cdots \{\mathsf{m}(\alpha_n, \beta_n)/\alpha_n\} \in \mathtt{instances}(\mathsf{m}, T, S)$;
- if $S' \in \mathtt{trees}(S)$ and $\{\beta_1, \dots, \beta_n\} = \mathtt{ftv}(S') \cap \mathtt{btv}(S)$ and $\{\alpha_1, \dots, \alpha_n\} \subseteq \mathtt{btv}(T)$, then $S'\{\mathsf{m}(\beta_1, \alpha_1)/\beta_1\} \cdots \{\mathsf{m}(\beta_n, \alpha_n)/\beta_n\} \in \mathtt{instances}(\mathsf{m}, T, S)$.

Observe that $T, S \in \mathtt{instances}(\mathsf{m}, T, S)$ and that $\mathtt{instances}(\mathsf{m}, T, S)$ is finite, since it contains finitely many instantiations of finitely many subtrees of T and S.

Proposition C.1. *Every endpoint type occurring in the derivation of $\emptyset \vdash_{\mathsf{m}} T \leqslant_{\mathsf{a}} S$ is in* $\mathtt{instances}(T, S)$.

Proof sketch. A simple induction on the derivation of $\emptyset \vdash_{\mathsf{m}} T \leqslant_{\mathsf{a}} S$. □

Lemma C.1. *Let $\emptyset \vdash_{\mathsf{m}} T \leqslant_{\mathsf{a}} S$ and $\{(T, S)\} \vdash_{\mathsf{m}} T' \leqslant_{\mathsf{a}} S'$. Then $\emptyset \vdash_{\mathsf{m}} T' \leqslant_{\mathsf{a}} S'$.*

Proof sketch. A simple induction on the proof of $\{(T, S)\} \vdash_{\mathsf{m}} T' \leqslant_{\mathsf{a}} S'$ where every application of rule (S-Axiom) for the pair (T, S) is replaced by a copy of the proof of $\emptyset \vdash_{\mathsf{m}} T \leqslant_{\mathsf{a}} S$. □

Theorem C.1 (Theorem 6.1). *Let T_0 and S_0 be independent endpoint types and m be a map as by Definition 6.2. Then $\vdash_{\mathsf{m}} T_0 \leqslant_{\mathsf{a}} S_0$ if and only if $T_0 \leqslant S_0$.*

Proof. ($\Rightarrow$) It is enough to show that

$$\mathscr{S} \overset{\text{def}}{=} \{(T, S) \mid \emptyset \vdash_{\mathsf{m}} T \leqslant_{\mathsf{a}} S \ \& \ T, S \in \mathtt{instances}(\mathsf{m}, T_0, S_0)\}$$
$$\cup \{(q\ T, q'\ S) \mid q \leq q' \ \& \ T, S \in \mathtt{instances}(\mathsf{m}, T_0, S_0) \ \& \ \emptyset \vdash_{\mathsf{m}} T \leqslant_{\mathsf{a}} S\}$$

is a coinductive subtyping. Let $(q\ T, q'\ S) \in \mathscr{S}$. Then $q \leq q'$ and $\emptyset \vdash_{\mathsf{m}} T \leqslant_{\mathsf{a}} S$. By definition of $\mathscr{S}$ we conclude $(T, S) \in \mathscr{S}$.

Let $(T, S) \in \mathscr{S}$. Then (J) $\emptyset \vdash_{\mathsf{m}} T \leqslant_{\mathsf{a}} S$. We reason by induction on the number of topmost applications of rules (S-Rec Left) and (S-Rec Right) (which must be finite because of contractivity of endpoint types) and by cases on the first (bottom-up) rule different from (S-Rec Left) and (S-Rec Right) applied for deriving (J), observing that is cannot be (S-Axiom) for the context is initially empty and rules (S-Rec Left) and (S-Rec Right) only add pairs of endpoint types where at least one of them begins with a recursion:

- (S-Rec Left) Then $T \equiv \mathtt{rec}\ \alpha.T'$ and $\{(T, S)\} \vdash_{\mathsf{m}} T'\{T/\alpha\} \leqslant_{\mathsf{a}} S$. From (J) and Lemma C.1 we derive $\emptyset \vdash_{\mathsf{m}} T'\{T/\alpha\} \leqslant_{\mathsf{a}} S$. By induction hypothesis we derive $T'\{T/\alpha\} \leqslant S$ and we conclude by observing that $T = T'\{T/\alpha\}$.
- (S-Rec Right) Symmetric of the previous case.
- (S-Var) Then $T \equiv S \equiv \alpha$ and there is nothing left to prove.
- (S-End) Then $T \equiv S \equiv \mathsf{end}$ and there is nothing left to prove.

- (S-Input) Then $T \equiv \{?m_i\langle\alpha_i\rangle(t_i).T_i\}_{i\in I}$ and $S \equiv \{?m_j\langle\beta_j\rangle(s_j).S_j\}_{j\in I\cup J}$. Let $\gamma_i = m(\alpha_i, \beta_i)$ for $i \in I$. From (S-Input) we deduce:
  - $\{(T, S)\} \vdash_{\mathsf{m}} t_i\{\gamma_i/\alpha_i\} \leqslant_{\mathsf{a}} s_i\{\gamma_i/\beta_i\}$ for every $i \in I$;
  - $\{(T, S)\} \vdash_{\mathsf{m}} T_i\{\gamma_i/\alpha_i\} \leqslant_{\mathsf{a}} S_i\{\gamma_i/\beta_i\}$ for every $i \in I$.

  From Lemma C.1 we derive:
  - $\emptyset \vdash_{\mathsf{m}} t_i\{\gamma_i/\alpha_i\} \leqslant_{\mathsf{a}} s_i\{\gamma_i/\beta_i\}$ for every $i \in I$;
  - $\emptyset \vdash_{\mathsf{m}} T_i\{\gamma_i/\alpha_i\} \leqslant_{\mathsf{a}} S_i\{\gamma_i/\beta_i\}$ for every $i \in I$.

  By definition of $\mathscr{S}$ we know that $T, S \in \mathtt{instances}(m, T_0, S_0)$. For every $i \in I$, we can deduce that $\gamma_i \notin \mathtt{ftv}(t_i) \cup \mathtt{ftv}(T_i) \cup \mathtt{ftv}(s_i) \cup \mathtt{ftv}(S_i)$, because γ_i can only substitute the free occurrences of α_i and of β_i and:
  - α_i is bound in the i-th branch of T and does not occur in S;
  - β_i is bound in the i-th branch of S and does not occur in T.

  Therefore, by alpha conversion we obtain:
  - $T = \{?m_i\langle\gamma_i\rangle(t_i\{\gamma_i/\alpha_i\}).T_i\{\gamma_i/\alpha_i\}\}_{i\in I}$;
  - $S = \{?m_j\langle\gamma_i\rangle(s_i\{\gamma_i/\beta_i\}).S_i\{\gamma_i/\beta_i\}\}_{i\in I} + \{?m_j\langle\beta_j\rangle(s_j).S_j\}_{j\in J\setminus I}$.

  We conclude $(s_i\{\gamma_i/\beta_i\}, t_i\{\gamma_i/\beta_i\}) \in \mathscr{S}$ and $(S_i\{\gamma_i/\beta_i\}, T_i\{\gamma_i/\beta_i\}) \in \mathscr{S}$ by definition of $\mathscr{S}$.

- (S-Output) Analogous to the previous case.

  ($\Leftarrow$) We prove that $T, S \in \mathtt{instances}(m, T_0, S_0)$ and $T \leqslant S$ imply $\mathscr{S} \vdash_{\mathsf{m}} T \leqslant_{\mathsf{a}} S$ by induction on $\mathtt{instances}(m, T, S) \setminus \mathscr{S}$. In the base case we have $(T, S) \in \mathscr{S}$ and we conclude with an application of (T-Axiom). For the inductive case we reason by case analysis on the structure of T and S, knowing that $T \leqslant S$:

- ($T \equiv S \equiv \alpha$) We conclude with an application of (S-Var).
- ($T \equiv S \equiv \mathsf{end}$) We conclude with an application of (S-End).
- ($T \equiv \mathtt{rec}\ \alpha.T'$) Since $T = T'\{T/\alpha\}$ we have $T'\{T/\alpha\} \leqslant S$. By induction hypothesis we know that $\mathscr{S} \cup \{(T, S)\} \vdash_{\mathsf{m}} T'\{T/\alpha\} \leqslant_{\mathsf{a}} S$ is derivable. We conclude with an application of (S-Rec Left).
- ($S \equiv \mathtt{rec}\ \alpha.S'$) Symmetric of the previous case.
- ($T \equiv \{?m_i\langle\alpha_i\rangle(q_i\ T_i').T_i\}_{i\in I}$ and $S \equiv \{?m_j\langle\beta_j\rangle(q_j'\ S_j').S_j\}_{j\in J}$ and $I \subseteq J$) From the hypothesis $T \leqslant S$ we know that for every $i \in I$ there exists γ_i such that $q_i \leqslant q_i'$ and $T_i'\{\gamma_i/\alpha_i\} \leqslant S_i'\{\gamma_i/\beta_i\}$ and $T_i\{\gamma_i/\alpha_i\} \leqslant S_i\{\gamma_i/\beta_i\}$. Since $T, S \in \mathtt{instances}(m, T_0, S_0)$ we know that $\delta_i = m(\alpha_i, \beta_i) \notin \mathtt{ftv}(T_i') \cup \mathtt{ftv}(T_i) \cup \mathtt{ftv}(S_i') \cup \mathtt{ftv}(S_i)$. We deduce $T_i'\{\delta_i/\alpha_i\} \leqslant S_i'\{\delta_i/\beta_i\}$ and $T_i\{\delta_i/\alpha_i\} \leqslant S_i\{\delta_i/\beta_i\}$ for every $i \in I$. By induction hypothesis we derive that $\mathscr{S} \cup \{(T, S)\} \vdash_{\mathsf{m}} T_i' \leqslant_{\mathsf{a}} S_i'$ and $\mathscr{S} \cup \{(T, S)\} \vdash_{\mathsf{m}} T_i \leqslant_{\mathsf{a}} S_i$ are derivable for every $i \in I$. Also, $\mathscr{S} \cup \{(T, S)\} \vdash_{\mathsf{m}} q_i\ T_i' \leqslant_{\mathsf{a}} q_i'\ S_i'$ is derivable with an application of (S-Type) for every $i \in I$. We conclude $\mathscr{S} \vdash_{\mathsf{m}} T \leqslant_{\mathsf{a}} S$ with an application of (S-Input).
- ($T \equiv \{!m_i\langle\alpha_i\rangle(t_i).T_i\}_{i\in I}$ and $S \equiv \{!m_j\langle\beta_j\rangle(s_j).S_j\}_{j\in J}$ and $J \subseteq I$) Analogous to the previous case. $\square$

C.2. **Type Weight.** We begin by proving that the weight algorithm is unaffected by foldings/unfoldings of recursive terms.

Proposition C.2 (Proposition 6.2). $\mathtt{W}(\Delta_0, \emptyset, \mathtt{rec}\ \alpha.T) = \mathtt{W}(\Delta_0, \emptyset, T\{\mathtt{rec}\ \alpha.T/\alpha\})$.

Proof. Let $\mathtt{W}(\Delta_0, \emptyset, \mathtt{rec}\ \alpha.T) = w$. We prove a more general statement, namely that for every S and Δ such that $\mathtt{W}(\Delta_0, \Delta, S) \leq w$ and $\mathtt{btv}(S) \cap \mathtt{ftv}(T) = \emptyset$ we have:

(1) $\alpha \in \Delta$ implies $\mathtt{W}(\Delta_0, \Delta, S) \leq \mathtt{W}(\Delta_0, \Delta \setminus \{\alpha\}, S\{\mathtt{rec}\ \alpha.T/\alpha\}) \leq \max\{w, \mathtt{W}(\Delta_0, \Delta, S)\}$;

(2) $\alpha \notin \Delta$ implies $\mathtt{W}(\Delta_0, \Delta, S) = \mathtt{W}(\Delta_0, \Delta \setminus \{\alpha\}, S\{\mathtt{rec}\ \alpha.T/\alpha\})$.

The statement then follows from (1) by taking $S = T$ and $\Delta = \{\alpha\}$ and noting that $\mathtt{W}(\Delta_0, \emptyset, \mathtt{rec}\ \alpha.T) = \mathtt{W}(\Delta_0, \{\alpha\}, T)$ by definition of algorithmic weight. We proceed by induction on S assuming, without loss of generality, that $(\{\alpha\} \cup \mathtt{ftv}(T)) \cap \mathtt{btv}(S) = \emptyset$:

- ($S \equiv \mathsf{end}$ or $S \equiv \{!\mathtt{m}_i\langle\alpha_i\rangle(t_i)T_i\}_{i\in I}$) Clear as $\mathtt{W}(\Delta_0, \Delta, S) = \mathtt{W}(\Delta_0, \Delta\setminus\{\alpha\}, S\{\mathtt{rec}\ \alpha.T/\alpha\}) = 0$.
- ($S \equiv \alpha$) We have $\mathtt{W}(\Delta_0, \Delta \setminus \{\alpha\}, S\{\mathtt{rec}\ \alpha.T/\alpha\}) = \mathtt{W}(\Delta_0, \Delta \setminus \{\alpha\}, \mathtt{rec}\ \alpha.T) = w$ therefore we conclude:
  (1) $\mathtt{W}(\Delta_0, \Delta, S) = 0 \le w = \mathtt{W}(\Delta_0, \Delta \setminus \{\alpha\}, S\{\mathtt{rec}\ \alpha.T/\alpha\}) = \max\{w, \mathtt{W}(\Delta_0, \Delta, S)\}$;
  (2) $\mathtt{W}(\Delta_0, \Delta, S) = \infty = w = \mathtt{W}(\Delta_0, \Delta \setminus \{\alpha\}, S\{\mathtt{rec}\ \alpha.T/\alpha\})$.
- ($S \equiv \beta \ne \alpha$) Trivial since $\mathtt{W}(\Delta_0, \Delta, S) = \mathtt{W}(\Delta_0, \Delta \setminus \{\alpha\}, S\{\mathtt{rec}\ \alpha.T/\alpha\})$.
- ($S \equiv \mathtt{rec}\ \beta.S'$) By induction hypothesis we deduce:
  (1) $\alpha \in \Delta$ implies

  $$\mathtt{W}(\Delta_0, \Delta\cup\{\beta\}, S') \le \mathtt{W}(\Delta_0, (\Delta\cup\{\beta\})\setminus\{\alpha\}, S'\{\mathtt{rec}\ \alpha.T/\alpha\}) \le \max\{w, \mathtt{W}(\Delta_0, \Delta\cup\{\beta\}, S')\};$$

  (2) $\alpha \notin \Delta$ implies $\mathtt{W}(\Delta_0, \Delta \cup \{\beta\}, S') = \mathtt{W}(\Delta_0, (\Delta \cup \{\beta\}) \setminus \{\alpha\}, S'\{\mathtt{rec}\ \alpha.T/\alpha\})$.
  We conclude by definition of algorithmic weight, since:
  - $\mathtt{W}(\Delta_0, \Delta, S) = \mathtt{W}(\Delta_0, \Delta, \mathtt{rec}\ \beta.S') = \mathtt{W}(\Delta_0, \Delta \cup \{\beta\}, S')$, and
  - $\mathtt{W}(\Delta_0, \Delta \setminus \{\alpha\}, S\{\mathtt{rec}\ \alpha.T/\alpha\}) = \mathtt{W}(\Delta_0, \Delta \setminus \{\alpha\}, (\mathtt{rec}\ \beta.S')\{\mathtt{rec}\ \alpha.T/\alpha\}) = \mathtt{W}(\Delta_0, \Delta \setminus \{\alpha\}, \mathtt{rec}\ \beta.(S'\{\mathtt{rec}\ \alpha.T/\alpha\})) = \mathtt{W}(\Delta_0, (\Delta \cup \{\beta\}) \setminus \{\alpha\}, S'\{\mathtt{rec}\ \alpha.T/\alpha\})$.
- ($S \equiv \{?\mathtt{m}_i\langle\alpha_i\rangle(t_i).T_i\}_{i\in I}$) By induction hypothesis on t_i and T_i for $i \in I$ we deduce:
  (1) $\mathtt{W}(\Delta_0, \emptyset, t_i) = \mathtt{W}(\Delta_0, \emptyset, t_i\{\mathtt{rec}\ \alpha.T/\alpha\})$;
  (2) $\alpha \in \Delta$ implies

  $$\mathtt{W}(\Delta_0, \Delta \setminus \{\alpha_i\}, T_i) \le \mathtt{W}(\Delta_0, \Delta \setminus \{\alpha_i, \alpha\}, T_i\{\mathtt{rec}\ \alpha.T/\alpha\}) \le \max\{w, \mathtt{W}(\Delta_0, \Delta \setminus \{\alpha_i\}, T_i)\};$$

  (3) $\alpha \notin \Delta$ implies $\mathtt{W}(\Delta_0, \Delta \setminus \{\alpha_i\}, T_i) = \mathtt{W}(\Delta_0, \Delta \setminus \{\alpha_i, \alpha\}, T_i\{\mathtt{rec}\ \alpha.T/\alpha\})$.
  If $\alpha \in \Delta$ we conclude:

$$\begin{aligned}
\mathtt{W}(\Delta_0, \Delta, S) &= \max\{1 + \mathtt{W}(\Delta_0, \emptyset, t_i), \mathtt{W}(\Delta_0, \Delta \setminus \{\alpha_i\}, T_i)\}_{i\in I} \\
&= \max\{1 + \mathtt{W}(\Delta_0, \emptyset, t_i\{\mathtt{rec}\ \alpha.T/\alpha\}), \mathtt{W}(\Delta_0, \Delta \setminus \{\alpha_i\}, T_i)\}_{i\in I} \\
&\le \max\{1 + \mathtt{W}(\Delta_0, \emptyset, t_i\{\mathtt{rec}\ \alpha.T/\alpha\}), \mathtt{W}(\Delta_0, \Delta \setminus \{\alpha_i, \alpha\}, T_i\{\mathtt{rec}\ \alpha.T/\alpha\})\}_{i\in I} \\
&= \mathtt{W}(\Delta_0, \Delta \setminus \{\alpha\}, S\{\mathtt{rec}\ \alpha.T/\alpha\}) \\
&\le \max\{1 + \mathtt{W}(\Delta_0, \emptyset, t_i\{\mathtt{rec}\ \alpha.T/\alpha\}), w, \mathtt{W}(\Delta_0, \Delta \setminus \{\alpha_i\}, T_i)\}_{i\in I} \\
&= \max\{1 + \mathtt{W}(\Delta_0, \emptyset, t_i), w, \mathtt{W}(\Delta_0, \Delta \setminus \{\alpha_i\}, T_i)\}_{i\in I} \\
&= \max\{w, \mathtt{W}(\Delta_0, \Delta, S)\}
\end{aligned}$$

If $\alpha \notin \Delta$ we conclude:

$$\begin{aligned}
\mathtt{W}(\Delta_0, \Delta, S) &= \max\{1 + \mathtt{W}(\Delta_0, \emptyset, t_i), \mathtt{W}(\Delta_0, \Delta \setminus \{\alpha_i\}, T_i)\}_{i\in I} \\
&= \max\{1 + \mathtt{W}(\Delta_0, \emptyset, t_i\{\mathtt{rec}\ \alpha.T/\alpha\}), \mathtt{W}(\Delta_0, \Delta \setminus \{\alpha_i, \alpha\}, T_i\{\mathtt{rec}\ \alpha.T/\alpha\})\}_{i\in I} \\
&= \mathtt{W}(\Delta_0, \Delta \setminus \{\alpha\}, S\{\mathtt{rec}\ \alpha.T/\alpha\}).
\end{aligned}$$

$\square$

The next lemma states that, if the weight algorithm determines a weight n for some endpoint type T, then n is a weight bound for T.

Lemma C.2. *If* $\mathtt{W}(\Delta, \emptyset, T) = n \in \mathbb{N}$, *then* $\Delta \vdash T :: n$.

Proof. It is enough to show that

$$\mathscr{W} \stackrel{\mathrm{def}}{=} \{(\Delta, T, n) \mid \mathtt{W}(\Delta, \emptyset, T) \le n \in \mathbb{N}\}$$

is a coinductive weight bound. Let $(\Delta, T, n) \in \mathscr{W}$. Then (h) $\mathtt{W}(\Delta, \emptyset, T) \leq n \in \mathbb{N}$. Without loss of generality we may assume that T does *not* begin with a recursion. If this were not the case, by contractivity of endpoint types we have $T = T'$ where T' does not begin with a recursion. Now, by Proposition 6.2 we deduce $\mathtt{W}(\Delta, \emptyset, T') = \mathtt{W}(\Delta, \emptyset, T) = n$ and therefore $(\Delta, T', n) \in \mathscr{W}$ by definition of $\mathscr{W}$.

We reason by cases on T:

- $(T \equiv \mathsf{end}$ or $T \equiv \{!\mathtt{m}_i\langle\alpha_i\rangle(t_i).T_i\}_{i \in I})$ There is nothing to prove.
- $(T \equiv \alpha)$ From (h) we deduce $\alpha \in \Delta$ and there nothing left to prove.
- $(T \equiv \{?\mathtt{m}_i\langle\alpha_i\rangle(q_i \; S_i).T_i\}_{i \in I})$ Then $0 < \mathtt{W}(\Delta, \emptyset, T) \leq n$. From (h) we deduce $\mathtt{W}(\Delta, \emptyset, S_i) \leq n-1$ and $\mathtt{W}(\Delta, \emptyset, T_i) \leq n$ for every $i \in I$. We conclude $(\Delta, S_i, n-1) \in \mathscr{W}$ and $(\Delta, T_i, n) \in \mathscr{W}$ for every $i \in I$. $\qquad\square$

The last auxiliary result proves that the weight algorithm computes the least upper weight bound for an endpoint type. We use σ to range over arbitrary substitutions of endpoint types in place of type variables, we write $T\sigma$ for T where the substitutions in σ have been applied, and $\mathtt{dom}(\sigma)$ for the domain of σ (the set of type variables that are instantiated).

Lemma C.3. *If* $\Delta \vdash T\sigma :: n$*, then* $\mathtt{W}(\Delta, \mathtt{dom}(\sigma), T) \leq n$*.*

Proof. By induction on T:

- $(T \equiv \mathsf{end}$ or $T \equiv \{!\mathtt{m}_i\langle\alpha_i\rangle(t_i).T_i\}_{i \in I})$ Easy since $\mathtt{W}(\Delta, \mathtt{dom}(\sigma), T) = 0$.
- $(T \equiv \alpha)$ From the hypothesis $\Delta \vdash T\sigma :: n$ we deduce $\alpha \in \Delta \cup \mathtt{dom}(\sigma)$. By definition of algorithmic weight we conclude $\mathtt{W}(\Delta, \mathtt{dom}(\sigma), T) = 0$.
- $(T \equiv \mathtt{rec} \; \alpha.S)$ Let $\sigma' = (\sigma \backslash \alpha), \{\alpha \mapsto T\}$ where $\sigma \backslash \alpha$ is the restriction of σ to $\mathtt{dom}(\sigma) \backslash \{\alpha\}$. We have $\Delta \vdash S\sigma' :: n$. By induction hypothesis we deduce $\mathtt{W}(\Delta, \mathtt{dom}(\sigma) \cup \{\alpha\}, S) \leq n$. By definition of algorithmic weight we conclude $\mathtt{W}(\Delta, \mathtt{dom}(\sigma), T) = \mathtt{W}(\Delta, \mathtt{dom}(\sigma), \mathtt{rec} \; \alpha.S) = \mathtt{W}(\Delta, \mathtt{dom}(\sigma) \cup \{\alpha\}, S) \leq n$.
- $(T \equiv \{?\mathtt{m}_i\langle\alpha_i\rangle(q_i \; S_i).T_i\}_{i \in I})$ For every $i \in I$ let $\sigma_i = \sigma \backslash \{\alpha_i\}$. From the hypothesis $\Delta \vdash T\sigma :: n$ we deduce $\Delta \vdash S_i\sigma_i :: n - 1$ and $\Delta \vdash T_i\sigma_i :: n$ for every $i \in I$. By induction hypothesis we deduce $\mathtt{W}(\Delta, \mathtt{dom}(\sigma_i), S_i) \leq n - 1$ and $\mathtt{W}(\Delta, \mathtt{dom}(\sigma_i), T_i) \leq n$. We conclude $\mathtt{W}(\Delta, \mathtt{dom}(\sigma), T) \leq n$ by definition of algorithmic weight. $\qquad\square$

Correctness of the weight algorithm is simply a combination of the two previous lemmas.

Theorem C.2 (Theorem 6.2)**.** $\|T\|_\Delta = \mathtt{W}(\Delta, \emptyset, T)$.

Proof. From Lemma C.2 we deduce $\|T\|_\Delta = \min\{n \in \mathbb{N} \mid \Delta \vdash T :: n\} \leq \mathtt{W}(\Delta, \emptyset, T)$. From Lemma C.3, by taking $\sigma = \emptyset$ (the empty substitution), we conclude $\mathtt{W}(\Delta, \emptyset, T) \leq \|T\|_\Delta$. $\qquad\square$