# Intersection Types and Runtime Errors in the Pi-Calculus

UGO DAL LAGO, University of Bologna & INRIA Sophia Antipolis, Italy & France
MARC DE VISME, Université de Lyon, ENS de Lyon, CNRS, UCB Lyon 1, LIP, France
DAMIANO MAZZA, CNRS, UMR 7030, LIPN, Université Paris 13, Sorbonne Paris Cité, France
AKIRA YOSHIMIZU, INRIA Sophia Antipolis, France

We introduce a type system for the $\pi$-calculus which is designed to guarantee that typable processes are *well-behaved*, namely they never produce a run-time error and, even if they may diverge, there is always a chance for them to "finish their work", *i.e.*, to reduce to an idle process. The introduced type system is based on non-idempotent intersections, and is thus very powerful as for the class of processes it can capture. Indeed, despite the fact that the underlying property is $\Pi_2^0$-complete, there is a way to show that the system is *complete*, *i.e.*, that any well-behaved process is typable, although for obvious reasons infinitely many derivations need to be considered.

CCS Concepts: • **Theory of computation → Process calculi**; **Linear logic**; **Type structures**;

Additional Key Words and Phrases: intersection types, $\pi$-calculus, runtime error analysis, linear logic

**ACM Reference Format:**
Ugo Dal Lago, Marc de Visme, Damiano Mazza, and Akira Yoshimizu. 2019. Intersection Types and Runtime Errors in the Pi-Calculus. *Proc. ACM Program. Lang.* 3, POPL, Article 7 (January 2019), 29 pages. https://doi.org/10.1145/3290320

## 1 INTRODUCTION

The concept of a type system has been a useful abstraction in the theory and practice of programming languages, since the very early days [Backus et al. 1962]. Types are assigned to code fragments including variables, subroutines, and dynamic allocated objects, and their purpose is to guarantee a form of *safety*, *i.e.*, that "well-typed programs cannot go wrong" [Milner 1978], or even more complex properties, *e.g.*, that well-typed program *terminate* without going wrong [Hughes et al. 1996]. Noticeably, the converse is not necessarily true: there can well be programs *satisfying* the prescribed property which are *not* typable. But the guarantee provided by types is anyway very valuable, particularly when the behavior of subprograms within programs turns out to be complex, like for so-called higher-order functions: in those scenarios, keeping track of who does what is even more crucial to prevent run-time errors.

Indeed, type systems for higher-order programming languages are among the best studied topics in the programming language and logic in computer science communities. A host of different type systems exist, and forms of polymorphic [Girard 1971; Reynolds 1974], refinement [Freeman and Pfenning 1991], sized [Hughes et al. 1996], or dependent types (to mention only a few) have been applied to a variety of programming languages. There is always a tension between expressiveness

Authors' addresses: Ugo Dal Lago, University of Bologna & INRIA Sophia Antipolis, Italy & France, ugo.dallago@unibo.it; Marc de Visme, Université de Lyon, ENS de Lyon, CNRS, UCB Lyon 1, LIP, France, marc.de-visme@ens-lyon.fr; Damiano Mazza, CNRS, UMR 7030, LIPN, Université Paris 13, Sorbonne Paris Cité, France, damiano.mazza@lipn.univ-paris13.fr; Akira Yoshimizu, INRIA Sophia Antipolis, France, akira.yoshimizu@unibo.it.

2475-1421/2019/1-ART7
https://doi.org/10.1145/3290320

and tractability here: the more expressive the type system is, the more computationally expensive type checking and type inference are bound to be, because the underlying program properties are invariably undecidable. From simple types, where type inference has polytime complexity, one can go all the way to System F, for which the same problem is well-known to be undecidable [Wells 1994], through ML-style type systems, in which inference is EXPTIME-complete [Kanellakis et al. 1994].

Logic and programming language *semantics* can help in studying the intrinsic limits of the type-based approach, by providing some guidelines as for *how* to design elegant type systems, or by exploring the intrinsic *limits* of the approach. A particularly relevant example is given by intersection types [Coppo and Dezani-Ciancaglini 1978], in which various normalization properties of $\lambda$-terms can be *precisely characterized* by typability (see for instance [Krivine 1993] for the standard examples: head normalization, weak normalization, strong normalization).

But what happens if one switches to concurrent calculi and process algebras? Type systems guaranteeing safety properties of processes have been studied for a long time now, with remarkable results coming out in the last twenty years. For example, quite robust and expressive type systems guaranteeing deadlock freedom of $\pi$-processes can be given [Suenaga and Kobayashi 2007]. As another example, session types [Honda et al. 1998] have had quite an impact in the way concurrent programs and processes can be structured. Finally, termination of processes is another area in which much progress can be observed in the last fifteen years [Demangeon et al. 2010a,b; Yoshida et al. 2004]. More information on related work can be found in Section 1.1.

Despite the very strong expressive power of the obtained type methodologies, not much is known about the *limits* of the type-based approach in a concurrency scenario, contrarily to what one observes in the realm of sequential languages. Is there any type system able not only to *guarantee* but also to *characterize* some relevant property of concurrent processes, like deadlock-freedom, liveness, or termination? In the literature, some attempts in this direction can be found, but none of them gives a definite answer. Giving a satisfactory answer to the question above is, we believe, a key step in understanding the deep nature of the aforementioned properties, which are universally accepted to be desirable in concurrent systems.

In this paper, we show that, indeed, an intersection type discipline may be given for a fragment of the $\pi$-calculus, and proved sound *and complete* for checking, in essence, a form of deadlock-freedom and termination for processes, which we dub *good behavior*. This is the first such result, as far as the authors know.

The route to the result is full of caveats, however. First of all, one needs to clarify that the intrinsic non-determinism of $\pi$-calculus processes prevents us from hoping to have such a type system, at least if one sticks to the usual scheme in which *one* type derivation is sufficient as a certificate for the given property. Indeed, the class of well-behaved processes turns out to be a $\Pi_2^0$-complete set, which by definition cannot be characterized by a finitary, recursively checkable, type system. This admittedly not surprising result will be proved formally in Sect. 2.4 below.

## 1.1 Related Work

Intersection types have been introduced by Coppo and Dezani [Coppo and Dezani-Ciancaglini 1978] as a generalization of simple types for ordinary $\lambda$-terms inspired by semantics. After their introduction, it became clear quite soon that intersection types could be used to characterize notions like that of a weakly normalizing [Coppo et al. 1981] or that of strongly normalizing $\lambda$-terms [Coppo and Dezani-Ciancaglini 1980]. By changing the properties satisfied by the intersection type operator, one can also turn intersection types into a way to read the complexity of normalization of a $\lambda$-term (i.e. the number of $\beta$-steps to normal form along a strategy) from one of its type derivations [Accattoli et al. 2018; de Carvalho 2018]. The same paradigm has been recently shown to be applicable to other kinds of $\lambda$-calculi, like probabilistic $\lambda$-calculi [Breuvart and Lago 2018].

Intersection type systems for $\pi$-calculus processes have already been proposed in the literature, e.g., by Castagna et al. [Castagna et al. 2006] and by Piccolo [Piccolo 2012]. In both of the cited papers, intersections are mixed with unions. But neither in one nor in the other, one can hope to have any completeness result. Moreover, intersection is used in an unusual way, namely for typing non-deterministic choices. In our work, intersections deal with the exponentials rather than the additives.

As already mentioned, many type systems for process algebras have been introduced in the last two decades. A cornerstone in the normalization theory for $\pi$-calculus is the work by Berger, Honda and Yoshida [Yoshida et al. 2004], in which a combination of logical and process-theoretical ideas led to a robust approach, capable of guaranteeing strong normalization for all typable processes.

Another rather prolific research line is centered around linearity in the sense of Girard's linear logic [Girard 1987], and has been initiated by the seminal work of Kobayashi, Pierce and Turner [Kobayashi et al. 1999]. This has been followed by many type systems, based or not on linearity, guaranteeing forms of deadlock-freedom [Kobayashi 2006; Padovani 2014], livelock-freedom [Kobayashi 2000], termination [Demangeon et al. 2010b; Deng and Sangiorgi 2004], or combinations thereof. Again, none of them aim at the kind of completeness property we are interested at here.

A very successful typing paradigm for concurrent processes is certainly the one of session types [Honda et al. 1998]. There, the inherently chaotic nature of concurrent interaction is tamed by way of the notion of a *session*, which is a tree-like description of the possible interactions happening between the (possibly multiple [Honda et al. 2008]) involved parties. Here, even more than in other type disciplines for processes, communication is constrained to follow a prescribed protocol, and so there is even less hope to get a completeness result. This is particularly visible in certainly formulations of session types (e.g. [Caires and Pfenning 2010]), in which *the only* way to type a parallel composition is to find one and only one channel that the two components share, and restrict it immediately after forming the parallel composition.

Summing up, our work lies more or less at the meeting point of a broad spectrum of type-based techniques: the hope of ensuring properties of processes by looking at just one derivation is traded for completeness which, as we prove in this work, makes the set of "well-typed" processes not recursively enumerable. It is thus hard to precisely contextualize this work, at least in the realm of concurrency theory. The comparison, e.g., with the work on termination [Demangeon et al. 2010b; Deng and Sangiorgi 2004] is a good example: their typability implies *strong* normalization, while we characterize a *weak* form of normalization (plus error-freedom), but guaranteeing completeness, which in their work is by construction impossible.

### 1.2 Contributions

This paper's contributions can be summarized as follows:

- We introduce a notion of asynchronous hyperlocalized $\pi$-process, study its expressive power by way of some examples, and define some basic behavioral properties for them. This is in Sect. 2. "Hyperlocalized" means that our $\pi$-calculus is in fact a fragment of the localized calculus, *i.e.*, we have further constraints on the presence of input names under input prefixes. These constraints do not hinder in a significant way the expressiveness of the calculus, as we show in Sect. 2.3.
- We prove that verifying the good behavior of an AHL $\pi$-term is a very hard problem, namely it is $\Pi_2^0$-complete. This is proved by reducing the universal termination problem for Turing

machines to it. Although not surprising, this is the first result of this kind in the literature, and shows that good behavior of $\pi$-processes is fundamentally different from termination in purely sequential languages like the $\lambda$-calculus, in which the property of being terminating (in any reasonable sense) can be proved by finitary, recursively checkable, certificates.

- We introduce our intersection type system and prove its main property: a novel notion of complete typability characterizes good behavior (Sect. 3). Furthermore, we provide examples to show how our type system works with some familiar concepts (Sect. 3.4).

We conclude by discussing our work and its perspectives (Sect. 4).

## 2 THE CALCULUS

In this section, we present the process calculus at the center of our paper. We give its syntax and reduction semantics (Section 2.1) and define well-behaved processes (Section 2.2), a class of processes that we will then characterize in the next section. The calculus we consider is based on the polyadic asynchronous localized $\pi$-calculus [Sangiorgi and Walker 2001], on which we impose a further, non-standard constraint that we call *hyperlocalization*. Despite being a non-trivial restriction, hyperlocalization does not significantly hinder the expressiveness of the calculus, as we show in Section 2.3. As announced, the property of being well-behaved is of an essentially different nature with respect to, say, termination in sequential programming languages like the $\lambda$-calculus: the latter is typically $\Sigma_1^0$-complete, while here we are strictly higher in the arithmetical hierarchy, as shown in Section 2.4.

### 2.1 Processes and Reduction

We fix a countably infinite set of *names*, ranged over by $x, y, z, w$. We write $\widetilde{x}$ for an arbitrary finite sequence of names, the length of which is denoted by $|\widetilde{x}|$. Later, the notation $\widetilde{\cdot}$ will also be used for sequences of other objects (types, etc.).

We start by considering the following variant of the polyadic asynchronous $\pi$-calculus:

$$\begin{array}{rll} P, Q ::= \mathbf{0} \;\Big|\; P \mid Q \;\Big|\; \nu x P & & \text{terminated process, parallel composition, name restriction} \\ \Big|\; \overline{x}\langle\widetilde{y}\rangle \;\Big|\; x(\widetilde{y}).P \;\Big|\; !x(\widetilde{y}).P & & \text{output particle, input prefix, replicated input prefix (server)} \end{array}$$

In an output particle $\overline{x}\langle\widetilde{y}\rangle$ or input prefix $x(\widetilde{y})$ or $!x(\widetilde{y})$, the name $x$ is said to occur as *subject*; in the case of the output particle, the names $\widetilde{y}$ are said to occur as *objects*. The set $\mathsf{fn}(P)$ of free names of $P$ is defined as usual: the names $\widetilde{y}$ are bound in $x(\widetilde{y}).P$ and $!x(\widetilde{y}).P$, and $x$ is bound in $\nu x P$; free names are those that are not bound. We write $\mathsf{out}(P)$ for the subset of $\mathsf{fn}(P)$ of those names appearing as subjects of output prefixes. As usual, we write $\overline{x}$, $x.P$ and $!x.P$ as short-hands for $\overline{x}\langle\rangle$, $x().P$ and $!x().P$, respectively.

*Structural congruence* is the contextual, symmetric-transitive closure of the following equations:

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

$$\nu x P \mid Q \equiv \nu x (P \mid Q) \qquad \text{provided that } x \notin \mathsf{fn}(Q).$$

As usual, we write $\nu(x_1, \ldots, x_n)P$ for an arbitrary sequence of restrictions, the order of which is irrelevant by the fourth equation. As customary in process algebras, structural congruence has no computational value and is meant to capture the fact that some processes, although syntactically different, should in fact be considered as identical.

Reduction between processes is defined as follows. First, we have the basic rules

$$\begin{array}{rcll} \overline{x}\langle\widetilde{y}\rangle \mid x(\widetilde{z}).P & \longrightarrow & P\{\widetilde{y}/\widetilde{z}\} & \text{provided that } |\widetilde{y}| = |\widetilde{z}|, \\ \overline{x}\langle\widetilde{y}\rangle \mid !x(\widetilde{z}).P & \longrightarrow & P\{\widetilde{y}/\widetilde{z}\} \mid !x(\widetilde{z}).P & \text{provided that } |\widetilde{y}| = |\widetilde{z}|, \\ \nu x(!x(\widetilde{z}_1).P_1 \mid \cdots \mid !x(\widetilde{z}_n).P_n) & \longrightarrow & \mathbf{0}. & \end{array}$$

The first rule is standard: it corresponds to the basic intuition that $R := x(\widetilde{z}).P$ is a process waiting to receive a number of names via channel $x$, and that $\overline{x}\langle\widetilde{y}\rangle$ represents a message consisting of a number of names $\widetilde{y}$ sent on $x$; when such an output particle meets $R$, the names $\widetilde{y}$ are received by $R$ and the process continues as $P\{\widetilde{y}/\widetilde{z}\}$, which denotes the substitution of each name in $\widetilde{y}$ to the corresponding formal parameter in $\widetilde{z}$. For such a substitution to be well-defined, the sequences $\widetilde{y}$ and $\widetilde{z}$ must have equal length, which explains the side condition.

The second rule is derivable from the first rule and the standard equation $!P \equiv P \mid !P$, which we prefer to avoid in order to keep structural congruence simpler. This rule corresponds to the intuition that $!x(\widetilde{z}).P$ acts as a server capable of behaving like $x(\widetilde{z}).P$ an indefinite number of times, *i.e.*, capable of accepting an indefinite number of requests.

The third rule is not strictly necessary but makes the presentation of our results cleaner. It is sensible under the server intuition described above: once it becomes clear that no-one will ever make any request on $x$, a "server pool" on $x$ may be shut down. Note that the case $n = 0$ is $\nu x\mathbf{0} \longrightarrow \mathbf{0}$, whose equational form is usually one of the axioms of structural congruence.

The first two steps are called *communication steps*, the third *clean-up step*. The *reduction relation* between processes, denoted by $\longrightarrow$, is the closure of the above basic rules under *evaluation contexts*

$$C ::= \{\cdot\} \;\Big|\; C \mid P \;\Big|\; \nu x C$$

and under structural congruence:

$$S \equiv P \longrightarrow Q \equiv R \quad \text{implies} \quad S \longrightarrow R.$$

We write $\longrightarrow^*$ for the reflexive-transitive closure of $\longrightarrow$.

*Definition 2.1 (hyperlocalized process).* A process of the above calculus is *hyperlocalized* if, in any subprocess of the form $x(\widetilde{y}).P$ or $!x(\widetilde{y}).P$, no $z \in \mathsf{fn}(P)$ occurs as the subject of an input prefix. The set of hyperlocalized processes forms the *polyadic asynchronous hyperlocalized $\pi$-calculus*, or AHL$\pi$ for short.

In the more standard *localized $\pi$-calculus* [Sangiorgi and Walker 2001]), the restriction to $x(\widetilde{y}).P$ and $!x(\widetilde{y}).P$ is that only the names in $\widetilde{y}$ are forbidden to appear as subjects of input prefixes, so our constraint is strictly stronger, hence the terminology. Let us give a few simple examples:

$$\begin{array}{rl} x(y).(\overline{y} \mid y.\mathbf{0} \mid \overline{w}\langle y\rangle) & \text{not localized;} \\ x(y).(\overline{y} \mid z.\mathbf{0} \mid \overline{w}\langle y\rangle) & \text{localized, not hyperlocalized;} \\ x(y).\nu z(\overline{y} \mid z.\mathbf{0} \mid \overline{w}\langle y\rangle) & \text{hyperlocalized.} \end{array}$$

The main property of the localized $\pi$-calculus is that, in a process of the form $\nu xP$, every possible input via $x$ is statically determined by the syntax of $P$. As [Sangiorgi and Walker 2001] put it, "every process that receives via a name is local to the process that created that name". This locality property is useful in several applications, such as distributed programming (where processes run at different sites) and object-oriented programming (where names represent object names). Our additional restriction is of logical nature; we defer a discussion to Sect. 4.

The following lemma, which shows that AHL$\pi$ is a well-defined subcalculus of the asynchronous $\pi$-calculus, is straightforward:

Lemma 2.2. *Let $P$ be a hyperlocalized process.*

1. *If $P \equiv Q$, then $Q$ is hyperlocalized.*
2. *If $P \longrightarrow Q$, then $Q$ is hyperlocalized.*

## 2.2 Behavioral Properties of Processes

We now define the properties of processes that we will be interested in. The first concept we need is the one of runtime error, which arguably encompasses those situations which we would like to avoid in the execution of a process.

*Definition 2.3 (Runtime Error).* A *runtime error* is one of the following configurations:

**Arity Mismatch:** a process of the form $\overline{x}\langle\widetilde{y}\rangle \mid x(\widetilde{z}).P$ or $\overline{x}\langle\widetilde{y}\rangle \mid !x(\widetilde{z}).P$ such that $|\widetilde{y}| \neq |\widetilde{z}|$; such configurations are stuck because two processes are trying to communicate but the number of actual parameters sent does not match the number of formal parameters expected on the receiving side.

**Failed Send:** a process of the form $\nu x(\overline{x}\langle\widetilde{y}\rangle \mid R)$, where $R$ contains no input prefix whose subject is $x$; this is a process in which data has been sent on channel $x$ but will never be consumed.

**Endless Wait:** a process of the form $\nu x(x(\widetilde{z}).Q \mid R)$, where $x \notin \mathrm{out}(R)$, *i.e.*, $R$ contains no output particle whose subject is $x$; dual to a Failed Send, this is a situation in which a process waits for an input on channel $x$ which will never arrive.

**Dependency Cycle:** an *i/o cycle* on $x_1, \ldots, x_n$, with $n \geq 1$, is a process of the form

$$\dagger x_1(\widetilde{y}_1).Q_1 \mid \cdots \mid \dagger x_n(\widetilde{y}_n).Q_n,$$

where $\dagger x(\widetilde{y}).Q$ stands for either $x(\widetilde{y}).Q$ or $!x(\widetilde{y}).Q$, such that $x_{i+1} \in \mathrm{out}(Q_i)$ for all $1 \leq i < n$ and $x_1 \in \mathrm{out}(Q_n)$ (remember that $\mathrm{out}(P)$ is the set of free names of $P$ occurring as subjects of output particles). A *dependency cycle* is a process of the form

$$\nu(x_1, \ldots, x_n)(C \mid R)$$

where $C$ is an i/o cycle on $x_1, \ldots, x_n$ and $R$ is arbitrary; this is a classic form of deadlock, in which the dependency graph between processes is indeed cyclic.

Dependency cycles are perhaps the hardest to grasp, so let us give an example:

$$\nu(x, y)(x.\overline{y} \mid y.\overline{x} \mid \overline{x}),$$

which fits the above definition once we take $x_1 = x$, $x_2 = y$, $C = x.\overline{y} \mid y.\overline{x}$ and $R = \overline{x}$. There is obviously a sort of a vicious circle here: the first subprocess of $C$ waits for the second subprocess to send over $x$, but the second subprocess cannot proceed because it is waiting for the first subprocess to send over $y$. Of course, a configuration containing a "self-cycle" such as $\nu x(x.\overline{x} \mid \overline{x})$ is also a dependency cycle (here, adding $\overline{x}$ in parallel makes the example more conspicuous because $\nu x\, x.\overline{x}$ is an endless wait as well as a dependency cycle).

The reader may object here that the vicious circles of our dependency cycles are not so "vicious" because that may be broken: for instance, in the first example above, the particle $\overline{x}$ may be used to "unblock" the first subprocess, yielding

$$\nu(x, y)(x.\overline{y} \mid y.\overline{x} \mid \overline{x}) \longrightarrow \nu(x, y)(\overline{y} \mid y.\overline{x}) \longrightarrow^* \nu x\, \overline{x}.$$

Note, however, that the resulting process is a failed send, *i.e.*, a runtime error. This is not a coincidence; we will come back to this point momentarily.

Now that the notion of a runtime error has been precisely formalized, defining well-behaved processes is relatively easy: we only need to remember that for a process to be error-free it is necessary that it does not produce a runtime error *at top-level.*

*Definition 2.4 (Well-Behaved Process).* A process $P$ contains a *top-level runtime error* if

$$P \equiv \nu\widetilde{z}(E \mid Q)$$

where $E$ is a runtime error. It is *error-free* if it contains no top-level runtime error. A process is *idle* if it is error-free and cannot reduce (*i.e.*, it is in normal form with respect to the rewriting rules of Sect. 2.1). A process $P$ is *well-behaved* if, whenever $P \longrightarrow^* Q$:

(1) $Q$ is error-free;
(2) there exists an idle process $I$ such that $Q \longrightarrow^* I$.

Note that the only closed idle process is $\mathbf{0}$. Therefore, in the closed case, a well-behaved process is one that never generates top-level runtime errors and always has a way of evolving to $\mathbf{0}$. This is a typical property that one would ask of an operating system: it may run forever, but it must never produce errors and may always be shut down.

Of course, our notion of good behavior is only as convincing as the notion of runtime error it is based upon. For what concerns arity mismatches, failed sends and endless waits, we hope that the reader will not have trouble recognizing these as unwanted configurations. Dependency cycles are a bit more subtle because, as observed above, we allow them to be broken. A less controversial notion would be the following:

*Definition 2.5 (Deadlock).* A dependency cycle $\nu(x_1, \dots, x_n)(C \mid R)$ is *breakable* if $R \longrightarrow^* \overline{x_i}\langle\widetilde{y}\rangle \mid R'$ for some $1 \leq i \leq n$. A *deadlock* is an unbreakable dependency cycle.

The simplest examples of deadlocks are dependency cycles in which $R = \mathbf{0}$, *e.g.* $\nu(x, y)(x.\overline{y} \mid y.\overline{x})$, but of course more complex examples may be imagined, such as $\nu(x, y)(x.\overline{y} \mid y.\overline{x} \mid z.\overline{x})$. These are clearly "bad" configurations because not only are they vicious circles but they are also persistent, *i.e.*, their presence cannot be removed by reduction. We therefore hope that the reader will be fully convinced by the following notion of "bad" behavior:

*Definition 2.6 (Ill-Behaved Process).* A process $P$ is *ill-behaved* if there exists $Q$ such that $P \longrightarrow^* Q$ and one of the following holds:

(1) $Q$ contains a top-level arity mismatch, failed send, endless wait or deadlock;
(2) $Q$ cannot reduce to an idle process.

It turns out that, in AHL$\pi$, the ill-behaved processes are exactly those that are *not* well-behaved. That is, for behavioral purposes, our looser notion of dependency cycle is equivalent to the less controversial notion of deadlock. More precisely, assuming hyperlocality, one may show (as we do below) that the presence of a dependency cycle, even breakable, implies the possibility of the appearance of an "uncontroversially bad" configuration (namely, a deadlock, a failed send or a process with no idle form). Since dependency cycles admit a simpler definition than deadlocks, this explains why we chose to ground our "official" definition of good behavior on the former rather than the latter.

In the following, we call a process that cannot reduce to an idle form *restless*.

Lemma 2.7. *Let $P := \nu(x, \widetilde{w})(\dagger x(\widetilde{z}).Q \mid \overline{x}\langle\widetilde{y}\rangle \mid R)$ be hyperlocalized, with $\dagger x(\widetilde{z}).Q$ standing for either $x(\widetilde{z}).Q$ or $!x(\widetilde{z}).Q$. Then:*

- *if $Q\{\widetilde{y}/\widetilde{z}\}$ is restless, then $P$ reduces to a restless process;*
- *if, on the contrary, $Q\{\widetilde{y}/\widetilde{z}\}$ reduces to an idle form, then*

$$P \longrightarrow^* \nu(x, \widetilde{w}')(\overline{s_1}\langle\widetilde{r_1}\rangle \mid \cdots \mid \overline{s_m}\langle\widetilde{r}_m\rangle \mid R)$$

*and, for all $p \in \mathrm{out}(Q)$, there exists $i$ such that $s_i = p$.*

Proof. Assume that $Q' := Q\{\widetilde{y}/\widetilde{z}\}$ is restless. We have

$$P \longrightarrow \nu(x, \widetilde{w})(Q' \mid R') =: S,$$

where $R' = R$ or $R' = !x(\widetilde{z}).Q \mid R$ depending on the form of $\dagger x(\widetilde{z}).Q$. By hyperlocality, no free name of $Q'$ is the subject of an input prefix in $Q'$, which means that every communication that $Q'$ may perform with $R'$ concerns an output particle $\overline{a}\langle\widetilde{b}\rangle$ produced by $Q'$ and consumed by $R'$. Furthermore, such an output particle cannot interact within $Q'$, for otherwise there would be an input on $a$ in $Q'$, contradicting locality. Therefore, neither these output particles nor $R'$ itself may bear any impact on the reductions causing $Q'$ to be restless, so the process $S$ is restless.

On the contrary, if $Q' \longrightarrow^* I$ with $I$ idle, then a straightforward case analysis shows that $I = \nu\widetilde{r}(\overline{s_1}\langle\widetilde{r_1}\rangle \mid \cdots \mid \overline{s_m}\langle\widetilde{r_m}\rangle)$ (there may be no inputs because of hyperlocality). Furthermore, if $p \in \mathrm{out}(Q)$, *i.e.*, $p$ is free in $Q$ and there is an output particle $\overline{p}\langle\widetilde{q}\rangle$ in $Q$, then there must be some $\overline{s_i}\langle\widetilde{r_i}\rangle$ which is equal to $\overline{p}\langle\widetilde{q}\rangle$, because by hyperlocality there cannot be any input on $p$ in $Q'$ and so no reduction of $Q'$ may consume $\overline{p}\langle\widetilde{q}\rangle$. □

Proposition 2.8. *A hyperlocalized process is ill-behaved iff it is not well-behaved.*

Proof. Ill-behaved processes are obviously not well-behaved, so let us turn to the converse. For this, it is enough to show that if a hyperlocalized process $P$ contains a top-level *breakable* dependency cycle, then $P$ is ill-behaved. So let

$$C := \dagger x_1(\widetilde{z_1}).Q_1 \mid \cdots \mid \dagger x_n(\widetilde{z_n}).Q_n$$

be an i/o cycle on $\widetilde{x} = x_1, \ldots, x_n$ and suppose that

$$P = \nu(\widetilde{x}, \widetilde{w})(C \mid R)$$

such that $R \longrightarrow^* \overline{x_i}\langle\widetilde{y}\rangle \mid R'$. We may assume without loss of generality that $i = 1$ (a cycle being perfectly symmetric). Let

$$P' := \nu(\widetilde{x}, \widetilde{w})(C \mid \overline{x_1}\langle\widetilde{y}\rangle \mid R').$$

We have two possibilities:

- either $Q_1\{\widetilde{y}/\widetilde{z_1}\}$ is restless, in which case, by Lemma 2.7, $P'$ (and hence $P$) reduces to a restless process and is therefore ill-behaved by definition;
- or $Q_1\{\widetilde{y}/\widetilde{z_1}\}$ is not restless, *i.e.*, it reduces to an idle form which, again by Lemma 2.7, is of the form $\nu\widetilde{r_1}(\overline{x_2}\langle\widetilde{y_2}\rangle \mid I_1)$ with $I_1$ consisting only of output particles (remember that $x_2 \in \mathrm{out}(Q_1)$ by hypothesis), and therefore

  $$P \longrightarrow P' \longrightarrow^* \nu(\widetilde{x}, \widetilde{w}, \widetilde{r_1})(Q_1' \mid \dagger x_2(\widetilde{z_2}).Q_2 \mid \cdots \mid \dagger x_n(\widetilde{z_n}).Q_n \mid \overline{x_2}\langle\widetilde{y_2}\rangle \mid R' \mid I_1),$$

  where $Q_1'$ is $!x_1(\widetilde{y_1}).Q_1$ or $\mathbf{0}$, depending on whether the input $\dagger x_1(\widetilde{y_1}).Q_1$ in $C$ is replicated or not.

In the second case, we are in position of iterating the same reasoning, which leads us (by induction on $n$, if we want to be fully formal) to the conclusion that either $P$ reduces to a restless process or

$$P \longrightarrow^* \nu(\widetilde{x}, \widetilde{w}')(C' \mid \overline{x_1}\langle\widetilde{y_1}\rangle \mid R' \mid I),$$

where $C'$ consists of the replicated inputs from the original cycle $C$ and $I$ consists of a number of output particles deriving from the idle forms of the various $Q_i\{\widetilde{y_i}/\widetilde{z_i}\}$.

Now, if the process on the right hand side has no input on $x_1$, we have a failed send and we may conclude. Otherwise, observe that, by hyperlocality, such an input must be at top-level in either $C'$ or $R'$. Let us start by supposing that it is in $R'$. In that case, the output particle $\overline{x_1}\langle\widetilde{y}\rangle$ in $P'$ may interact directly with $R'$, preventing the cycle to break. We have two possibilities:

- $R$ is able to produce only finitely many output particles interacting with $C$: in this case, we have shown that these lead either to a restless process, or they may actually be consumed "internally" to $R$ itself; therefore, either we find a restless process or, after finitely many steps, the dependency cycle loses every chance of being broken and becomes a true deadlock;
- $R$ produces infinitely many output particles interacting with $C$: in this case, the reasoning may be iterated again and again, showing that $P$ is restless.

Suppose now that the input on $x_1$ is in $C'$, which means that it is equal to $!x_1(\widetilde{z}_1).Q_1$. We have two subcases:

- every input of the cycle $C$ is replicated, *i.e.*, $C' = C$. Then, $P'$ is restless: indeed, we have arrived here from $P'$ by avoiding reductions to restless processes, *i.e.*, all other possibilities lead to restless processes; in spite of this, we still obtained a process essentially identical to $P'$, so we are bound to loop forever.
- There is at least one input of $C$ which is not replicated. Let us suppose for simplicity that this is $x_2(\widetilde{y}_2).Q_2$ (the other possibilities are not different in an essential way). In this case, we let the output $\overline{x_1}\langle\widetilde{y}_1\rangle$ be consumed by $!x_1(\widetilde{z}_1).Q_1$ in $C'$, we apply the same reasoning as above and obtain $\nu(\widetilde{x}, \widetilde{w}')(C' \mid \overline{x_2}\langle\widetilde{y}'_2\rangle \mid R' \mid I')$, except that now in $C'$ there is no input on $x_2$, so we are brought to the case already considered above in which an input on $x_2$, if present at all, is in $R'$, and we conclude.

□

### 2.3 Expressiveness

The aim of this section is to give some convincing arguments on the expressiveness of AHL$\pi$. More specifically, we show that three different forms of processes, the first one sequential and the other two truly concurrent, can all be written as AHL$\pi$ processes.

*2.3.1 Functional Programming.* We will start by considering a parallel and non-deterministic $\lambda$-calculus, denoted by $\Lambda^{[\cdot],\mid}$. This is a nontrivial extension of the pure, untyped $\lambda$-calculus endowed with a form of parallel composition, and in which a function can be applied to *more than one* term at the same time. Terms of $\Lambda^{[\cdot],\mid}$ are the expressions generated by the following grammar:

$$M, N ::= x \;\Big|\; \lambda x.M \;\Big|\; M[N_1, \ldots, N_k] \;\Big|\; \mathbf{0} \;\Big|\; M \mid N$$

As usual, the parallel operator is assumed to be associative, commutative and with neutral element $\mathbf{0}$. Reduction is defined from the following rules (we write $\vec{N}$ for $[N_1, \ldots, N_k]$)

$$(\lambda x.M)\vec{N} \longrightarrow M\{N_i/x\}, \qquad \mathbf{0}\vec{N} \longrightarrow \mathbf{0}, \qquad (M \mid N)\vec{P} \longrightarrow M\vec{P} \mid N\vec{P},$$

closed under *weak head evaluation contexts* $E ::= \{\cdot\} \;\Big|\; EM \;\Big|\; E \mid M$. Observe how the first rule *non-deterministically* picks one $N_i$ and substitutes it to $x$. Because of this rule, the calculus is not confluent. This extension of the $\lambda$-calculus may be encoded in our fragment of AL$\pi$ as follows:

$$\begin{aligned}
[\![x]\!]u &:= \overline{x}\langle u\rangle \\
[\![\lambda x.M]\!]u &:= \nu a(\overline{u}\langle a\rangle \mid a(x, v).[\![M]\!]v) \\
[\![M[N_1, \ldots, N_k]]\!]u &:= \nu v([\![M]\!]v \mid v(a).\nu z(\overline{a}\langle z, u\rangle \mid !z(w).[\![N_1]\!]w) \\
&\qquad\qquad \mid \cdots \mid v(a).\nu z(\overline{a}\langle z, u\rangle \mid !z(w).[\![N_k]\!]w)) \\
[\![\mathbf{0}]\!]u &:= \mathbf{0} \\
[\![M \mid N]\!]u &:= [\![M]\!]u \mid [\![N]\!]u
\end{aligned}$$

where $u$ is a fresh name. The fact that the encoding lands in AHL$\pi$ is immediate once we observe the following invariant, easily proved by induction: $\mathsf{fn}([\![M]\!]u) = \mathsf{fv}(M) \cup \{u\}$ (where $\mathsf{fv}(M)$ denotes the set of free variables of $M$) and, moreover, every name in $\mathsf{fn}([\![M]\!]u)$ occurs in $[\![M]\!]u$ itself in *output* position (as subject or object).

The above encoding is very similar to (the asynchronous variant of) Sangiorgi's encoding of the call-by-name $\lambda$-calculus given in [Sangiorgi and Walker 2001] (Milner's original encoding [Milner 1992] is slightly more succinct but is not localized). Indeed, one may look at the above calculus as a "reverse engineering" of a parallel liberalization of Sangiorgi's encoding: one takes the image of that encoding, closes it under parallel composition and adds the possibility of having an arbitrary number of concurrent arguments in the encoding of an application (which is quite natural from the $\pi$-calculus point of view). From this, one may "read back" a $\lambda$-calculus (the one presented above) which maps to such processes, treating parallel composition homomorphically.

The main property of the encoding, which follows from the corresponding property of Sangiorgi's encoding via the above discussion, is the following (we use $\simeq^c$ to denote barbed congruence on AHL$\pi$, which is defined as customary [Sangiorgi and Walker 2001]):

Proposition 2.9. *Let $M$ be a term of $\Lambda^{[\cdot],|}$.*

1. *$M \longrightarrow M'$ implies $[\![M]\!]u \longrightarrow^*\simeq^c [\![M']\!]u$;*
2. *$[\![M]\!]u \longrightarrow^* P$ implies that $M \longrightarrow M'$ for some $M'$ such that $P \longrightarrow^*\simeq^c [\![M']\!]u$.*

Although we will never use them here, it is straightforward to endow $\Lambda^{[\cdot],|}$ with control operators (like those corresponding to classical logic proofs in the sense of the Curry-Howard correspondence), obtaining a non-deterministic and parallel extension of the $\lambda\mu$-calculus:

- one adds the construct $\mu a.[b]M$ to the grammar, where $a, b$ range over continuation variables;
- one adds the rule
$$(\mu a.[b]C\{[a]M_1,\dots,[a]M_n\})\vec{N} \longrightarrow \mu a.[b]C\{[a]M_1N_{i_1},\dots,[a]M_nN_{i_n}\},$$
where $C$ is a $n$-hole context and $i_1,\dots,i_n \in \{1,\dots,k\}$ are arbitrary and not necessarily distinct, *i.e.*, the rule picks $n$ terms among $N_1,\dots,N_k$, possibly using the same term more than once and discarding some terms;
- the encoding $[\![\cdot]\!]$ is extended by setting $[\![\mu a.[b]M]\!]u := [\![M]\!]b\{u/a\}$.

*2.3.2 Locks.* A lock is one of the most basic concurrent primitives, being a way to model the safe access to resources which can only be accessed sequentially, like critical regions. A particularly economic way of implementing a lock in the $\pi$-calculus is to see it as a buffer containing at most one value: in $\nu l(P_1 \mid \cdots \mid P_n \mid \overline{l})$, each $P_i$ competes to read from $l$, and puts back a value once it has done. Unfortunately, this simple solution is not acceptable from our point of view: $P_i$ will typically be of the form $l.Q_i\{\overline{l}\}$, *i.e.*, $Q_i$ contains $\overline{l}$, inducing a dependency cycle. Consider instead

$$L := !a(z).\nu v(\overline{p}\langle v\rangle \mid v.\overline{z}\langle z\rangle)$$
$$\mathsf{Lock} := \nu a(\overline{a}\langle a\rangle \mid L).$$

A process using the above lock will be of the form $P_i := p(v).Q_i$ with $Q_i \longrightarrow^* R_i \mid \overline{v}$, *i.e.*, $P_i$ waits for the signal from the lock, then performs some operations, upon completion of which sends a release signal via the channel received from the lock process. The process

$$P_1 \mid \cdots \mid P_i \mid \cdots \mid P_n \mid \mathsf{Lock}$$

reduces to

$$\nu v(P_1 \mid \cdots \mid Q_i \mid \cdots \mid P_n \mid \nu a(v.\overline{a}\langle a\rangle \mid L))$$

in which a non-deterministically chosen process is ready to execute, while the others are waiting for it to release the lock. We will later analyze the behavior of this process via types. For the moment, it suffices to observe that this is an AHL$\pi$ process which does not exhibit a top-level run-time error.

*2.3.3 A Toy Operating System.* Consider the following definitions:

$$\begin{aligned}
\mathsf{Serv}_i &:= !r_i(a).\overline{a} \\
\mathsf{OpSys} &:= \mathsf{Serv}_1 \mid \cdots \mid \mathsf{Serv}_n \\
R_i(z) &:= \nu a(\overline{r_i}\langle a\rangle \mid a.\overline{z}\langle z\rangle) \\
U(z) &:= \nu s(\overline{s} \mid !s.R_1(z) \mid \cdots \mid !s.R_n(z) \mid !s.\mathbf{0}) \\
\mathsf{User} &:= \nu u(\overline{u}\langle u\rangle \mid !u(z).U(z))
\end{aligned}$$

OpSys is the abstraction of a multi-user, concurrent operating system offering services $\mathsf{Serv}_i$ for $1 \leq i \leq n$. Each subprocess $\mathsf{Serv}_i$ is the abstraction of a system call, which is invoked by sending a request on channel $r_i$, together with a channel where the answer has to be sent. User is the abstraction of a user who is logged into the system. The real definition is in $U$, the rest is needed to encode recursion: the user requests an indefinite number of services, in random order, and may eventually decide to logout (this is represented by the subprocess $!s.\mathbf{0}$).

The operating system with an arbitrary number of logged-in users may then be modeled by the process

$$\mathsf{User} \mid \cdots \mid \mathsf{User} \mid \mathsf{OpSys}.$$

Again, below we will perform a behavioral analysis via types also for this process.

## 2.4 Hardness of Checking Good Behavior

By resorting to some computable encoding of processes as integers (*i.e.*, Gödelization), the problem of checking whether a process is well-behaved may be formalized as a subset of $\mathbb{N}$. More precisely, let us define

$$W := \{n \in \mathbb{N} \mid n \text{ is the code of a well-behaved AHL}\pi \text{ process}\}.$$

We will now show that $W$ is $\Pi^0_2$-complete. For simplicity, we will use the Gödelization transparently, *i.e.*, we will say "a process $P$" instead of "the code $n$ of a process $P$".

That $W$ is a $\Pi^0_2$ set is fairly clear: runtime errors are finite configurations, so there is a $\Delta^0_0$ formula (*i.e.*, using only bounded quantifiers) $\mathrm{ErrorFree}(P)$ which holds iff $P$ is an error-free process; moreover, there is a $\Delta^0_0$ formula $\mathrm{Red}(P, Q, r)$ (similar to Kleene's T predicate) holding iff $r$ is the code of a reduction $P \longrightarrow^* Q$; finally, there is a $\Delta^0_0$ formula $\mathrm{Idle}(P)$ holding iff $P$ is idle. So $W$ is defined by the formula

$$\forall r.\forall Q.\exists q.\exists I.\mathrm{Red}(P, Q, r) \Rightarrow \mathrm{ErrorFree}(Q) \land \mathrm{Red}(Q, I, q) \land \mathrm{Idle}(I),$$

which contains $P$ as only free variable and is $\Pi^0_2$ by definition.

For what concerns the completeness of $W$, we decompose the reduction from a $\Pi^0_2$-complete problem in two steps. Call a term $M$ of $\Lambda^{[\cdot],|}$ *safe* if, whenever $M \longrightarrow^* M'$, there exists $N$ normal such that $M' \longrightarrow^* N$. Let $S$ be the set of (codes of) safe terms. Now, one may check that:

- thanks to the functional nature of the $\lambda$-calculus, the encoding $[\![M]\!]u$ of a $\lambda$-term $M$ never produces runtime errors, *i.e.*, for all $[\![M]\!]u \longrightarrow^* Q$, $Q$ is error-free;
- if $N$ is normal, then $[\![N]\!]u$ is idle.

Therefore, by the correctness of the encoding (Proposition 2.9), $M$ is safe iff $[\![M]\!]u$ is well-behaved. Since $[\![\cdot]\!]$ is clearly computable, we just proved that $S$ reduces to $W$. It is thus enough to show that

$S$ is $\Pi^0_2$-hard, which we will do by reducing to $S$ the archetypal $\Pi^0_2$-complete set, namely

$$H := \{n \in \mathbb{N} \mid n \text{ is the code of a Turing machine halting on all inputs}\}.$$

It is of course enough to restrict to Turing machines working on the binary alphabet $\{0, 1\}$. The following are completely standard results [Barendregt 1984]:

- a binary string $w \in \{0, 1\}^*$ may be encoded as a $\lambda$-term $\underline{w}$ such that there are $\lambda$-terms $\mathsf{cons}_i$, with $i \in \{0, 1\}$, satisfying $\mathsf{cons}_i\underline{w} \longrightarrow^* \underline{iw}$ by means of head reduction;
- for every Turing machine $M$ one may explicitly construct a $\lambda$-term $\underline{M}$ such that, for every $w \in \{0, 1\}^*$, the head reduction of $\underline{M}w$ terminates on $I$ (the identity $\lambda$-term $I = \lambda x.x$) iff $M$ terminates on input $w$.

By applying terms to a suitable number of identities, the above results transfer to weak call-by-name (remember that our $\lambda$-calculus of Sect. 2.3.1 does not reduce under abstraction).

Consider now the following terms of the $\lambda$-calculus of Sect. 2.3.1:

$$\begin{aligned} \mathsf{proj}_i &:= \lambda z_1.\lambda z_2.\lambda z_3.z_i, && i \in \{1, 2, 3\},\\ X &:= \lambda x.\lambda w.\lambda m.(\lambda p.p(mw)(xx(\mathsf{cons}_0 w)m)(xx(\mathsf{cons}_1 w)m))[\mathsf{proj}_1, \mathsf{proj}_2, \mathsf{proj}_3],\\ G_m &:= XX\underline{\varepsilon}m, \end{aligned}$$

where $m$ is a variable. The intuition behind $G_m$ is that it is a (recursively-defined) "random" string generator: we start from the empty string and, at each step, firing the redex involving multiapplication in $X$ non-deterministically chooses whether we add a 0, a 1 or whether we stop and apply $m$ to the string generated so far. One may easily check that:

(1) for all $w_1 \cdots w_n \in \{0, 1\}^n$, we have $G_m \longrightarrow^* m(\mathsf{cons}_{w_1}(\ldots \mathsf{cons}_{w_n}\underline{\varepsilon} \ldots))$;
(2) for all $G_m \longrightarrow^* N$, there exists $w_1 \cdots w_n \in \{0, 1\}^n$ s.t. $N \longrightarrow^* m(\mathsf{cons}_{w_1}(\ldots \mathsf{cons}_{w_n}\underline{\varepsilon} \ldots))$.

We now contend that a Turing machine $M$ terminates on all inputs iff $G_{\underline{M}}$ is safe, which is enough to prove that $H$ reduces to $S$. Suppose that $M$ terminates on all inputs, and let $G_{\underline{M}} \longrightarrow^* M'$. Now, there are two cases: either $\underline{M}$ reached the head position at some point during the reduction, in which case by property (2) above we have $G_{\underline{M}} \longrightarrow^* \underline{M}(\mathsf{cons}_{w_1}(\ldots \mathsf{cons}_{w_n}\underline{\varepsilon} \ldots)) \longrightarrow^* M'$ for some $w_1 \cdots w_n \in \{0, 1\}^n$; or $\underline{M}$ has not reached the head yet, in which case $M'$ is like the term $N$ of property (2) above, and we have $M' \longrightarrow^* \underline{M}(\mathsf{cons}_{w_1}(\ldots \mathsf{cons}_{w_n}\underline{\varepsilon} \ldots))$ for some $w_1 \cdots w_n \in \{0, 1\}^n$. In both cases, we are essentially computing $\underline{M}\,\underline{w}$ for some $w \in \{0, 1\}^*$, which head-normalizes by hypothesis, so $G_{\underline{M}}$ is safe.

Suppose now that there exists $w = w_1 \cdots w_n \in \{0, 1\}^n$ on which $M$ does not terminate. Then by property (1) above $G_{\underline{M}} \longrightarrow^* \underline{M}(\mathsf{cons}_{w_1}(\ldots \mathsf{cons}_{w_n}\underline{\varepsilon} \ldots))$, which behaves like $\underline{M}\,\underline{w}$, whose head reduction does not terminate by hypothesis, showing that $G_{\underline{M}}$ is not safe.

## 3 INTERSECTION TYPES

This section is devoted to presenting our type discipline for processes, also proving that it captures good behavior as we defined it in Section 2.2 above. We will also instantiate our type discipline to some example processes (Section 3.4). From here on, all processes are tacitly taken to be in AHL$\pi$.

### 3.1 Types and Derivations

Types cannot be defined directly here, and will be introduced as those pre-types satisfying a coherence condition. *Pre-types* are defined by mutual induction with *type sequences*:

$$\begin{aligned} A, B, C &::= \Theta_1 \wedge \cdots \wedge \Theta_k && \text{pre-types}\\ \Theta, \Xi &::= * \;\Big|\; (A_1, \ldots, A_n) && \text{type sequences} \end{aligned}$$

The nullary case (*i.e.*, $k = 0$) of intersection is written $\top$. Pre-types are considered up to commutativity of intersection, *e.g.* $() \wedge (*) = (*) \wedge ()$. Note that the constant $*$ is not the same as the empty sequence, denoted by $()$. Also observe that intersection is *not* idempotent: $\Theta \wedge \Theta \wedge \Xi$ is not the same as $\Theta \wedge \Xi$. If $A = \Theta_1 \wedge \cdots \wedge \Theta_k$ and $B = \Xi_1 \wedge \cdots \wedge \Xi_p$ are pre-types, then $A \wedge B$ is naturally defined as

$$\Theta_1 \wedge \cdots \wedge \Theta_k \wedge \Xi_1 \wedge \cdots \wedge \Xi_p.$$

In particular, $A \wedge \top = \top \wedge A = A$. There are no base types (except $*$, which however is not really a base type but a constant); including them would change nothing to our results.

We define the binary relation $\frown$ on pre-types and type sequences, called *coherence*, as the smallest symmetric relation satisfying

$$\frac{\forall \Theta}{* \frown \Theta} \qquad \frac{A_i \frown B_i \quad \forall i \in \{1, \ldots, n\}}{(A_1, \ldots, A_n) \frown (B_1, \ldots, B_n)} \qquad \frac{\Theta_i \frown \Xi_j \quad \forall i \in \{1, \ldots, k\},\ \forall j \in \{1, \ldots, p\}}{\Theta_1 \wedge \cdots \wedge \Theta_k \frown \Xi_1 \wedge \cdots \wedge \Xi_p}$$

A *type* is a pre-type $A$ such that $A \frown A$. An obvious example of type is $*$. By applying the rightmost rule when $k = 0$, we see that $\top \frown A$ for any $A$, so $\top$ is also a type. A more complex example is given by the family $A_0 := \top$, $A_{i+1} := (A_i) \wedge A_i$. Checking that $A_n$ is a type for all $n \in \mathbb{N}$ may be done by proving, by induction on $m + n$, the stronger statement that $A_m \frown A_n$ and $(A_m) \frown A_n$ for all $m, n \in \mathbb{N}$.

The idea behind the definition of pre-type is the following. Type sequences are needed to type input/output prefixes/particles: intuitively (and approximately), in $x(y_1, \ldots, y_n).P$ or $\overline{x}\langle y_1, \ldots, y_n\rangle$, if each $y_i$ has type $A_i$, then $x$ will be typed with $(A_1, \ldots, A_n)$. Since a name may appear more than once as subject of an input/output prefix/particle, intersection is used to collect all the corresponding type sequences: for instance, if a process contains two output particles $\overline{x}\langle\widetilde{y}\rangle$ and $\overline{x}\langle\widetilde{z}\rangle$, and if the types of $\widetilde{y}$ and $\widetilde{z}$ are $\widetilde{A}$ and $\widetilde{B}$, respectively, then $x$ will receive the type $(\widetilde{A}) \wedge (\widetilde{B})$. This also explains why intersection is non-idempotent. If $x$ is never used, then it will get type $\top$. Of course, this means that we may read back usage information from types: for instance, if $x : (()) \wedge (\top)$ types $x$ as input, we know that there is a reduction sequence in which $x$ will be used twice, once receiving a name which will in turn be used once for a nullary output (the type $()$) and once receiving a name which will not be used at all (the type $\top$).

There is a subtlety concerning inputs: when an input is never used, we must distinguish the replicated and non-replicated case. Indeed, while $!x(\widetilde{z}).P$ and $x(\widetilde{z}).P$ are both innocuous processes which must be typable, we have that $\nu x\, !x(\widetilde{y}).P \longrightarrow \mathbf{0}$, which must still be typable, whereas $\nu x\, x(\widetilde{y}).P$ is a runtime error (an endless wait, Definition 2.3) and must *not* be typable. This is the reason behind the presence of the constant $*$: it is used to type the subject of an unused non-replicated input prefix, whereas for unused replicated inputs we use $\top$.

While the input/output distinction does not appear at the level of types (they are, in a sense, "self-dual"), it will appear in the type system: a type on the left (resp. right) of a judgment will be an input (resp. output) type. A key property then will be for a name to appear with identical types on both sides of a typing judgment: indeed, $x : A$ on the left means "channel $x$ is used by input prefixes expecting specification $A$", so $x : A$ on the right means that such a specification is met by the outputs on $x$. This information is crucial for typing name restrictions (res rule, Fig. 1).

We have thus far explained the shape of pre-types but said nothing as to why these have to be restricted. The reason is that we want our type system to ensure at least absence of arity mismatches (the first kind of runtime error in Definition 2.3): recall that these arise when an output particle sending $n$ names on a channel meets an input prefix on the same channel expecting $m$ names, with $m \neq n$. The obvious solution is to force intersections to be "uniform": if $\Theta_1 \wedge \cdots \wedge \Theta_k$ is the pre-type of a channel, we want the length of each $\Theta_i$ to be equal. But of course this is too naive: the length must be the same "hereditarily", *i.e.*, the pre-types in each $\Theta_i$ must also be uniform, and so on. This "hereditary uniformity" is captured by the above coherence relation.

Let us now get to the type system. We consider typing judgments of the form

$$\Gamma \vdash P :: \Delta$$

where

- $P$ is a process of AHL$\pi$;
- $\Delta$ is a finite sequence (permutable at will) of declarations of the form

  $$y : B$$

  where $B$ is a type and $y$ is a name. Such a declaration corresponds to the possible presence in $P$ of free occurrences of $y$ as subjects or objects of an *output* particle.
- $\Gamma$ is a finite sequence (permutable at will) of declarations of the form

  $$(x; X) : A$$

  where $A$ is a type, $x$ a name and $X$ a finite set of names. Such a declaration corresponds to the possible presence in $P$ of free occurrences of $x$ as subjects of *input* prefixes and $X$ contains all those names, declared in $\Delta$, which may depend on $x$, in the sense that, if $y \in X$, then there may be an output particle mentioning $y$ whose emission ultimately depends on a reception on $x$ (a trivial example would be $x.\overline{y}$). We write $(x; y_1, \dots, y_k)$ for $(x; \{y_1, \dots, y_k\})$.

Please observe once again how $\Delta$ provides type information for channels involved in output actions, while $\Gamma$ does the same for input actions. A name $x$ may be declared both in $\Gamma$ and in $\Delta$, meaning that $x$ is possibly used both as input and output in $P$.

The typing rules are given in Fig. 1. Let us explain the notations used therein:

- the notation $\widetilde{x} : \widetilde{A}$ stands for $x_1 : A_1, \dots, x_n : A_n$, with $n$ arbitrary (including null).
- The notation $\widetilde{z} : \top$ on the right is short for an arbitrary sequence $z_1 : \top, \dots, z_n : \top$; on the left, $\widetilde{w} : \top$ is short for $(w_1;) : \top, \dots, (w_n;) : \top$; in both cases, the names declared do not appear free in the subject process.
- In the !in rule, $k \in \mathbb{N}$ is arbitrary and is said to be the *arity* of the instance of the rule. The intersections $\bigwedge_{j=1}^{k} \widetilde{C^j}$ are defined pointwise. For instance, the unary case is identical to the in rule (except for the subject process), whereas a typical binary instance may look like

  $$\dfrac{\vdash P :: y : A^1, z : C^1 \qquad \vdash P :: y : A^2, z : C^2}{(x; z) : (A^1) \wedge (A^2) \vdash\ !x(y).P :: z : C^1 \wedge C^2}\ !\mathsf{in}$$

  (we assumed that $\mathsf{fn}(P) = \{y, z\}$). The nullary case has the following general shape, which is very similar to that of the $\mathsf{in}_*$ rule, the difference being that here $x$ gets type $\top$ instead of $*$:

  $$\dfrac{}{\widetilde{w} : \top, (x; \mathsf{fn}(P) \setminus \widetilde{y}) : \top \vdash\ !x(\widetilde{y}).P :: \widetilde{z} : \top}\ !\mathsf{in}$$

  and of course we demand that $\widetilde{z}$ contains at least $\mathsf{fn}(P) \setminus \widetilde{y}$.
- in the par rule, the operation $\wedge$ on sequences of type declarations is pointwise. For instance, the following is an instance of the rule:

  $$\dfrac{\begin{array}{c}(x; v, z) : A, (y;) : C \vdash P :: v : D_1, w : E_1, z : F_1 \\ (x; w, z) : B, (y; v) : \top \vdash Q :: v : D_2, w : E_2, z : F_2\end{array}}{(x; v, w, z) : A \wedge B, (y; v) : C \vdash P \mid Q :: v : D_1 \wedge D_2, w : E_1 \wedge E_2, z : F_1 \wedge F_2}\ \mathsf{par}$$

The pre-condition of the out rule looks a bit complicated, so let us explain it. If we are typing an output particle $\overline{x}\langle z_1, \dots, z_n\rangle$, then the output context must declare at least $x, z_1, \dots, z_n$. The complication comes from the fact that some of the $z_j$ may be equal to each other and, furthermore,

$$\frac{}{\widetilde{w} : \top \vdash \mathbf{0} :: \widetilde{z} : \top}\ \text{zero}$$

$$\frac{A_1, \dots, A_n \text{ types} \quad f : \{1, \dots, n\} \to \{x, \widetilde{y}\} \quad C_i = \bigwedge_{j \in f^{-1}(y_i)} A_j,\ 1 \le i \le |\widetilde{y}|}{\widetilde{w} : \top \vdash \overline{x}\langle f(1), \dots, f(n)\rangle :: x : (A_1, \dots, A_n) \wedge \bigwedge_{j \in f^{-1}(x)} A_j, \widetilde{y} : \widetilde{C}}\ \text{out}$$

$$\frac{\vdash P :: \widetilde{y} : \widetilde{A}, \widetilde{z} : \widetilde{C}}{\widetilde{w} : \top, (x; \mathsf{out}(P) \setminus \widetilde{y}) : (\widetilde{A}) \vdash x(\widetilde{y}).P :: \widetilde{z} : \widetilde{C}}\ \text{in}$$

$$\frac{\widetilde{z} \text{ contains at least } \mathsf{fn}(P) \setminus \widetilde{y}}{\widetilde{w} : \top, (x; \mathsf{out}(P) \setminus \widetilde{y}) : * \vdash x(\widetilde{y}).P :: \widetilde{z} : \top}\ \text{in}_*$$

$$\frac{\vdash P :: \widetilde{y} : \widetilde{A}^1, \widetilde{z} : \widetilde{C}^1 \qquad \dots \qquad \vdash P :: \widetilde{y} : \widetilde{A}^k, \widetilde{z} : \widetilde{C}^k}{\widetilde{w} : \top, (x; \mathsf{out}(P) \setminus \widetilde{y}) : (\widetilde{A}^1) \wedge \cdots \wedge (\widetilde{A}^k) \vdash !x(\widetilde{y}).P :: \widetilde{z} : \bigwedge_{j=1}^{k} \widetilde{C}^j}\ \text{!in}$$

$$\frac{\widetilde{(x, X)} : \widetilde{A} \vdash P :: \widetilde{y} : \widetilde{C} \quad \widetilde{(x, Y)} : \widetilde{B} \vdash Q :: \widetilde{y} : \widetilde{D}}{\widetilde{(x; X \cup Y)} : \widetilde{A} \wedge \widetilde{B} \vdash P \mid Q :: \widetilde{y} : \widetilde{C} \wedge \widetilde{D}}\ \text{par}$$

$$\frac{\Gamma, (x; X) : A, (w_1; Y_1, x) : B_1, \dots, (w_n; Y_n, x) : B_n \vdash P :: \Delta, x : A \quad x \notin X, \Gamma}{\Gamma, (w_1; Y_1 \cup X) : B_1, \dots, (w_n; Y_n \cup X) : B_n \vdash \nu x P :: \Delta}\ \text{res}$$

Fig. 1. Typing rules.

equal to $x$ itself. In other words, the various $z_j$ are not names but *occurrences* of names. So we consider the names declared to be $x, \widetilde{y}$, with $\widetilde{y}$ a sequence of names (without repetitions and pairwise distinct from $x$) such that there is a function $f : \{1, \dots, n\} \to \{x, \widetilde{y}\}$ such that, for all $1 \le j \le n$, $f(j) = z_j$. The types $A_1, \dots, A_n$ mentioned in the rule are the types of the occurrences $z_1, \dots, z_n$; then, if *e.g.* $z_3 = z_7 = y_2$ (*i.e.*, $f^{-1}(y_2) = \{3, 7\}$), the context will contain the declaration $y_2 : A_3 \wedge A_7$. If no $z_j$ equals $y_i$ (*i.e.*, $f^{-1}(y_i) = \emptyset$), then $y_i$ gets type $\top$ (the empty intersection). This is where the seemingly complicated shape of the types of $x, \widetilde{y}$ comes from. The type of $x$ further reflects the fact that it is the subject of the output, so it must contain at least the type sequence $(A_1, \dots, A_n)$. To give a couple of concrete examples, we have that $\overline{x}\langle y, z, z\rangle$ may be typed in context $x : (A_1, A_2, A_3), y : A_1, z : A_2 \wedge A_3, w : \top$, whereas $\overline{x}\langle x, z, z\rangle$ may be typed in context $x : (A_1, A_2, A_3) \wedge A_1, z : A_2 \wedge A_3, w : \top$.

We have not yet commented on input declarations, and why they are of the form $(x; X) : A$ rather than just $x : A$. The reason is that we want to avoid dependency cycles: the pair $(x; X)$ indicates, as mentioned above, that there are outputs whose subjects are in $X$ which are guarded by an input prefix whose subject is $x$. The idea then is to maintain such information so that $x \in X$ implies that we have an i/o cycle, which explains the side condition of the res rule. At the same time, the res rule updates the dependency information by stating that, if in $P$ the names $X$ depend on $x$ and $x$ depends on $w$, then in $\nu x P$ the names $X$ depend on $w$. The par rule also updates the information by joining dependencies.

## 3.2 Typability and Complete Typability

As stressed in the Introduction, a process $P$ being typable in our type system does *not* imply that $P$ cannot exhibit any bad behavior, *i.e.*, that $P$ is well-behaved. This section is devoted to introducing the notion of *complete* typability, which will be the one characterizing good behavior. As such, because of the result of Sect. 2.4, it cannot be turned into a recursively enumerable predicate on processes, but relies instead on the existence of (potentially) infinitely many type derivations for the process under consideration, say $P$. Each of these derivations is a "witness" of the good behavior of a possible execution of $P$. If all possible executions have a witness, we say that $P$ is *completely typable*. In order to formalize this idea, we need to say what it means for a type derivation to *match* a reduction sequence, which in turn requires us to recall a few standard concepts of rewriting theory (essentially, the notion of *ancestor*).

*Definition 3.1 (Typability).* A process $P$ is *typable* if there exists a derivation of $\Gamma \vdash P :: \Delta$ using the rules of Fig. 1, such that, for all $x : A$ in $\Delta$ and $(x; X) : B$ in $\Gamma$, it holds that $B = A \wedge C$ for some $C$.

It is not hard to show that typability is stable under structural congruence: $P$ typable and $P \equiv Q$ implies $Q$ typable, in the same context. This fact will be tacitly used in the sequel.

We will now define the concept of *ancestor*, a standard notion in rewriting theory. In what follows, we will use $\sigma, \rho$ to range over reduction sequences, and we will write $\rho : P \longrightarrow^* Q$, or $P \overset{\rho}{\longrightarrow}{}^* Q$ to mean that $\rho$ is a reduction sequence from $P$ to $Q$.

*Definition 3.2 (Prefix Occurrence).* An *output prefix*, denoted by $o$, is of the form $\overline{x}\langle\widetilde{y}\rangle$; an *input prefix*, denoted by $\iota$, is of the form $x(\widetilde{y})$ or $!x(\widetilde{y})$. Input and output prefixes are generically denoted by $\pi$. A *prefix context* is defined as follows:

$$\mathrm{C} ::= \{\cdot\} \;\Big|\; \{\cdot\}.P \;\Big|\; x(\widetilde{y}).\mathrm{C} \;\Big|\; !x(\widetilde{y}).\mathrm{C} \;\Big|\; \mathrm{C} \mid P \;\Big|\; P \mid \mathrm{C} \;\Big|\; \nu x\mathrm{C}.$$

Note that a prefix context C contains exactly one occurrence of $\{\cdot\}$, the *hole*; we say that C is an *input context* if the hole occurs as $\{\cdot\}.P$ inside C, otherwise it is an *output context.* A *prefix occurrence* is a pair $(\pi, \mathrm{C})$ such that $\pi$ is an input (resp. output) prefix and C an input (resp. output) prefix context. We denote by $\mathrm{C}\{\pi\}$ the process obtained by substituting $\pi$ for the hole $\{\cdot\}$ in C. Given a process $P$, the set of prefix occurrences of $P$ is

$$\mathsf{pref}(P) := \{(\pi, \mathrm{C}) \mid \mathrm{C}\{\pi\} = P\}.$$

Note that $P \equiv Q$ implies $\mathsf{pref}(P) \cong \mathsf{pref}(Q)$, *i.e.*, the sets of prefix occurrences of the two processes are in bijection and such a bijection is uniquely determined. In what follows, for simplicity we will identify a prefix occurrence $(\pi, \mathrm{C})$ with the prefix $\pi$, unless this generates confusion.

Consider now the basic reduction rules of AHL$\pi$:

$$\begin{aligned} \overline{x}\langle\widetilde{y}\rangle \mid x(\widetilde{z}).P &\longrightarrow P\{\widetilde{y}/\widetilde{z}\}, \\ \overline{x}\langle\widetilde{y}\rangle \mid !x(\widetilde{z}).P &\longrightarrow P\{\widetilde{y}/\widetilde{z}\} \mid !x(\widetilde{z}).P, \\ \nu x(!x(\widetilde{z}_1).P_1 \mid \cdots \mid !x(\widetilde{z}_n).P_n) &\longrightarrow \mathbf{0}. \end{aligned}$$

Call $L$ and $R$ the left and right hand sides of the rules, respectively. It is clear that every prefix appearing in $R$ is a renaming of a unique prefix appearing in $L$. More formally, we may define a function $\mathrm{Anc} : \mathsf{pref}(R) \to \mathsf{pref}(L)$ such that: if $\pi \in \mathsf{pref}(P\{\widetilde{y}/\widetilde{z}\})$, then $\mathrm{Anc}(\pi) = \sigma$ where $\sigma$ is the unique prefix in $\mathsf{pref}(P)$ (as a subprocess of $L$) such that $\pi = \sigma\{\widetilde{y}/\widetilde{z}\}$; if $\pi \in \mathsf{pref}(!x(\widetilde{z}).P)$, then $\mathrm{Anc}(\pi) = \pi$ as a prefix of $L$, *i.e.*, $\pi$ is mapped to "itself".

The function Anc above may be extended to a function $\mathrm{Anc}_\rho : \mathsf{pref}(Q) \to \mathsf{pref}(P)$ for every one-step reduction $\rho : P \longrightarrow Q$, by simply setting it to be the "identity" on prefixes not directly involved in the communication. (This is well-defined because, as observed above, $\mathrm{pref}(\cdot)$ is stable under structural equivalence, modulo a unique bijection). Furthermore, if we have a reduction sequence of length $n$

$$\rho : P \xrightarrow{\rho_1} \cdots \xrightarrow{\rho_n} Q,$$

we may define $\mathrm{Anc}_\rho : \mathrm{pref}(Q) \to \mathrm{pref}(P)$ by setting $\mathrm{Anc}_\rho := \mathrm{Anc}_{\rho_1} \circ \cdots \circ \mathrm{Anc}_{\rho_n}$. Note that, if we denote composition of reductions $\rho : P \longrightarrow^* Q$, $\sigma : Q \longrightarrow^* R$ by $\rho;\sigma$, then $\mathrm{Anc}_{\rho;\sigma} = \mathrm{Anc}_\rho \circ \mathrm{Anc}_\sigma$.

*Definition 3.3 (Ancestor).* Let $\rho : P \longrightarrow^* Q$ be a reduction sequence, and let $\pi \in \mathrm{pref}(Q)$. The prefix $\mathrm{Anc}_\rho(\pi) \in \mathrm{pref}(P)$ is called the *ancestor* of $\pi$ along $\rho$.

Note that, because of duplication, the ancestor function need not be injective: for instance, in

$$\rho : \nu x(\overline{x}\langle a\rangle \mid \overline{x}\langle b\rangle \mid !x(z).\overline{z}) \longrightarrow^* \overline{a} \mid \overline{b},$$

the output particles $\overline{a}$ and $\overline{b}$ on the right have the same ancestor $\overline{z}$. It is interesting to observe that our type system has a very precise way of accounting for such duplications. For instance, the left hand side above may be typed with the following derivation, which we call $\delta$:

$$\dfrac{\dfrac{\dfrac{}{(x;) : \top \vdash \overline{x}\langle a\rangle :: a : (), b : \top, x : (())}\,\text{out} \qquad \dfrac{}{(x;) : \top \vdash \overline{x}\langle b\rangle :: a : \top, b : (), x : (())}\,\text{out} \qquad \delta'}{\overline{\overline{(x;z) : (()) \wedge (()) \vdash \overline{x}\langle a\rangle \mid \overline{x}\langle b\rangle \mid !x(z).\overline{z} :: a : (), b : (), x : (()) \wedge (())}}\,\text{par}}{\vdash \nu x(\overline{x}\langle a\rangle \mid \overline{x}\langle b\rangle \mid !x(z).\overline{z}) :: a : (), b : ()}\,\text{res}$$

where the double bar indicates several applications of the par rule and $\delta'$ is the following derivation:

$$\dfrac{\dfrac{}{\vdash \overline{z} :: z : (), a : \top, b : \top, x : \top}\,\text{out} \qquad \dfrac{}{\vdash \overline{z} :: z : (), a : \top, b : \top, x : \top}\,\text{out}}{(x;z) : (()) \wedge (()) \vdash !x(z).\overline{z} :: a : \top, b : \top, x : \top}\,!\text{in}$$

Observe that the output particle $\overline{z}$ (the common ancestor of $\overline{a}$ and $\overline{b}$) is typed *twice* in $\delta$, once for each branch of the binary !in rule in $\delta'$. This is because $\delta$ is somehow anticipating on the fact that $\overline{z}$ will be used twice during the reduction of the process it types.

Another very important remark is that $\delta$ contains precise information about the reduction $\rho$ itself. This is composed of two communication steps, one concerning $\overline{x}\langle a\rangle$ and the other $\overline{x}\langle b\rangle$, plus a clean-up step. Disregarding the latter (which has no computational value), the two communication steps of $\rho$ are visible in $\delta$ by the fact that $x$ receives twice the type $(())$ as output, once as the subject of $\overline{x}\langle a\rangle$ and once as the subject of $\overline{x}\langle b\rangle$, and it receives the type $(()) \wedge (())$ as input, as the subject of $!x(z)$. The fact that output types and input types match is not a chance but is a crucial phenomenon which is at the heart of our type system, and which we now proceed to make formal.

In what follows, we write $\delta :: P$ to mean that $\delta$ is a type derivation concluding with a judgment typing $P$.

*Definition 3.4 (Matching).* Let $\rho : P \longrightarrow^* Q$ and let $\sigma : Q \longrightarrow Q'$ be a communication step concerning an output prefix $o$ and an input prefix $\iota$. Let $\delta :: P$. We say that $\delta$ *matches* $\sigma$ if there exists a type $A$ such that:

(1) if $x$ is the subject of $o' := \mathrm{Anc}_{\rho;\sigma}(o)$, $\delta$ contains a rule out whose conclusion is $\Gamma \vdash o' :: \Delta, x : A$;
(2) if $y$ is the subject of $\iota' := \mathrm{Anc}_{\rho;\sigma}(\iota)$, $\delta$ contains a rule in or !in whose conclusion is $\Gamma, (y;Y) : A \wedge B \vdash \iota'.Q :: \Delta$ for some type $B$ (which may be $\top$).

Let $\delta :: P$ and $\rho : P \longrightarrow^* Q$. We say that $\delta$ *matches* $\rho$, and we write $\delta \bowtie \rho$, if $\delta$ matches every communication step fired along $\rho$.

Note that any derivation for $P$ matches the identity reduction on $P$. Also observe that if $\delta \bowtie (\rho;\sigma)$ then $\delta \bowtie \rho$. We are now ready to give the most important definition of the whole paper:

*Definition 3.5 (Complete Typability).* A process $P$ is *completely typable* if, for all $\rho : P \longrightarrow^* Q$, there exists $\delta :: P$ such that $\delta \bowtie \rho$.

Note that complete typability implies typability: simply let $\rho$ be the identity reduction on $P$.

### 3.3 Complete Typability Captures Good Behavior

This Section is technically the most relevant of the present paper, proving that complete typability is a necessary *and* sufficient condition for well-behaved processes. The first step towards our goal is to prove four key properties of our type systems. On the one hand, idle are typable, while typable processes are necessarily error-free. On the other hand, typability is preserved by anti-reduction, and (only existentially) by reduction. In what follows, we write $|\delta|$ for the size of a derivation (number of rules in it).

Proposition 3.6. *The type system enjoys the following properties:*

**Idle Typability:** *idle processes are typable;*
**Error-Freedom:** *a typable process is error-free;*
**Subject Expansion:** *if $P$ is typable and $Q \longrightarrow P$, then $Q$ is also typable;*
**Quantitative, Existential Subject Reduction:** *given $\delta :: P$ with $P$ reducible, there exists $\rho \colon P \longrightarrow Q$ with $\delta \bowtie \rho$ and there exists a derivation $\varepsilon :: Q$ such that $|\varepsilon| < |\delta|$.*

Proof. We prove each of the four properties separately:

**Idle Typability.** Let $P \equiv\, !x_1(\widetilde{y_1}).Q_1 \mid \dots \mid\, !x_l(\widetilde{y_l}.Q_l) \mid z_1(\widetilde{w_1}).R_1 \mid \dots \mid z_m(\widetilde{w_m}.R_m) \mid \overline{u_1}\langle\widetilde{v_1}\rangle \mid \dots \mid \overline{u_n}\langle\widetilde{v_n}\rangle$ be an arbitrary idle process ($l, m, n$ may be 0). It suffices to give one derivation for $P$; we always have the following derivation.

$$\frac{\gamma_1 \quad \dots \quad \gamma_l \quad \delta_1 \quad \dots \quad \delta_m \quad \varepsilon_1 \quad \dots \quad \varepsilon_n}{(\widetilde{x;\mathrm{out}(Q)} \setminus \widetilde{y}) \colon \top, (\widetilde{z;\mathrm{out}(R)} \setminus \widetilde{w}) \colon * \vdash P :: \widetilde{a} \colon \top, \widetilde{b} \colon \top, \widetilde{s} : \widetilde{C}}\ \mathsf{par}^*$$

where par$^*$ is a repeated application of par rule, each $\gamma_i$ is

$$\frac{}{(x_i; \mathrm{out}(Q_i) \setminus \widetilde{y_i}) : \top \vdash\, !x_i(\widetilde{y_i}).Q_i :: \widetilde{a^i} : \top}\ \mathsf{!in} \quad ,$$

each $\delta_i$ is

$$\frac{}{(z_i; \mathrm{out}(R_i) \setminus \widetilde{w_i}) : * \vdash\, !z_i(\widetilde{w_i}).R_i :: \widetilde{b^i} : \top}\ \mathsf{in}_* \quad ,$$

each $\varepsilon_i$ is

$$\frac{}{\vdash \overline{u_i}\langle\widetilde{v_i}\rangle :: u_i : A^i, s^i_1 : C^i_1, \dots, s^i_k : C^i_k}\ \mathsf{out} \quad ,$$

and contexts in $\varepsilon_i$'s are as in the explanation after Fig. 1.

**Error-Freedom.** It suffices to show that any process with a runtime error (arity mismatch, dependency cycle, failed send, or endless wait) is untypable. Arity mismatch is untypable by the definition of types: the rules out, in, in$_*$, and !in necessarily introduce pre-types with one or more tuples with the same arity as the input/output prefix. If arity mismatch is in a process, then a name in the context must be typed by a pre-type $A$ that does not satisfy $A \frown A$ since some sub-pre-types of $A$ have different arity.

A dependency cycle at the top level is not typable. Because any such process contains a subprocess in the form

$$\nu(x_1, \dots, x_n)(\dagger x_1(\widetilde{y_1}).Q_1 \mid \cdots \mid \dagger x_n(\widetilde{y_n}).Q_n \mid R)$$

where $\dagger x_i(\widetilde{y_i})$ denotes either $x_i(\widetilde{y_i}).Q_i$ or $!x_i(\widetilde{y_i}).Q_i$, each $x_i$ satisfies $x_i \in \mathrm{fn}(Q_{i+1})$ for $1 \le i < n$ and $x_n \in \mathrm{out}(Q_1)$, the rules in (or in$_*$) and !in that derive those input particles necessarily introduces $(x_1; x_n, \widetilde{p_1})$ and $(x_i; x_{i-1}, \widetilde{p_i})$ for $1 < i \le n$. It is easy to show that res rule on all $x_1, \dots, x_n$ over such a subprocess $\dagger x_1(\widetilde{y}_1).Q_1 \mid \cdots \mid \dagger x_n(\widetilde{y}_n).Q_n \mid R$ is impossible (by induction on $n$): restricting $x_1$ with $(x_1; x_1, \widetilde{p}_1)$ is directly prohibited by the side-condition of res rule; when we have $(x_1; x_n, \widetilde{p}_1)\colon A_1, (x_2; x_1, \widetilde{p}_2)\colon A_2, \dots, (x_n; x_{n-1}, \widetilde{p}_n)\colon A_n$ in a context, applying res rule on (without loss of generality) $x_n$, we will obtain $(x_1; x_{n-1}, \widetilde{p}_n, \widetilde{p}_1)\colon A_1, (x_2; x_1, \widetilde{p}_2)\colon A_2, \dots, (x_{n-1}; x_{n-2}, \widetilde{p}_{n-1})\colon A_{n-1}$ that is impossible to apply res rule over $x_1, \dots, x_{n-1}$ by I.H.

Failed send cannot be typed because a typing derivation for a failed-send process must contain a subderivation that types a process $Q \equiv \overline{x}\langle\widetilde{y^1}\rangle \mid \cdots \mid \overline{x}\langle\widetilde{y^n}\rangle$. The process $Q$ can only be typed as $\widetilde{w} : \top \vdash \overline{x}\langle\widetilde{y^1}\rangle \mid \cdots \mid \overline{x}\langle\widetilde{y^n}\rangle :: \Sigma$ where $x$ cannot be typed by $\top$ in $\Sigma$; since $\widetilde{w} : \top$ cannot contain any type other than $\top$, the rule res cannot be applied to the derived judgment.

Endless wait is untypable by a similar reason: to type $\nu x(x(\widetilde{y^1}).Q_1 \mid \cdots \mid x(\widetilde{y^n}).Q_n)$, due to syntax-directedness of the type system $x(\widetilde{y^1}).Q_1 \mid \cdots \mid x(\widetilde{y^n}).Q_n$ must first be typed, say $\Gamma \vdash x(\widetilde{y^1}).Q_1 \mid \cdots \mid x(\widetilde{y^n}).Q_n :: \Sigma$, with the name $x$ contained in the two contexts $\Gamma$ and $\Sigma$. Each application of in rule for $x(\widetilde{y^i}).Q_i$ necessarily types $x$ by a non-$\top$ type in the context $\Gamma$; however $x$ in the context $\Sigma$ can only be typed by $\top$, hence the types of $x$ in $\Gamma$ and that in $\Sigma$ never match and res rule cannot be applied.

**Subject Expansion.** Suppose $Q$ is typable and $\rho\colon P \longrightarrow Q$.

- If the reduction is by the rule $\overline{x}\langle\widetilde{y}\rangle \mid x(\widetilde{z}).R \longrightarrow R\{\widetilde{y}/\widetilde{z}\}$, by the syntax-directedness the derivation (call it $\delta$) for $Q$ must contain a subderivation $\delta'$ in the following form:

$$\dfrac{\begin{array}{c}\delta'\\ \vdots\end{array}}{\Gamma \vdash R\{\widetilde{y}/\widetilde{z}\} :: \Sigma.}$$

Since $R$ is inside an input particle in the redex, $R$ does not have a free occurrence of input particle and thus $\Gamma$ can only contain $\top$ types. Since the redex and the reduct are given by the reduction, we know every output prefixes $\overline{v_i}\langle\widetilde{w}_i\rangle$ that is renamed under the reduction $\rho$, and thus we can spot all the sub-derivations

$$\dfrac{}{\Gamma_i \vdash \overline{v_i}\langle\widetilde{w}_i\rangle :: v_i : A^i, u^i_1 : C^i_1, \dots, u^i_m : C^i_m}\ \text{out}$$

in $\delta'$ where $\Gamma_i$ only contains $\top$ types and the subderivation is associated with a sequence of types $\widetilde{B}^i$ and a function $f\colon \{1, \dots, |w_i|\} \to \{u^i_1, \dots, u^i_m\}$ that designates the types $A^i, C^i_1, \dots, C^i_m$. Some of the names $v_i, u^i_1, \dots, u^i_m$ have been renamed under $\rho$; consider the set $X = \{u'^i_1, \dots, u'^i_l\}$ that is defined by

$u'^i_1 \in X$ if $u'^i_1 \in \{v_i, u^i_1, \dots, u^i_m\}$ and $u'^i_1$ is not renamed under $\rho$;

$z_j \in X$ if $u^i_1 = y_j \in \widetilde{y}$ and the occurrence of $z_j$ is renamed by $y_j$ in $\overline{v_i}\langle\widetilde{w}_i\rangle$ under $\rho$

and define $f'\colon \{1, \dots, |w_i|\} \to \{u'^i_1, \dots, u'^i_l\}$ by

$f'(j) = f(j)$ if the $j$-th occurrence of name in $\widetilde{y}$ is not renamed under $\rho$;

$f'(j) = z_j$ if the $j$-th occurrence of name in $\widetilde{z}$ is renamed by $y_j$ under $\rho$.

Using $f'$ and erasing $\Gamma_i$, we obtain the following subderivation:

$$\dfrac{}{\vdash \mathrm{Anc}(\overline{v_i}\langle\widetilde{w}_i\rangle) :: u'^i_1 : C'_1, \dots, u'^i_l : C'_l}\ \text{out}$$

where $C'_j$'s are as specified by $\widetilde{B}$ and $f'$. Here $\Gamma_i$ must be erased because we later apply a in rule; it can be safely erased since all the types in $\Gamma_i$ are $\top$, which means that $\Gamma_i$ is introduced by (co)weakening. Let $\gamma'$ be a derivation obtained by replacing every such subderivation in $\delta'$. Note that such a replacement makes $z_i$'s appear in the contexts. Then we have

$$\dfrac{\dfrac{}{\vdash \overline{x}\langle\widetilde{z}\rangle :: x\colon (\widetilde{D}), \widetilde{s}\colon \widetilde{D}}\,\text{out} \qquad \dfrac{\begin{array}{c}\gamma'\\ \vdots\\ \vdash R :: \Sigma', \widetilde{z}\colon \widetilde{D}\end{array}}{(x; \mathrm{out}(R)\setminus\widetilde{z}) : (\widetilde{D}) \vdash x(\widetilde{z}).R :: \Sigma'}\,\text{in}}{(x;\mathrm{out}(R)\setminus\widetilde{z}) : E \vdash \overline{x}\langle\widetilde{y}\rangle.R \mid x(\widetilde{z}).R :: \Sigma', x\colon(\widetilde{D}), \widetilde{s}\colon \widetilde{D}}\,\text{par}$$

By replacing $\delta'$ in the original derivation $\delta$ by the derivation above, we obtain a derivation for $P$. Note that this derivation, say $\varepsilon$, satisfies $\varepsilon \bowtie P$.

- Similarly, if the reduction is by the rule $\overline{x}\langle\widetilde{y}\rangle \mid !x(\widetilde{z}).P \longrightarrow P\{\widetilde{y}/\widetilde{z}\} \mid !x(\widetilde{z}).P$, we can replace the subderivation for $P\{\widetilde{y}/\widetilde{z}\} \mid !x(\widetilde{z}).P$ by the one for $\overline{x}\langle\widetilde{y}\rangle \mid !x(\widetilde{z}).P$: the only difference is that now we do not newly put !in rule to the subderivation, but add one premise to the existing application of !in rule.
- If the reduction is by the rule $\nu x(!x(\widetilde{z}_1).P_1 \mid \cdots \mid !x(\widetilde{z}_n).P_n) \longrightarrow \mathbf{0}$, the process $\nu x(!x(\widetilde{z}_1).P_1 \mid \cdots \mid !x(\widetilde{z}_n).P_n)$ can be typed by the following derivation and it can be inserted by par rule to the place where the redex is in $P$:

$$\dfrac{\dfrac{\dfrac{}{(x;\mathrm{out}(P_1)\setminus\widetilde{z}_1)\colon \top \vdash !x(\widetilde{z}_1).P_1 :: x\colon \top}\,\text{!in} \quad \cdots \quad \dfrac{}{(x;\mathrm{out}(P_n)\setminus\widetilde{z}_n)\colon \top \vdash !x(\widetilde{z}_n).P_n ::}\,\text{!in}}{(x; \bigcup_i \mathrm{out}(P_i)\setminus \bigcup_i \widetilde{z}_i)\colon \top \vdash !x(\widetilde{z}_1).P_1 \mid \cdots \mid !x(\widetilde{z}_n).P_n :: x\colon \top}\,\text{par}*}{\vdash \nu x(!x(\widetilde{z}_1).P_1 \mid \cdots \mid !x(\widetilde{z}_n).P_n) ::}\,\text{res} \qquad .$$

Since this derivation has empty contexts, inserting it in the original derivation does not have any effect on validity of the derivation.

**Quantitative, Existential Subject Reduction.** First we need a sub-lemma (proof is by structural induction).

Lemma 3.7. *Let $\delta :: P$. Then $\delta\{\widetilde{x}/\widetilde{y}\} :: P\{\widetilde{x}/\widetilde{y}\}$, where $\delta\{\widetilde{x}/\widetilde{y}\}$ is defined by renaming $\widetilde{y}$ by $\widetilde{x}$ in each processes and contexts in the derivation (if both $z$ and $v$ are renamed by $w$ and $z\colon A, v\colon B$ in a context, replace them by $w\colon A \wedge B$).*

Existence of $\rho : P \longrightarrow Q$ matching $\delta$ is a consequence of the typability condition (Definition 3.1) and the side condition on the res rule, which ensure that every output type is matched by an input type. Consider then such a $\rho$.

- Suppose that the reduction is by the rule $\overline{x}\langle\widetilde{y}\rangle \mid x(\widetilde{z}).R \longrightarrow R\{\widetilde{y}/\widetilde{z}\}$. Since $\delta \bowtie \rho$, the derivation $\delta$ must contain a subderivation $\delta'$ in the following shape:

$$\dfrac{\dfrac{}{\vdash \overline{x}\langle\widetilde{y}\rangle :: x\colon (\widetilde{A}), \widetilde{w}\colon \widetilde{A}'}\,\text{out} \qquad \dfrac{\begin{array}{c}\gamma\\ \vdots\\ \vdash R :: \widetilde{z}\colon \widetilde{A}, \Sigma\end{array}}{(x; \mathrm{out}(R)\setminus\widetilde{z}) : (\widetilde{A}) \vdash x(\widetilde{z}).R :: \Sigma}\,\text{in}}{(x;\mathrm{out}(R)\setminus\widetilde{z}) : (\widetilde{A}) \vdash \overline{x}\langle\widetilde{y}\rangle \mid x(\widetilde{z}).R :: x\colon(\widetilde{A}), \widetilde{w}\colon \widetilde{A}', \Sigma}\,\text{par}$$

where $\widetilde{w}$ is the set of names obtained by removing duplicates in $\widetilde{y}$, the type $(\widetilde{A})$ of $x$ is shared in the two contexts because $\delta \bowtie \rho$. By replacing $\delta'$ in the whole derivation $\delta$ by the derivation $\gamma\{\widetilde{y}/\widetilde{z}\}$, we obtain a valid derivation with two less applications of rules. The derivation $\gamma\{\widetilde{y}/\widetilde{z}\}$, and the derivation obtained by replacing $\delta'$ in $\delta$ by $\gamma\{\widetilde{y}/\widetilde{z}\}$ satisfies that: every contexts below $\gamma\{\widetilde{y}/\widetilde{z}\}$ except $x$ are the same as $\delta$, and the type of $x$ (if any) is $B$ where $x$ is typed by $(\widetilde{A}) \wedge B$ in $\delta$. Especially, when applying res rule, the condition that the types of the same name match is satisfied as long as it is in the original derivation $\delta$.

- Similarly, if the reduction $\rho\colon P \longrightarrow Q$ is by the rule $\overline{x}\langle\widetilde{y}\rangle \mid !x(\widetilde{z}).R \longrightarrow R\{\widetilde{y}/\widetilde{z}\} \mid !x(\widetilde{z}).R$, the derivation $\delta$ must contain

$$\dfrac{\dfrac{}{\vdash \overline{x}\langle\widetilde{y}\rangle :: x\colon (\widetilde{A^j}), \widetilde{w}\colon \widetilde{A'^j}}\,\text{out} \qquad \dfrac{\begin{matrix}\gamma_1\\ \vdots\\ \vdash R :: \widetilde{z}\colon \widetilde{A^1}, \Sigma_1\end{matrix} \quad \cdots \quad \begin{matrix}\gamma_n\\ \vdots\\ \vdash R :: \widetilde{z}\colon \widetilde{A^n}, \Sigma_n\end{matrix}}{(x;\widetilde{p}) : \bigwedge_i(\widetilde{A^i}) \vdash !x(\widetilde{z}).R :: \bigwedge_i \Sigma_i}\,!\text{in}}{(x;\widetilde{p}) : \bigwedge_i(\widetilde{A^i}) \vdash \overline{x}\langle\widetilde{y}\rangle \mid !x(\widetilde{z}).R :: x\colon (\widetilde{A^j}), \widetilde{w}\colon \widetilde{A'^j}, \bigwedge_i \Sigma_i}\,\text{par}$$

and replacing it in $\delta$ by the following derivation $\gamma'$ yields a valid derivation for $Q$ with exactly one less applications of rules. As in the former case, contexts are preserved except the name $x$ loses one intersection in its type, hence all the other applications of rules present in $\delta$ remain valid.

$$\dfrac{\dfrac{}{\vdash R\{\widetilde{y}/\widetilde{z}\} :: \widetilde{w}\colon \widetilde{A'^j}}\,\text{out} \qquad \varepsilon}{(x;\widetilde{p}) : \bigwedge_{i\neq j}(\widetilde{A^i}) \vdash R\{\widetilde{y}/\widetilde{z}\} \mid !x(\widetilde{z}).R :: x\colon (\widetilde{A^j}), \widetilde{w}\colon \widetilde{A'^j}, \bigwedge_{i\neq j} \Sigma_i}\,\text{par}$$

where $\varepsilon$ is

$$\dfrac{\begin{matrix}\gamma_1\\ \vdots\\ \vdash R :: \widetilde{z}\colon \widetilde{A^1}, \Sigma_1\end{matrix} \quad \cdots \quad \begin{matrix}\gamma_{j-1}\\ \vdots\\ \vdash R :: \widetilde{z}\colon \widetilde{A^{j-1}}, \Sigma_{j-1}\end{matrix} \quad \begin{matrix}\gamma_{j+1}\\ \vdots\\ \vdash R :: \widetilde{z}\colon \widetilde{A^{j+1}}, \Sigma_{j+1}\end{matrix} \quad \cdots \quad \begin{matrix}\gamma_n\\ \vdots\\ \vdash R :: \widetilde{z}\colon \widetilde{A^n}, \Sigma_n\end{matrix}}{(x;\widetilde{p}) : \bigwedge_{i\neq j}(\widetilde{A^i}) \vdash !x(\widetilde{z}).R :: \bigwedge_{i\neq j} \Sigma_i}\,!\text{in}$$

□

Corollary 3.8. *A process $P$ is typable iff there exists an idle process $I$ such that $P \longrightarrow^* I$.*

Proof. **(if)** Suppose there exists an idle process $I$ such that $P \longrightarrow^* I$. Then, since $I$ is typable by Proposition 3.6, $P$ is itself typable by subject expansion in Proposition 3.6.

**(only if)** Suppose $P$ is typable by a derivation $\delta$. By quantitative, existential subject reduction in Proposition 3.6, there exists a reduction sequence $P \longrightarrow Q_1 \longrightarrow \cdots Q_k$ where each $Q_i$ is typable for $1 \leq k$, the last process $Q_k$ does not reduce, and $k \leq |\delta|$. Since a typable process is error-free by Proposition 3.6, $Q_k$ is a error-free irreducible process, *i.e.*, idle.

□

Lemma 3.9. *Let $P$ be completely typable and let $P \longrightarrow Q$. Then, $Q$ is completely typable.*

Proof. Let $\rho\colon P \longrightarrow Q$ and $\sigma\colon Q \longrightarrow^* R$. Then we have a sequence of reductions $\rho;\sigma$ starting from $P$. Since P is completely typable, there exists a derivation $\delta$ that satisfies $\delta :: P$ and $\delta \bowtie \rho$. As in the proof of the existential subject reduction in Proposition 3.6, we can modify the derivation $\delta$ to another derivation $\delta'$ for $Q$. Since the modification only renames the rest of derivation, the prefixes appearing in redexes in $\sigma$ still satisfy the condition in Definition 3.4, and hence $\delta' \bowtie \sigma$ for any $\sigma\colon Q \longrightarrow^* R$. □

Note that subject expansion of Proposition 3.6 is constructive: given $\rho : P \longrightarrow^* Q$, every derivation $\varepsilon :: Q$ induces a derivation $\rho^*(\varepsilon) :: P$, which we call the *pullback of $\varepsilon$ along $\rho$*.

Lemma 3.10 (pullback). *Let $\rho : P \longrightarrow^* Q$. Then, for all $\varepsilon :: Q$, $\rho^*(\varepsilon) \bowtie \rho$.*

Proof. Let $\rho = \sigma;\sigma'$ where $\sigma\colon P \longrightarrow^* Q'$ and $\sigma'\colon Q' \longrightarrow Q$, and let $\varepsilon :: Q$. The derivation for $Q'$ we constructed from $\varepsilon$ in the proof of subject expansion indeed matches $\sigma'$; by induction on the length of $\rho$ we obtain a derivation that matches the whole $\rho$ that we take as $\rho^*(\varepsilon)$. □

THEOREM 3.11. *A process is completely typable iff it is well-behaved.*

PROOF. **(if)** Let $P$ be a well-behaved process and $\rho\colon P \longrightarrow^* Q$ be an arbitrary sequence of reductions. Since $P$ is well-behaved, there must be an idle process $I$ satisfying $\sigma\colon Q \longrightarrow^* I$. By Proposition 3.6, $I$ is typable by some derivation $\delta$; by Lemma 3.10, there exists a derivation $\sigma^*(\delta)$ that matches $\sigma$. By repeatedly applying Lemma 3.10 we obtain $\rho^*(\sigma^*(\delta))$ that matches $\rho;\sigma$, and hence $\rho$.

**(only if)** Let $P$ be completely typable. Then for any sequence of reductions $\rho\colon P \longrightarrow^* Q$, $Q$ is also completely typable by Lemma 3.9, and thus $Q$ is typable. By Proposition 3.6 $Q$ is error-free, and there exists another sequence of reductions $\sigma\colon Q \longrightarrow^* R$ where $R$ is irreducible; since $\sigma;\rho$ is also a sequence of reductions starting from $P$, $R$ is also typable and hence error-free, *i.e.*, idle.

□

### 3.4 Examples

We now go back to the examples from Sect. 2.3 and look at them from the point of view of typing.

*3.4.1 Functional Programming.* In the case of functional programming, there is not much to be said: the encoding of a non-deterministic $\lambda$-term (or $\lambda\mu$-term) is such that no runtime error may ever arise, so good behavior reduces to termination of weak head reduction. Let us say that a term $M$ *may converge* if $M \longrightarrow^* N$ for some (weak head) normal form $N$; let us say that it *may diverge* if $M \longrightarrow^* \Omega$ for some term $\Omega$ having no normal form. Our type system gives us the standard result expected from intersection types: a functional process is typable (resp. completely typable) iff its evaluation under weak head reduction may converge (resp. may not diverge, *i.e.*, it is *safe*, in the terminology of Sect. 2.4).

*3.4.2 Locks.* Consider the final process of Sect. 2.3.2, *i.e.*, the lock together with the processes competing for it. We will now give a complete analysis of its possible type derivations.

Let $A_0 := \top$, $A_{i+1} := (A_i) \wedge A_i$. We saw above (just after the definition of type) that these are all types. We start by observing that we have

$$\dfrac{}{\vdash \overline{a}\langle a\rangle :: a : A_n}\;\mathsf{out}$$

for all $n > 0$. Let now $Q := \nu v(\overline{p}\langle v\rangle \mid v.\overline{z}\langle z\rangle)$, so that $L = !a(z).Q$. We have, for all $n > 0$

$$\dfrac{\dfrac{\dfrac{}{(v;\,) : \top \vdash \overline{p}\langle v\rangle :: z : \top, p : (()), v : ()}\;\mathsf{out} \qquad \dfrac{\dfrac{}{\vdash \overline{z}\langle z\rangle :: z : A_n, p : \top, v : \top}\;\mathsf{out}}{(v;z) : () \vdash v.\overline{z}\langle z\rangle :: z : A_n, p : \top, v : \top}\;\mathsf{in}}{(v;z) : () \vdash \overline{p}\langle v\rangle \mid v.\overline{z}\langle z\rangle :: z : A_n, p : (()), v : ()}\;\mathsf{par}}{\vdash Q :: z : A_n, p : (())}\;\mathsf{res}$$

and we also have

$$\dfrac{\dfrac{\dfrac{}{(v;\,) : \top \vdash \overline{p}\langle v\rangle :: z : \top, p : (*), v : *}\;\mathsf{out} \qquad \dfrac{}{(v;z) : * \vdash v.\overline{z}\langle z\rangle :: z : \top, p : \top, v : \top}\;\mathsf{in}_*}{(v;z) : * \vdash \overline{p}\langle v\rangle \mid v.\overline{z}\langle z\rangle :: z : \top, p : (*), v : *}\;\mathsf{par}}{\vdash Q :: z : \top, p : (*)}\;\mathsf{res}$$

Now, note that, for all $m \in \mathbb{N}$, $A_{m+1} = (A_m) \wedge \cdots \wedge (A_1) \wedge (A_0)$, and recall that $A_0 = \top$. Thanks to this observation, we have

$$\dfrac{\vdash Q :: z : A_m, p : (()) \quad \ldots \quad \vdash Q :: z : A_1, p : (()) \quad \vdash Q :: z : A_0, p : (*)}{(a;p) : A_{m+1} \vdash L :: p : (())^m \wedge (*)}\;\mathsf{!in}$$

for all $m \in \mathbb{N}$, where $(())^m$ is short for $(()) \wedge \cdots \wedge (())$ $m$ times. Combining the above, we get

$$\vdash \mathsf{Lock} :: p : (())^m \wedge (*)$$

for all $m \in \mathbb{N}$.

Suppose now that we have, for all $1 \leq i \leq n$,

$$\vdash Q_i :: v : (), p : \top$$

that is, $v$ is used as the subject of a nullary output in $Q_i$ (the lock release signal), $p$ is not used in $Q_i$ and, for simplicity, we assume that $Q_i$ has no other free name (for instance, we may simply take $Q_i := \overline{v}$, *i.e.*, $Q_i$ releases the lock as soon as it obtains it). We then have

$$\frac{\vdash Q_i :: v : (), p : \top}{(p;) : (()) \vdash P_i :: p : \top}\ \text{in}$$

so we finally obtain

$$\frac{(p;) : (()) \vdash P_1 :: p : \top \quad \ldots \quad (p;) : (()) \vdash P_n :: p : \top \quad (p;) : \top \vdash \mathsf{Lock} :: p : (())^m \wedge (*)}{(p;) : (())^n \vdash P_1 \mid \cdots \mid P_n \mid \mathsf{Lock} :: p : (())^m \wedge (*)}\ \text{par}$$

Note that, by syntax directedness and the type constraints on rules, these are all the type derivations for the lock with $n$ processes. The derivations in which $m = n$ are the most interesting: they match a full reduction to idle form, in which every $P_i$ has acquired the lock at some point and released it. The order in which the lock is acquired is given by the types $A_n, \ldots, A_1, A_0$. Indeed, channel $p$ (the lock channel) has input type $(())$ in each $P_i$ (this is visible in the last derivation); note that, in the derivation typing $L$, there are $n + 1$ copies of $Q$ in which $p$ gets output type $(())$, and each of these copies of $Q$ has some $z : A_j$ in its context, $0 \leq j \leq n$. The possible lock-acquisition orderings correspond to the possible matchings of the $i$-th input type $(())$ typing $P_i$ with the $j$-th output type $(())$ typing the $j$-th copy of $Q$. The 0-th copy of $Q$, in which $p : (*)$, corresponds to the last reduction $\mathsf{Lock} \longrightarrow \nu v(\overline{p}\langle v\rangle \mid v.\overline{z}\langle z\rangle \mid L)$ leading to the idle form after all $P_i$'s have terminated.

*3.4.3 Toy Operating System.* Let us give another example of how very precise information about the behavior of processes may be extracted from typing derivations. To make things more understandable, we first consider a (drastically) simplified version of the toy operating system of Sect. 2.3.3:

$$\begin{aligned}
\mathsf{SimpOpSys} &:= !r.\mathbf{0} \\
V(z) &:= \nu s(\overline{s} \mid !s.(\overline{r} \mid \overline{z}\langle z\rangle) \mid !s.\mathbf{0}) \\
\mathsf{SimpUser} &:= \nu u(\overline{u}\langle u\rangle \mid !u(z).V(z))
\end{aligned}$$

SimpOpSys offers just one service: receiving a “ping” on channel $r$. SimpUser pings the operating system indefinitely, and may decide at any time to log out. As we did for the lock process, we will now give a complete analysis of the behavior of the SimpOpSys/SimpUser system by means of a complete analysis of its type derivations.

First, for all $m \in \mathbb{N}$, we have

$$\frac{\dfrac{}{\vdash \mathbf{0} ::}\ \text{zero}}{(r;) : ()^m \vdash \mathsf{SimpOpSys} ::}\ \text{!in}$$

where, as in the previous section, $()^m$ stands for $() \wedge \cdots \wedge ()$ $m$ times. In fact, by syntax directedness, these are *all* the possible type derivations for SimpOpSys.

Before analyzing the simplified user, let us make a notational simplification: in par rules, we will discard unnecessary declarations of the form $(x;) : \top$ or $x : \top$, writing

$$\dfrac{(x;X) : A \vdash P :: z : C \qquad \vdash Q :: y : B, z : D}{(x;X) : A \vdash P \mid Q :: y : B, z : C \wedge D}\,\text{par}$$

instead of

$$\dfrac{(x;X) : A \vdash P :: y : \top, z : C \qquad (x;) : \top \vdash Q :: y : B, z : D}{(x;X) : A \vdash P \mid Q :: y : B, z : C \wedge D}\,\text{par}$$

which is the "official" version of the rule given in Fig. 1. The "compact" version of the rule is of course admissible because declarations of the form $(x;) : \top$ or $x : \top$ may be added at will to derivable judgments, preserving derivability (this is a standard property provable by induction).

So, for the simplified user, we have

$$\dfrac{\dfrac{\dfrac{}{\vdash \overline{s} :: s : ()}\,\text{out} \qquad \dfrac{\dfrac{\dfrac{}{\vdash \overline{r} :: r : ()}\,\text{out} \qquad \dfrac{}{\vdash \overline{z}\langle z\rangle :: z : A_{n+1}}\,\text{out}}{\vdash \overline{r} \mid \overline{z}\langle s\rangle :: z : A_{n+1}, r : ()}\,\text{par}}{(s; z, u) : () \vdash\, !s.(\overline{r} \mid \overline{z}\langle s\rangle) :: z : A_{n+1}, r : ()}\,!\text{in} \qquad \dfrac{}{(s;) : \top \vdash\, !s.\mathbf{0} ::}\,!\text{in}}{(s; z, r) : () \vdash \overline{s} \mid\, !s.(\overline{r} \mid \overline{z}\langle z\rangle) \mid\, !s.\mathbf{0} :: z : A_{n+1}, r : (), s : ()}\,\text{par}}{\vdash V(z) :: z : A_{n+1}, r : ()}\,\text{res}$$

and we also have

$$\dfrac{\dfrac{\dfrac{}{\vdash \overline{s} :: s : ()}\,\text{out} \qquad \dfrac{}{(s; z, u) : \top \vdash\, !s.(\overline{r} \mid \overline{z}\langle s\rangle) :: z : A_0, r : \top}\,!\text{in} \qquad \dfrac{\dfrac{}{\vdash \mathbf{0} ::}\,\text{zero}}{(s;) : () \vdash\, !s.\mathbf{0} ::}\,!\text{in}}{(s; z, r) : () \vdash \overline{s} \mid\, !s.(\overline{r} \mid \overline{z}\langle z\rangle) \mid\, !s.\mathbf{0} :: z : A_0, r : \top, s : ()}\,\text{par}}{\vdash V(z) :: z : A_0, r : \top}\,\text{res}$$

which exhausts all possibilities for $V(z)$. Besides syntax-directedness, the key point here is the final res rule, which forces the input and output type of $s$ to match. Because of the presence of a single output particle $\overline{s}$, the output type cannot be but $()$, hence the total arity of the two !in rules for $!s.(\overline{r} \mid \overline{z}\langle z\rangle)$ and $!s.\mathbf{0}$ must add up to 1. This corresponds to the fact that the output $\overline{s}$ is matched to exactly one of the two inputs, and the other input is discarded (which is possible because it is a server).

For SimpUser, we get

$$\dfrac{\dfrac{\dfrac{}{\vdash \overline{u}\langle u\rangle :: u : A_{n+1}}\,\text{out} \qquad \dfrac{\begin{matrix}\vdots\\ \vdash V(z) :: z : A_n, r : ()\end{matrix} \quad \ldots \quad \begin{matrix}\vdots\\ \vdash V(z) :: z : A_1, r : ()\end{matrix} \quad \begin{matrix}\vdots\\ \vdash V(z) :: z : A_0, r : \top\end{matrix}}{(u; r) : A_{n+1} \vdash\, !u(z).V(z) :: r : ()^n}\,!\text{in}}{(u; r) : A_{n+1} \vdash \overline{u}\langle u\rangle \mid\, !u(z).V(z) :: r : ()^n, u : A_{n+1}}\,\text{par}}{\vdash \mathsf{SimpUser} :: r : ()^n}\,\text{res}$$

which exhausts all possibilities because of the above discussion and because, again due to the last res rule, the input and output type of $u$ must match and the only ways to type $\overline{u}\langle u\rangle$ are with $u : A_{n+1}$ for $n \in \mathbb{N}$, where $A_n$ is the same family of types considered in the above section, defined by $A_0 = \top$, $A_{i+1} = (A_i) \wedge A_i$ and thus satisfying $A_{n+1} = (A_n) \wedge \cdots \wedge (A_1) \wedge (A_0)$.

Let us consider the simple case in which there is only one user. Thanks to the above, we have

$$(r;) : ()^m \vdash \mathsf{SimpUser} \mid \mathsf{SimpOpSys} :: r : ()^n$$

for all $m, n \in \mathbb{N}$. Each of these derivations describes a possible behavior of the user/system process. The integer $m$ corresponds to the number of times the user pings the system. The cases $m < n$ match reductions in which the user has not yet logged out; when $m \geq n$, the user has logged out and the idle form SimpOpSys has been reached. Similarly to the lock example, each typing of $V(z)$ in a derivation of SimpUser is labelled by an integer $0 \leq i \leq n$. The positive $i$'s correspond to pings by the user: $i = n$ is the first, $i = n - 1$ the second and so on. The case $i = 0$ corresponds to the log out. Therefore, each derivation of the user/system process corresponds to an evolution in which eventually the user logs out; however, when $m < n$, such a log out operation is still in the future and the derivation only matches the first $m$ pings.

A similar analysis may be performed for the toy operating system of Sect. 2.3.3. Without giving the details, we get

$$(r_1;) : (())^{k_1}, \ldots, (r_n;) : (())^{k_n} \vdash \mathsf{OpSys} ::$$

and

$$\vdash \mathsf{User} :: r_1 : B_1, \ldots, r_n : B_n$$

where each $B_i$ is of the form $(())^{l_i}$ and at most one of them may have the form $(())^{l_i} \wedge (*)$. If there are $m$ users, ranged over by $j$, we may write $B_i^j$ for the above types. In analogy with the simplified case, we have:

- the number $k_i$ corresponds to the number of times the $i$-th service is invoked;
- for a given user $j$, the number $l_i^j$ corresponds to the number of times the user invokes the $i$-th service;
- the presence of the type $(*)$ in one of the $B_i^j$'s corresponds to the fact that user $j$ has not yet logged out and that the user's last request was for the $i$-th service;
- as in the lock example, the order in which requests from the different users are met by the operating system may be inferred by the presence of types of the form $A_i$ in the derivation.

The details are slightly more complex (the derivations are bigger) but the essence is identical, which is why we do not give them as they would not add much to the understanding of the type system. The point is that, in certain cases, it is possible to perform a complete analysis of the type derivations of a process, a task which is facilitated by the syntax-directedness of the type system. From such an analysis, a complete description of the behavior of a process may be inferred.

## 4 DISCUSSION

*Where does our type system come from?* The type system presented here was not "pulled out of a hat" but obtained from a recently introduced construction [Mazza et al. 2018], which is based on the idea that intersection types come from approximations in linear logic. In a nutshell, one of the main consequences of the construction is that a programming language has an "intersection-flavored" type discipline as soon as it may be meaningfully translated in linear logic. For instance, Mazza et al. show how every major system of intersection types for the $\lambda$-calculus arises from applying their construction to Girard's translation of intuitionistic logic (and hence the $\lambda$-calculus) in linear logic.

It is known that the $\pi$-calculus may be translated in linear logic: several encodings and correspondences were proposed in the past decade or so [Caires and Pfenning 2010; Ehrhard and Laurent 2010; Honda and Laurent 2010; Wadler 2014]. For our work, we turned to Honda and Laurent's correspondence, but amended it in the light of Ehrhard and Laurent's encoding, which is more liberal (for instance, Honda and Laurent consider the *private* $\pi$-calculus and restrict to replicated input only, while no such restrictions are present in our case). The resulting encoding of the $\pi$-calculus in linear logic (using proof nets) is described in [de Visme and Mazza 2017]. Our type system results by applying, in a nearly automatic way, the construction of [Mazza et al. 2018] to this latter encoding. Albeit that work also contains a general theorem for inferring dynamic properties from the type systems built by the construction, the assumptions of the theorem do not cover concurrent languages like the $\pi$-calculus, so we had to prove our results "by hand" (in light of Sect. 2.4, it was impossible to be otherwise). Nevertheless, we think that this is a nice application of the methodology brought forth by that work: we would not have been capable of coming up with our type system without the help of the above-mentioned construction.

*Why asynchronous hyperlocalized?* The choice of $\pi$-calculus fragment in our work is dictated by its connection with polarized linear logic [Honda and Laurent 2010]. Let us try and give as much intuition as possible about the two restrictions (asynchrony and hyperlocalization), knowing that a fully technical explanation is impossible without introducing proof nets.

When encoding the $\pi$-calculus in linear logic, $\pi$-calculus reduction is simulated by *shallow* proof net reduction (also known as *surface* reduction in linear $\lambda$-calculi [Simpson 2005]). The concept of "shallowness" comes from the syntax of proof nets, which includes certain constructs called "boxes", associated with the exponential modality $!(-)$ and marking sub-proof nets that may be duplicated or erased. Shallow reduction never reduces inside a box. This is reminiscent of weak reduction in the $\lambda$-calculus, which does not reduce under an abstraction. Boxes are therefore a natural way of encoding the blocking behavior of prefixes in the $\pi$-calculus, *i.e.*, the fact that $\pi.P$ does not reduce even if $P$ may reduce. Recall however that a box is associated with the modality $!(-)$; so, if one kind of prefix (input or output) is associated with the presence of a box, then it will be associated with $!(-)$, which implies that the dual kind of prefix will be associated with $?(-)$, which *does not* come with a box. Therefore, matching input/output duality with linear logical duality forces one prefix to be blocking and the other to be non-blocking; since it makes little sense for input to be non-blocking, this leads straight to the asynchronous $\pi$-calculus.

Locality is a consequence of polarization. Like [Honda and Laurent 2010], we are using an encoding of the $\pi$-calculus in the *polarized* fragment of linear logic. This forces all names of $\widetilde{y}$ in a prefix like $x(\widetilde{y})$ to be of the same polarity, opposite to that of $x$. Since polarities coincide with the input/output distinction, we have that $\widetilde{y}$ must be composed of output names.

Hyperlocalization is a bit more technical and is a sort of "collateral damage" of the use of boxes in the encoding of input prefixes. Indeed, in the encoding of $x(\widetilde{y}).P$ (or $!x(\widetilde{y}).P$), the encoding of $P$ must be put "inside a box"; however, not every proof net may be put "inside a box": its conclusions (corresponding to the free names of $P$) must all be of negative polarity, and this happens to be the output polarity, hence the restriction.

It is natural to ask whether these restrictions may be overcome, typically by changing the encoding of the $\pi$-calculus in proof nets. This is trickier than what it seems. For what concerns hyperlocalization, there appears to be no alternative to the use of a box in encoding $!x(\widetilde{y}).P$; for the reasons mentioned above, this inevitably leads to the restriction that no free name of $P$ be used as subject of an input. The case of non-replicated input prefixes looks simpler to amend, but the fact that such prefixes are blocking still strongly suggests the use of boxes, leading to the same obstruction.

About lifting the asynchronous restriction, an obvious solution to encode synchronous communication would be to associate a combination of modalities with each prefix, something like $!?(-)$ (and its dual $?!(-)$), in such a way that a box is *always* present, so that both prefixes are blocking. Although possible in principle, this has the disadvantage of propagating the hyperlocality restriction to output prefixes, *i.e.*, in $\overline{x}\langle\widetilde{y}\rangle.P$, the free names of $P$ would be limited to output use only. Also, it is likely that, in such an encoding, the sequence $\widetilde{y}$ above would be forced to contain names used only as subjects of *input* prefixes. That this is not the case at present is precisely thanks to asynchrony. All these induced restrictions make it unclear whether we may meaningfully deal with a synchronous setting.

*What have we achieved and where to go from here?* To the best of our knowledge, this is the first application of intersection types to concurrency which is comparable, in terms of results, to the traditional applications of intersection types to functional programming, in the sense the we get a system which is both sound and complete for a given behavioral property. In particular, our type system characterizes "may termination". Granted, this is not a very useful property in a concurrent setting, but no other type system that we are aware of achieves this (*i.e.*, completeness).

The interesting property, good behavior, is hopelessly out of reach for a type system to capture, at least if we consider the usual meaning of "capturing" (*i.e.*, by means of existence of a type). In this respect, our type system is the best we can get: although no single, finite type derivation may ever capture good behavior, it does capture a fraction of it, and our type system is complete in the sense that it misses none of these fractions.

The theoretical limitation of Sect. 2.4 ($\Pi^0_2$-completeness) and Theorem 3.11 together make our type system unsuitable for practical usage, which of course is not a problem in the present context since we are interested in a theoretical exploration of the limits of intersection types for concurrent programming. However, it is important to observe that this does not prevent our work from having a potential impact on more practical investigations. Indeed, as Sect. 3.4 has hopefully shown, there are non-trivial cases in which it is possible to completely describe the type derivations for a process, which amounts to completely describing its behavior. Such a description could be a form of *parametric* derivation: in the examples of Sect. 3.4, the parameters were integers, but one may imagine more complex structures. This leads us towards a type discipline in the style of bounded linear logic [Girard et al. 1992], which is known to be related to intersection types (via the notion of approximation considered in [Mazza et al. 2018]). More precisely, it seems that a similar application of (linear) dependent types as that given by [Dal Lago and Gaboardi 2011] is possible here, which would yield more practical type systems, necessarily sacrificing completeness but hopefully still of remarkable expressiveness.

Another, perhaps more technical contribution of our work is that it clarifies the meaningfulness of encodings of $\pi$-calculi in linear logic. Indeed, it was shown that, for abstract reasons due to the absence of so-called *confusion* in differential proof nets (the target of both encodings of [Ehrhard and Laurent 2010; Honda and Laurent 2010]), no encoding fully respecting the parallelism of the $\pi$-calculus could possibly be sound. Accordingly, the encoding at the basis of our work [de Visme and Mazza 2017] is syntactically unsound: if $[\![\cdot]\!]$ denotes the encoding, there are processes $P$ such that $[\![P]\!]$ may exhibit reductions having no match in $P$. However, thanks to the hyperlocalization restriction, it turns out that these "faulty" reductions may only happen when $P$ is already "faulty" on its own, *i.e.*, when it has no idle form. By Corollary 3.8, $P$ is not typable, so the encoding is morally correct. This property is similar to that expressed by Proposition 2.8 and may be proved along the same lines.

One more thing worth noting is that the construction of [Mazza et al. 2018], which is based on approximations, is in fact also related to the Taylor expansion of differential linear logic and its relational semantics. This is visible in our types, which are essentially the elements of the relational semantics of proof nets. From this perspective, another way of stating that the lack of correctness of the proof net encoding is harmless is that "faulty" reductions only happen in proof nets/processes whose semantics is empty. More generally, the connection with denotational semantics, especially that with (linear) approximations, is another topic worth investigating further. Finally, connections between our type system and Girard's Geometry of Interaction [Girard 1989] are worth being investigated, especially in view of some recent developments [Dal Lago et al. 2017].

## ACKNOWLEDGMENTS

This work was partially supported by ANR grants Elica (ANR-14-CE25-0005) and Repas (ANR-16-CE25-0011), and INRIA/JSPS EA Crecogi.

## REFERENCES

Beniamino Accattoli, Stéphane Graham-Lengrand, and Delia Kesner. 2018. Tight typings and split bounds. *PACMPL* 2, ICFP (2018), 94:1–94:30.

J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, P. Naur, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, M. Woodger, and W. L. van der Poel. 1962. Revised report on the algorithmic languageAlgol 60. *Numer. Math.* 1 (1962), 420–453.

Henk P. Barendregt. 1984. *The Lambda Calculus* (revised ed.). North Holland.

Flavien Breuvart and Ugo Dal Lago. 2018. On Intersection Types and Probabilistic Lambda Calculi. In *Proc. of PPDP 2018, Frankfurt am Main, Germany, September 03-05, 2018*, David Sabel and Peter Thiemann (Eds.). ACM, 8:1–8:13. https://doi.org/10.1145/3236950.3236968

Luís Caires and Frank Pfenning. 2010. Session Types as Intuitionistic Linear Propositions, See [Gastin and Laroussinie 2010], 222–236. https://doi.org/10.1007/978-3-642-15375-4_16

Giuseppe Castagna, Mariangiola Dezani-Ciancaglini, and Daniele Varacca. 2006. Encoding CDuce in the Cpi-Calculus. In *Proc. of CONCUR 2006*. 310–326.

Mario Coppo and Mariangiola Dezani-Ciancaglini. 1978. A new type assignment for $\lambda$-terms. *Arch. Math. Log.* 19, 1 (1978), 139–156.

Mario Coppo and Mariangiola Dezani-Ciancaglini. 1980. An extension of the basic functionality theory for the $\lambda$-calculus. *Notre Dame Journal of Formal Logic* 21, 4 (1980), 685–693.

Mario Coppo, Mariangiola Dezani-Ciancaglini, and Betti Venneri. 1981. Functional Characters of Solvable Terms. *Math. Log. Q.* 27, 2-6 (1981), 45–58.

Ugo Dal Lago and Marco Gaboardi. 2011. Linear Dependent Types and Relative Completeness. *Logical Methods in Computer Science* 8, 4 (2011).

Ugo Dal Lago, Ryo Tanaka, and Akira Yoshimizu. 2017. The geometry of concurrent interaction: Handling multiple ports by way of multiple tokens. In *Proc. of LICS 2017, Reykjavik, Iceland, June 20-23, 2017*. IEEE Computer Society, 1–12. https://doi.org/10.1109/LICS.2017.8005112

Daniel de Carvalho. 2018. Execution time of $\lambda$-terms via denotational semantics and intersection types. *Mathematical Structures in Computer Science* 28, 7 (2018), 1169–1203.

Marc de Visme and Damiano Mazza. 2017. On the Concurrent Meaning of Logical Correctness. (2017). Unpublished manuscript. Available on the second author's web page.

Romain Demangeon, Daniel Hirschkoff, and Davide Sangiorgi. 2010a. Termination in higher-order concurrent calculi. *J. Log. Algebr. Program.* 79, 7 (2010), 550–577.

Romain Demangeon, Daniel Hirschkoff, and Davide Sangiorgi. 2010b. Termination in Impure Concurrent Languages, See [Gastin and Laroussinie 2010], 328–342. https://doi.org/10.1007/978-3-642-15375-4_23

Yuxin Deng and Davide Sangiorgi. 2004. Towards an Algebraic Theory of Typed Mobile Processes. In *Proc. of ICALP 2004, Turku, Finland, July 12-16, 2004*. 445–456.

Thomas Ehrhard and Olivier Laurent. 2010. Interpreting a Finitary Pi-Calculus in Differential Interaction Nets. *Information and Computation* 208, 6 (2010), 606–633.

Timothy S. Freeman and Frank Pfenning. 1991. Refinement Types for ML. In *Proc. of PLDI, Toronto, Ontario, Canada, June 26-28, 1991*. 268–277.

Paul Gastin and François Laroussinie (Eds.). 2010. *CONCUR 2010 - Concurrency Theory, 21th International Conference, CONCUR 2010, Paris, France, August 31-September 3, 2010. Proceedings*. Lecture Notes in Computer Science, Vol. 6269. Springer. https://doi.org/10.1007/978-3-642-15375-4

Jean-Yves Girard. 1971. Une extension de l'interpretation de Gödel a l'analyse, et son application a l'élimination des coupures dans l'analyse et la théorie des types. In *Proc. of the Second Scandinavian Logic Symposium*. Studies in Logic and the Foundations of Mathematics, Vol. 63. Elsevier, 63 – 92.

Jean-Yves Girard. 1987. Linear Logic. *Theor. Comput. Sci.* 50, 1 (1987), 1–102.

Jean-Yves Girard. 1989. Geometry of Interaction I: interpretation of System F. In *Proc. of the Logic Colloquium '88*. North Holland, 221–260.

Jean-Yves Girard, Andre Scedrov, and Philip J. Scott. 1992. Bounded linear logic: a modular approach to polynomial-time. *Theoretical Computer Science* 97, 1 (1992), 1–66.

Kohei Honda and Olivier Laurent. 2010. An exact correspondence between a typed pi-calculus and polarised proof-nets. *Theor. Comput. Sci* 411, 22-24 (2010), 2223–2238.

Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. 1998. Language Primitives and Type Discipline for Structured Communication-Based Programming. In *Proc. of ESOP*. 122–138.

Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2008. Multiparty asynchronous session types. In *Proc. of POPL 2008, San Francisco, California, USA, January 7-12, 2008*. 273–284.

John Hughes, Lars Pareto, and Amr Sabry. 1996. Proving the Correctness of Reactive Systems Using Sized Types. In *Proc. of POPL 1996*. 410–423.

Paris C. Kanellakis, Gerd G. Hillebrand, and Harry G. Mairson. 1994. An Analysis of the Core-ML Language: Expressive Power and Type Reconstruction. In *Proc. of ICALP 1994 (Lecture Notes in Computer Science)*, Vol. 820. Springer, 83–105.

Naoki Kobayashi. 2000. Type Systems for Concurrent Processes: From Deadlock-Freedom to Livelock-Freedom, Time-Boundedness. In *Proc. of IFIP TCS 2000, Sendai, Japan, August 17-19, 2000*. 365–389.

Naoki Kobayashi. 2006. A New Type System for Deadlock-Free Processes. In *Proc. of CONCUR 2006*. 233–247.

Naoki Kobayashi, Benjamin C. Pierce, and David N. Turner. 1999. Linearity and the pi-calculus. *ACM Trans. Program. Lang. Syst.* 21, 5 (1999), 914–947.

Jean-Louis Krivine. 1993. *Lambda-calculus, types and models.* Masson.

Damiano Mazza, Luc Pellissier, and Pierre Vial. 2018. Polyadic approximations, fibrations and intersection types. *PACMPL* 2, POPL (2018), 6:1–6:28. https://doi.org/10.1145/3158094

Robin Milner. 1978. A theory of type polymorphism in programming. *J. Comput. System Sci.* 17, 3 (1978), 348 – 375. https://doi.org/10.1016/0022-0000(78)90014-4

Robin Milner. 1992. Functions as Processes. *Mathematical Structures in Computer Science* 2, 2 (1992), 119–141.

Luca Padovani. 2014. Deadlock and lock freedom in the linear $\pi$-calculus. In *Proc. of CSL-LICS '14, Vienna, Austria, July 14 - 18, 2014*. 72:1–72:10.

Mauro Piccolo. 2012. Strong Normalization in the $\pi$-calculus with Intersection and Union Types. *Fundam. Inform.* 121, 1-4 (2012), 227–252.

John C. Reynolds. 1974. Towards a Theory of Type Structure. In *Proc. of Programming Symposium, Colloque Sur La Programmation*. Springer-Verlag, 408–423.

Davide Sangiorgi and David Walker. 2001. *The Pi-Calculus - a theory of mobile processes.* Cambridge University Press.

Alex K. Simpson. 2005. Reduction in a Linear Lambda-Calculus with Applications to Operational Semantics. In *Proc. of RTA 2005, Nara, Japan, April 19-21, 2005 (Lecture Notes in Computer Science)*, Jürgen Giesl (Ed.), Vol. 3467. Springer, 219–234. https://doi.org/10.1007/978-3-540-32033-3_17

Kohei Suenaga and Naoki Kobayashi. 2007. Type-Based Analysis of Deadlock for a Concurrent Calculus with Interrupts. In *Proc. of ESOP 2007*. 490–504.

Philip Wadler. 2014. Propositions as sessions. *J. Funct. Program.* 24, 2-3 (2014), 384–418.

J. B. Wells. 1994. Typability and Type-Checking in the Second-Order lambda-Calculus are Equivalent and Undecidable. In *Proc. of LICS 1994*. IEEE Computer Society, 176–185.

Nobuko Yoshida, Martin Berger, and Kohei Honda. 2004. Strong normalisation in the pi-calculus. *Information and Computation* 191, 2 (2004), 145–202.