Causal Computational Complexity of Distributed Processes

Romain Demangeon, Nobuko Yoshida

Causal Computational Complexity of Distributed Processes

Romain Demangeon
Sorbonne Université, CNRS, LIP6

Nobuko Yoshida
Imperial College London

Abstract

This paper studies the complexity of π-calculus processes with respect to the quantity of transitions caused by an incoming message. First we propose a typing system for integrating Bellantoni and Cook's characterisation of polynomially-bound recursive functions into Deng and Sangiorgi's typing system for termination. We then define computational complexity of distributed messages based on Degano and Priami's causal semantics, which identifies the dependency between interleaved transitions. Next we apply a syntactic flow analysis to typable processes to ensure the computational bound of distributed messages. We prove that our analysis is *decidable* for a given process; *sound* in the sense that it guarantees that the total number of messages causally dependent of an input request received from the outside is bounded by a polynomial of the content of this request; and *complete* which means that each polynomial recursive function can be computed by a typable process.

Keywords computational complexity, concurrent process calculi, type system, causal dependency, flow analysis

ACM Reference Format:
Romain Demangeon and Nobuko Yoshida. 2018. Causal Computational Complexity of Distributed Processes. In *LICS '18: 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, July 9–12, 2018, Oxford, United Kingdom.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3209108.3209122

1 Introduction

Complexity of Distributed Services. A common requirement for large distributed systems, such as web applications involving a series of HTTP requests and Remote Procedure Calls, is to ensure the answer to a request arrives within a certain amount of time. Efficiency analyses for such systems separate *communication complexity*, which studies the quantity of information exchanged between the remote components of the service, from *sequential complexity*, which handles the way each component of a service is implemented in a given location. Since in most distributed services the time spent in local computations is negligible compared to the time cost of sending messages over networks, we aim in this paper to study message complexity by giving a bound on the message overhead triggered by incoming requests. Specifically we define complexity over the amount of messages *directly dependent* of an initial request, disregarding messages which are not contributing to that computation, such as communications caused by concurrent requests.

$$
\begin{aligned}
&!add(x,y,r).[x=0]\,\overline{r}\langle y\rangle + \\
&\qquad [x\neq 0]\,(\nu c)\,(\overline{add}\langle x-1,y,c\rangle \mid c(z).\overline{r}\langle z+1\rangle) \\
&\mid !mult(x,y,r).[x=0]\,\overline{r}\langle 0\rangle + [x\neq 0]\,(\nu d_1,d_2)\,(\overline{mult}\langle x-1,y,d_1\rangle \\
&\qquad \mid d_1(res).\overline{add}\langle y,res,d_2\rangle \mid d_2(z).\overline{r}\langle z\rangle) \\
&\mid !fact(x,r).[x=0]\,\overline{r}\langle 1\rangle + [x\neq 0]\,(\nu d_1,d_2)\,(\overline{fact}\langle x-1,d_1\rangle \\
&\qquad \mid d_1(res).\overline{mult}\langle x,res,d_2\rangle \mid d_2(z).\overline{r}\langle z\rangle)
\end{aligned}
$$

Figure 1. Example of Arithmetic Services.

To develop this theory, we use the π-calculus [22] and define a sound notion of complexity for processes. Previous works [3, 19, 20] have defined complexity analyses for π-calculi based on the number of reductions a process performs w.r.t. its size. Our analysis differs from theirs, using a more lax version of *causality* for the π-calculus [11] which identifies dependency links between service invocations. This introduction of causality is not a precondition for soundness: our analysis guarantees polynomial bounds for a reduction semantics. Causality is required in order to define a notion of computational complexity we can apply to open systems modeling service interactions thanks to a transition semantics.

Implicit Computational Complexity (ICC) for Processes. ICC is an area which studies the design of *a priori* complexity analyses based on type systems and syntactical characterisations [4, 5, 18, 21]: constraints guarantee that any accepted program belongs to a given complexity class. A classical ICC analysis introduced by Bellantoni and Cook [5] gives a simple syntactical characterisation of the polytime functions by dividing their parameters into two sorts (safe and unsafe) and by restricting the recursion and composition schemes to *predicative* recursion: preventing recursive usage of the result of a recursive call. We adapt this framework into one for the asynchronous π-calculus where a combination of name creation and channel passing makes establishing such an analysis challenging. When encoding services, usage of names in the service code has to be controlled in order to prevent interferences. Our analysis guarantees that, in all (finite or infinite) computations starting from a *sound* process, the set of transitions causally dependent from an external input $!f(\tilde{v})$ is finite and bounded by a polynomial in the integer components of $\tilde{v}$. To be able to state such a result for an open system, we propose a way to count transitions depending from an initial request. We introduce a relation of *service causality* on computations, which identifies causally dependent pairs of interactions.

In order to illustrate which "safe" polynomial behaviours we guarantee, we introduce an example modelling three arithmetic services in Figure 1. This process can receive requests on channel *add* and performs recursively the addition of the two integer parameters of the request. A single request $add(n,m,c)$ spawns $\Theta(n)$ transitions. Requests $mult(n,m,c)$ are also accepted, triggering a recursive computation of the multiplication $n*m$, using *add* as auxiliary service. A single request $mult(n,m,c)$ spawns $\Theta(n*m)$ transitions. Service

Acknowledgements This work has been partially sponsored by: EPSRC EP/K034413/1, EP/K011715/1, EP/L00058X/1, EP/N027833/1, EP/N028201/1, EU COST Action IC1402 ARVI, CNRS PEPS APRES, ANR project ELICA, and Laboratoire d'Informatique de Paris 6 (SU) .

LICS '18, July 9–12, 2018, Oxford, United Kingdom

ACM ISBN 978-1-4503-5583-4/18/07.
https://doi.org/10.1145/3209108.3209122

fact computes the factorial function $n!$, using *mult* an as auxiliary service. A single request $fact(n, c)$ spawns $\Theta(n!)$ transitions.

Our aim is to reject service *fact* and to guarantee that *add* and *mult* are polynomially bounded w.r.t. a causal definition of complexity (defined in § 4): if service *mult* receives two different requests $mult(2, 3, r_1)$ and $mult(10^2, 10^3, r_2)$, transitions caused by the second request do not count in the complexity bound of the first one.

Our analysis is divided into two steps. First, we propose a ***type system*** (§ 3) which checks that these services are terminating (using techniques from [15]) and enforces the predicativity of recursion [5]: it detects that a result of recursive call *res* inside service *fact* is used in another recursion in an auxiliary call to service *mult* and rules out *fact*. However, this type system alone is not enough to ensure a polynomial bound. As the second step, we introduce a ***flow analysis*** (§ 5) which guarantees that no integer received from the outside is used inside recursion: if name c were free inside service *add* instead of being restricted, the whole service would be rejected: an external input $c(10^3)$ received during a computation caused by $add(1, 3, r_3)$ would let the service give a wrong answer which could be used by another service to generate a large number of transitions, breaking expected complexity bounds. These service examples are detailed in § 5 (***A***, ***P*** and ***F*** of Figure 4).

Contributions. (i) a type system (§ 3) for the asynchronous π-calculus (§ 2); (ii) a notion of *service causality* (§ 4) inspired from [9–11] which identifies the messages dependent from service calls; (iii) a static flow analysis (§ 5) enforcing predicative recursion in replicated processes and controlling information flow; (iv) the theorems (§ 6) stating type soundness (every accepted process is polynomial) and completeness (every polytime recursive function can be computed by an accepted process); and (v) decidability results of the analyses. Related works and possible extensions are discussed in § 7. A long version[1], with proofs and further examples, and a prototype for inference[2], written in OCaml, are available.

2 The π-Calculus and Labelled Transition System

Syntax. We introduce a variant of the asynchronous π-calculus [16] used for our analysis. We consider infinite sets of *channels* $a, b, c, ...$; natural numbers $0, 1, ...$; *variables* $x, y, ...$ (for both channels and numbers); identifiers for numbers n, m and identifiers for channels (names) u, w. We also use N, M for natural numbers; and $\tilde{v}$ for a tuple $v_i, \ldots, v_k$ for some k (similarly for other sets). The syntax of our calculus is given by the following grammar:

$$\begin{array}{rcl} u, w & ::= & x, y, z, ... \mid a, b, c, ... \\ n, m & ::= & x, y, z, ... \mid 0, 1, 2, ... \\ v & ::= & n \mid u \qquad e ::= v \mid e + 1 \mid e - 1 \\ P, Q & ::= & \mathbf{0} \mid u(\tilde{x}).P \mid \overline{u}\langle\tilde{e}\rangle \mid !u(\tilde{y}).P \mid P|P \mid (\nu c)P \\ & \mid & [e = 0]P + [e \neq 0]P \end{array}$$

v describes a value which is either a number or a channel. Expressions e are either values or integer expressions built from natural numbers, integer variables and successor and predecessor operations. An ordering on integer expressions used by the type system is given by $e - 1 < e$, $e < e + 1$ and the usual ordering on $\mathbb{N}$. We use $|\tilde{v}|$ for the sum of the integer values of $\tilde{v}$. Process $\mathbf{0}$ is inactive; prefix $u(\tilde{x}).P$ is a non-replicated (linear) input on name u, receiving messages $\tilde{x}$ and guarding continuation P; prefix $\overline{u}\langle\tilde{e}\rangle$ is an output on name u sending expressions $\tilde{e}$ (and has no continuation); and prefix $!u(\tilde{y}).P$ is a replicated input on name u. When the object tuple of a prefix is empty we simply write u, $!u$ and $\overline{u}$ and we omit trailing occurrences of $\mathbf{0}$. Process $P_1|P_2$ is a parallel composition and $(\nu c)P$ is the creation of a fresh channel c whose scope is P. Limited matching is introduced together with choice; matching is only possible on integers and a branching condition is always an equality test with 0; it is equivalent to conditional "`ifzero` e `then` P `else` Q" branching structure. We sometimes write $[e \neq 0]P$ as a shortcut to $[e \neq 0]P + [e = 0]\mathbf{0}$. We say that a process has *well-formed integer expressions* if the subexpression $x - 1$ only appears in a subprocess guarded by $[x \neq 0]$. Hereafter we suppose all processes to have well-formed integer expressions.

We denote $\mathrm{fn}(P)$/$\mathrm{bn}(P)$ for the set of free/bound names in P and $\equiv_\alpha$ denotes α-conversion. $P[\tilde{v}/\tilde{x}]$ denotes substitution of variables $\tilde{x}$ by values $\tilde{v}$ in P. We write $P \in Q$ when P is a subprocess of Q.

Labelled Transition System with Paths. Transition labels contains *paths* θ as in [11] (a standard definition to identify where in processes actions are played). We define the grammar of actions $(\alpha, \beta, ...)$, paths $(\theta, \theta', ...)$ and labels $(l, l', ...)$ as follows:

$$\begin{array}{l} \alpha ::= a(\tilde{v}) \mid !a(\tilde{v}) \mid (\nu\tilde{c})\,\overline{a}\langle\tilde{v}\rangle \\ \theta ::= \epsilon \mid 0.\theta \mid 1.\theta \qquad l ::= \theta.\alpha \mid \theta.\langle\theta.\alpha, \theta.\alpha\rangle \end{array}$$

The set of names of label l, denoted by $\mathrm{n}(l)$, is the set of all names appearing in l if l is an input or an output and $\emptyset$ if l is a communication. The set of bound names of label l, denoted by $\mathrm{bn}(l)$ is the elements of $\tilde{c}$ if $l = (\nu\tilde{c})\,\overline{a}\langle\tilde{v}\rangle$ and $\emptyset$ otherwise. Actions α consist of non-replicated inputs, replicated inputs and outputs. The objects of actions (messages) carry values. An output action is written $(\nu\tilde{c})\,\overline{a}\langle\tilde{v}\rangle$ with $\tilde{c} \subseteq \tilde{v}$, which denotes restricted channels $\tilde{c}$ from the message $\tilde{v}$ are extruded. We write $\overline{a}\langle v\rangle$ when $\tilde{c}$ is empty. Paths θ are either empty; or the left side $0.\theta$ or the right side $1.\theta$ of a parallel. Labels l are either composed of a path θ leading to an action; or to a communication, consisting itself of two paths θ_1 and θ_2, leading to the matching actions. We write $\theta.\tau$ for any $\theta.\langle\theta_1.\alpha, \theta_2.\alpha\rangle$ when the matching actions are of no importance; and $\alpha \in l$ whenever $l = \theta.\alpha$ or $l = \theta.\langle\theta_1.\beta_1, \theta_2.\beta_2\rangle$ and α is β_1 or β_2.

The Labelled Transition System (LTS) is defined in Figure 2. We use $e \Downarrow v$ to denote that expression e evaluates to value v (evaluation is identity on names and integer evaluation on integer expressions). Rule (Out) describes the asynchronous output action. Parallel composition is handled by rules (Par0) and (Par1), memorising in path θ the side the action takes place. Rules (RCom) and (Com) describe (respectively *replicated* and *linear*) communications between two matching actions, at positions in the process given by paths θ_1 and θ_2. Other rules are standard.

3 Types and Typing Sytem

Types. To control potential computational explosions and infinite behaviours, we decorate the types of names used in replicated inputs with integer ***levels*** N, M, similar to the ones from [15], and we use the standard ordering of $\mathbb{N}$ to compare them. We also divide the integer expressions appearing in messages into two categories, reminiscent of the ones in [5]: nat is the type of *safe* integers; typing rules prevent recursions to be performed on them, as they can contain results of recursive calls. $\mathsf{nat}_\star$ is the type of *unsafe* integers on which recursions can be performed. We use $\circ\mathsf{nat}$ to denote integers of any kind. The syntax of channel types is given by $(N \geq 0)$:

[1] https://www-apr.lip6.fr/ demangeon/Recherche/lics18.pdf
[2] https://www-apr.lip6.fr/ demangeon/Recherche/protolics.ml

$$(\text{Alpha})\frac{P' \xrightarrow{l} Q \quad P \equiv_\alpha P'}{P \xrightarrow{l} Q} \qquad (\text{Rep})\frac{}{!a(\tilde{y}).P \xrightarrow{!a(\tilde{v})} (!a(\tilde{y}).P \mid P[\tilde{v}/\tilde{y}])} \qquad (\text{In})\frac{}{a(\tilde{x}).P \xrightarrow{a(\tilde{v})} P[\tilde{v}/\tilde{x}]} \qquad (\text{Out})\frac{\forall i, e_i \Downarrow v_i}{\overline{a}\langle \tilde{e}\rangle \xrightarrow{\overline{a}\langle \tilde{v}\rangle} \mathbf{0}}$$

$$(\text{Par0})\frac{P \xrightarrow{l} P' \quad \mathsf{bn}(l) \cap \mathsf{fn}(Q) = \emptyset}{P \mid Q \xrightarrow{0.l} P' \mid Q} \qquad (\text{Par1})\frac{P \xrightarrow{l} P' \quad \mathsf{bn}(l) \cap \mathsf{fn}(Q) = \emptyset}{Q \mid P \xrightarrow{1.l} Q \mid P'} \qquad (\text{RCom})\frac{P \xrightarrow{\theta_1.!a(\tilde{v})} P' \quad Q \xrightarrow{\theta_2.(\nu \tilde{c})\overline{a}\langle \tilde{v}\rangle} Q' \quad \forall i, c_i \notin \mathsf{fn}(P)}{P \mid Q \xrightarrow{\langle \theta_1.!a(\tilde{v}), \theta_2.(\nu \tilde{c})\overline{a}\langle \tilde{v}\rangle\rangle} (\nu \tilde{c})(P' \mid Q')}$$

$$(\text{Com})\frac{P \xrightarrow{\theta_1.a(\tilde{v})} P' \quad Q \xrightarrow{\theta_2.(\nu \tilde{c})\ \overline{a}\langle \tilde{v}\rangle} Q' \quad \forall i, c_i \notin \mathsf{fn}(P)}{P \mid Q \xrightarrow{\langle \theta_1.a(\tilde{v}), \theta_2.(\nu \tilde{c})\ \overline{a}\langle \tilde{v}\rangle\rangle} (\nu \tilde{c})\ (P' \mid Q')} \qquad (\text{Res})\frac{P \xrightarrow{l} P' \quad a \notin \mathsf{n}(l)}{(\nu a)\ P \xrightarrow{l} (\nu a)\ P'} \qquad (\text{Open})\frac{P \xrightarrow{\theta.(\nu \tilde{c})\ \overline{a}\langle \tilde{v}\rangle} P' \quad b \in \tilde{v} - \tilde{c} \quad b \neq a}{(\nu b)\ P \xrightarrow{\theta.(\nu b, \tilde{c})\ \overline{a}\langle \tilde{v}\rangle} P'}$$

$$(\text{Cho0})\frac{P \xrightarrow{l} P' \quad e \Downarrow 0}{[e = 0]P + [e \neq 0]Q \xrightarrow{l} P'} \qquad (\text{Cho1})\frac{Q \xrightarrow{l} Q' \quad e \Downarrow N \neq 0}{[e = 0]P + [e \neq 0]Q \xrightarrow{l} Q'}$$

Figure 2. Labelled Transition System.

$$T, S ::= \mathsf{nat} \mid \mathsf{nat}_\star \mid (\tilde{T})_N \mid (\tilde{T})$$

Types divide names into two sets which cannot interact with each other: (1) *Replicated names* of types $(\tilde{T})_N$ are used for the design and usage of persistent services. Reception on replicated names can only be done by replicated inputs. (2) *Linear names* of types $(\tilde{T})$ are used to carry other messages. Reception on linear names can only be done by non-replicated input.

In the following, we use terms which give insight on the computational aspect of the processes we analyse, identifying the subprocesses playing the role of computing services.

Definition 3.1 (Terminology)**.** In a process P typed with environment Γ, a *service* is a name of P given replicated type $(\tilde{T})_N$ by Γ, a *service definition of* a is a replication $!a(\tilde{x}).Q$ in P when a is a service, a *call* (resp. a *request*) is an output prefix or an output action $\overline{a}\langle \tilde{v}\rangle$ (resp. a replicated input action $!a(\tilde{v})$) on a service a.

In service definition $!a(\tilde{x}).Q$ of P: (1) a call $\overline{a}\langle \tilde{v}\rangle$ for some $\tilde{v}$ is called a *recursive call*. If $\Gamma(v_i) = \mathsf{nat}_\star$, and v_i is the integer expression $x_i - 1$, the i-th argument is *a recursion position* of service a; and (2) a call $\overline{b}\langle \tilde{v}\rangle$ for some $\tilde{v}$, with $b \neq a$ is called an *auxiliary call* and b an *auxiliary service* of a. In both cases, if $\Gamma(v_i) = (\tilde{T})$, v_i is an *answer channel*; in this case, if prefix $v_i(\tilde{z})$ appears in Q, and $\Gamma(z_j) = \circ\mathsf{nat}$, then z_j is *a result* of the recursive call.

Multiset $\mathsf{calls}(P; \Gamma)$ of calls of typable process P, w.r.t. a context Γ, is defined as follows (we use $\uplus$ for multiset union):

$$\mathsf{calls}(0; \Gamma) = \mathsf{calls}(!a(\tilde{x}).P; \Gamma) = \emptyset$$
$$\mathsf{calls}(\overline{a}\langle \tilde{v}\rangle; \Gamma) = \begin{cases} \{\overline{a}\langle \tilde{v}\rangle\} \text{ if } \Gamma(a) \neq (\tilde{T})_N \\ \emptyset \qquad \text{otherwise} \end{cases}$$
$$\mathsf{calls}(a(\tilde{x}).P; \Gamma) = \mathsf{calls}(!a(\tilde{x}).P; \Gamma) = \mathsf{calls}((\nu c)\ P; \Gamma) = \mathsf{calls}(P; \Gamma)$$
$$\mathsf{calls}(P_1 \mid P_2; \Gamma) = \mathsf{calls}(P_1; \Gamma) \uplus \mathsf{calls}(P_2; \Gamma)$$
$$\mathsf{calls}([e = 0]P_1 + [e \neq 0]P_2; \Gamma) = \mathsf{calls}(P_1; \Gamma) \uplus \mathsf{calls}(P_2; \Gamma)$$

Using this terminology gives insight about the relation between our concurrent system and sequential computation: services are functions and a composition is performed through the exchange of calls and answers. In the definition of a service, some outputs are identified as recursive calls, triggering recursive computations of the service. Other calls send information to existing services, along some answer channels, used to get back the results of the computations done by these auxiliary services.

Well-formedness of Types. We define two predicates on linear types: a type $(\tilde{T})$ is *safe* (resp. *unsafe*), whenever $\mathsf{nat}_\star \notin \tilde{T}$ (resp. $\mathsf{nat} \notin \tilde{T}$). That is, safe channels only carry safe integers (and possibly any type of channels) (and reciprocally for unsafe channels). Note that safety of types does not descend recursively inside the carried channel types, the property only describes the carried integer types.

We say that a type T is well-formed if either: (1) $T = \mathsf{nat}$ or $T = \mathsf{nat}_\star$; (2) $T = (\tilde{S})$, all $\tilde{S}$ are well-formed and T is either safe or unsafe; or (3) $T = (\tilde{S})_N$, all $\tilde{S}$ are well-formed and all linear types S_j are safe. Intuitively, a well-formed type is such that *i*) there is no mixing of integer kinds inside the carried types of a linear type and such that *ii*) linear channels passed on replicated channels are safe. Hereafter we suppose all types are well-formed. Condition *i*) enables the type system to treat reception on linear channels: names received are either all safe or all unsafe. Condition *ii*) prevents results of recursive calls to be given unsafe types: when opening the result of a call inside a computation, by default, the integers received are given safe types.

For instance $T_1 = (\mathsf{nat}_\star, \mathsf{nat}_\star, (\mathsf{nat}))_3$ is well-formed as the inside channel type is safe. The typing rules for output (in Figure 3 below) ensures that, in a recursive call $\overline{mult}\langle x - 1, y, r\rangle$ on *mult* of type T_1, channel r is a safe linear channel, and in a further reception $r(z)$, z will be given safe type nat.

Type $T_2 = (\mathsf{nat}_\star, \mathsf{nat}_\star, (\mathsf{nat}_\star))_3$ is not well-formed, as it violates condition *ii*). A recursive call $\overline{mult}\langle x - 1, y, r\rangle$ on *mult* of type T_2 is dangerous, as a further reception $r(z)$ would give type $\mathsf{nat}_\star$ to the result of the recursive call, allowing it to be used in a recursion position, and breaking the predicativity of recursion.

Typing Judgements. We denote by Γ the *typing environments*, considered as oracles, as in [12, § 2], associating all identifiers present in a process, bound and free, with a type. We write $\Gamma \vdash e : T$ for the judgement associating type T to expression e w.r.t. environment Γ. Judgement $\Gamma \vdash_N P$ states that under the typing environment Γ, process P is typable at level N. Associating typing judgements to levels allows one to control message loops arising from replications: the system ensures a process P typable at level N can only perform outputs on levels $\leq N$. Top-level (not under replication) processes can be typed with level ∞, which never appears inside types.

Output Multiset. Whenever $\Gamma \vdash_M P$ for some M, we denote by $\mathrm{out}_N(P;\Gamma)$ the multiset of all outputs of P which are given level N by Γ, reminiscent of the output set $\mathtt{os}(P)$ from rule (T – Rep) in [15]. We use $\uplus$ to denote multiset union:

$$\mathrm{out}_N(0;\Gamma) = \mathrm{out}_N(!a(\tilde{x}).P;\Gamma) = \emptyset$$
$$\mathrm{out}_N(\overline{a}\langle\tilde{v}\rangle;\Gamma) = \begin{cases} \emptyset & \text{if } \Gamma(a) \neq (\tilde{T})_N \\ \{\overline{a}\langle\tilde{v}\rangle\} & \text{otherwise} \end{cases}$$
$$\mathrm{out}_N((\nu c)\,P;\Gamma) = \mathrm{out}_N(P;\Gamma) = \mathrm{out}_N(a(\tilde{x}).P;\Gamma)$$
$$\mathrm{out}_N(P_1|P_2;\Gamma) = \mathrm{out}_N(P_1 + P_2;\Gamma) = \mathrm{out}_N(P_1;\Gamma) \uplus \mathrm{out}_N(P_2;\Gamma)$$

Operator $\mathrm{out}_N(P;\Gamma)$ recursively goes down inside the structure of a typed process and collect the multiset of all outputs at level N. It is used, as in [15], in rule (Serv) for service definitions, in order to compare the replicated input with the outputs inside the replication. For instance, if $P = !a(x).(\nu c)\,(\overline{c} \mid \overline{b} \mid \overline{a}\langle x-1\rangle \mid b.\overline{c})$ is typed with $\Gamma = a : (\mathsf{nat}_\star)_3, b : (), c : ()_1, x : \mathsf{nat}_\star$, then $\mathrm{out}_3(P;\Gamma) = \{\overline{a}\langle x-1\rangle\}$, $\mathrm{out}_1(P;\Gamma) = \{\overline{c}, \overline{c}\}$ and $\mathrm{out}_2(P;\Gamma) = \emptyset$.

Lifting and Argument Partition. In the judgements, we use two operations. The *unsafe lifting*, denoted by $[T]_\star$, casts a safe linear type into an unsafe one, a safe integer type into an unsafe one, and is the identity on other well-formed types. Unsafe lifting is used to allow "unsafe calls" (rule (UOut)) to be passed. It is defined as: (1) if $T = \mathsf{nat}$ then $[T]_\star = \mathsf{nat}_\star$; (2) if $T = (\tilde{S})$ then $[T]_\star = (\tilde{S'})$ with $S'_j = \mathsf{nat}_\star$ if $S_j = \mathsf{nat}$ and $S'_j = S_j$; otherwise $[T]_\star = T$.

If $\tilde{T} = T_1, \ldots, T_k$ is a tuple of types and $\tilde{e} = e_1, \ldots, e_k$ is a tuple of expressions of the same length, we define $(\tilde{e} \triangleleft \tilde{T}) = (\tilde{e}_r; \tilde{e}_s)$ as the *partition* of the integer expressions of $\tilde{e}$ into *unsafe* arguments e_i s.t. $T_i = \mathsf{nat}_\star$ and *safe* arguments e_j s.t. $T_j = \mathsf{nat}$. Comparisons between tuples of integer expressions use the product composition of the ordering given in § 2: $\tilde{e} < \tilde{e}'$ whenever for all i, $e_i \leq e'_i$ and there exists one j s.t. $e_j < e'_j$. Comparison between separated arguments is done with $(\tilde{e}^1_r; \tilde{e}^1_s) < (\tilde{e}^2_r; \tilde{e}^2_s)$ whenever $\tilde{e}^1_r < \tilde{e}^2_r$ and $\tilde{e}^1_s = \tilde{e}^2_s$, that is when the unsafe arguments are strictly smaller and the safe arguments are equal.

These comparisons are used in the type system to identify decreasing among the arguments. Suppose we have a replication $!a(x,y,z,r).P$ and a context Γ s.t. $\Gamma(a) = (\mathsf{nat}_\star, \mathsf{nat}, \mathsf{nat}_\star, (\mathsf{nat}))_3$. And suppose that in P we have $\overline{a}\langle x-1, y, z, d\rangle \in P$. We have, $(x,y,z,r \triangleleft \tilde{T}) = (x,z;y)$ and $(x-1,y,z,d \triangleleft \tilde{T}) = (x-1,z;y)$. When comparing the content of messages, we can assert a "decreasing" exists by stating that $(x,z;y) < (x-1,z;y)$.

Typing Rules The typing of values is defined in the first line of Figure 3. Subtyping for integers is treated by the last rule which states that any unsafe integer value can be given a safe integer type (our control of predicativity of recursion relies on the fact that the opposite is not sound).

The typing system for processes is given in Figure 3. (Nil) states that $\mathbf{0}$ is typable at any level. Then rules (In, Out, Res, Cond, Res) are standard, with (In, Out) only applying to linear prefixes.

Typing for non-linear outputs is divided into two rules, following the way these outputs are used inside a service definition: we distinguish between *i*) recursive calls and safe calls to auxiliary services and *ii*) unsafe auxiliary calls. Rule (SOut) types, at level N, a *safe call*: an output inside a replication s.t. the channels passed in the messages are all safe. An output is performed either on a name of level $M < N$ (an auxiliary call) or on a name of level N (a recursive call). Well-formedness of types forces the passed channels to be safe (so the result will not be use inside recursion). Rule (UOut) types, at level N, an *unsafe auxiliary call* on a name of level M strictly lower than N. Every passed channel has to be unsafe, and every passed integer has to be unsafe (no recursion result is used and the result can be used in recursion again): this condition is enforced thanks to the lifting operator $[]_\star$.

The crux of our system is rule (Serv) which types replicated inputs, seen as services. It checks that continuation P is typable at the level N of name u: this ensures that no output on level strictly greater than N is present in the continuation. It also checks that there is at most one output on N in the continuation, captured by $\mathrm{out}_N(P;\Gamma)$ and that this output is sent with strictly smaller unsafe arguments and identical safe arguments (condition with $(\tilde{e} \triangleleft \tilde{T}) < (\tilde{y} \triangleleft \tilde{T})$). Replicated inputs are always typed as level ∞ (Example 4.1 explains the reason).

For instance, consider the example above $P = !a(x).(\nu c)\,(\overline{c} \mid \overline{b} \mid \overline{a}\langle x-1\rangle \mid b.\overline{c})$ with $\Gamma = a : (\mathsf{nat}_\star)_3, b : (), c : ()_1, x : \mathsf{nat}_\star$. We obtain directly premises $\Gamma \vdash u : (\mathsf{nat}_\star)_3$ and $\Gamma \vdash x : \mathsf{nat}_\star$. All replicated outputs inside the continuation have levels smaller than 3, which allows it to be typed at level 3 and we obtain premise $\Gamma \vdash_3 (\nu c)\,(\overline{c} \mid \overline{b} \mid \overline{a}\langle x-1\rangle \mid b.\overline{c})$. We have computed above $\mathrm{out}_P(\Gamma;3) = \{\overline{a}\langle x-1\rangle\}$, hence we need to check $\Gamma(a) = \Gamma(a)$ and $(x-1 \triangleleft \mathsf{nat}_\star) < (x \triangleleft \mathsf{nat}_\star)$, which holds by definition of $<$ on integer expressions. As a result we deduce $\Gamma \vdash_\infty !a(x).(\nu c)\,(\overline{c} \mid \overline{b} \mid \overline{a}\langle x-1\rangle \mid b.\overline{c})$; any process containing P will be typed at level ∞, preventing it to occur inside another replication, as replicated names have finite levels.

Example 3.2. We explain here how the type system validates predicativity of recursion on distributed services using the examples of Figure 4.

(1) Simple recursive service. Process $\boldsymbol{A}$ performs the addition of the integer values received on *add*. To type it, we use the following environment Γ_1:

$\Gamma_1(add)$	=	$(\mathsf{nat}_\star, \mathsf{nat}, (\mathsf{nat}))_1$	$\Gamma_1(x)$	=	$\mathsf{nat}_\star$
$\Gamma_1(y)$	=	nat	$\Gamma_1(r)$	=	(nat)
$\Gamma_1(c)$	=	(nat)	$\Gamma_1(z)$	=	nat

The typing derivation is presented in Figure 5. The main process is typed at level ∞, as it contains a replication. As *add* is of level 1, the continuation of the replication has to be typed at this level. Premises ensure there is at most one output on level 1: in this case, it is a recursive call and the side conditions ensure there is a strict decreasing on unsafe integer argument $x - 1 < x$ (and that safe argument y is untouched). Rule (Cond) allows to type the two sides of the $+$. On the left-hand side, we type the output on r as a linear output, and on the right-hand side, we type the recursive call with (Serv), the input on c and the final output with (In) and (Out), checking that arguments have appropriate types.

(2) Recursive service using an auxiliary service. Process $\boldsymbol{P}$ performs the multiplication of its first two parameters via the use of the addition performed by $\boldsymbol{A}$. To type $\boldsymbol{P}$, we α-convert its subprocess $\boldsymbol{A}$ (because of bound name collisions) and define Γ_2 as the α conversion of Γ_1 together with:

$\Gamma_2(r)$	=	(nat)	$\Gamma_2(mult)$	=	$(\mathsf{nat}_\star, \mathsf{nat}_\star, (\mathsf{nat}))_2$
$\Gamma_2(x)$	=	$\mathsf{nat}_\star$	$\Gamma_2(y)$	=	$\mathsf{nat}_\star$
$\Gamma_2(d_1)$	=	(nat)	$\Gamma_2(d_2)$	=	(nat)
$\Gamma_2(res)$	=	nat	$\Gamma_2(z)$	=	nat

The *mult* service definition is typed with rule (Serv), checking that there is only one call at level 2, and that it is done on strictly smaller

Value Typing

$$\frac{\Gamma(v) = T}{\Gamma \vdash v : T} \quad \frac{e \in \mathbb{N}}{\Gamma \vdash e : \circ\mathsf{nat}} \quad \frac{\Gamma \vdash e : \circ\mathsf{nat}}{\Gamma \vdash e - 1 : \circ\mathsf{nat}} \quad \frac{\Gamma \vdash e : \circ\mathsf{nat}}{\Gamma \vdash e + 1 : \circ\mathsf{nat}} \quad \frac{\Gamma \vdash e : \mathsf{nat}_\star}{\Gamma \vdash e : \mathsf{nat}}$$

Process Typing

$$(\text{Nil})\frac{}{\Gamma \vdash_N \mathbf{0}} \quad (\text{In})\frac{\Gamma \vdash_N P \quad \Gamma \vdash u : (\tilde{T}) \quad \Gamma \vdash \tilde{x} : \tilde{T}}{\Gamma \vdash_N u(\tilde{x}).P} \quad (\text{Out})\frac{\Gamma \vdash u : (\tilde{T}) \quad \Gamma \vdash \tilde{e} : \tilde{T}}{\Gamma \vdash_N \overline{u}\langle\tilde{e}\rangle} \quad (\text{Par})\frac{\Gamma \vdash_N P_i \ \ (i = 1, 2)}{\Gamma \vdash_N P_1 \mid P_2} \quad (\text{Res})\frac{\Gamma \vdash_N P}{\Gamma \vdash_N (\nu c)\, P}$$

$$(\text{Cond})\frac{\Gamma \vdash_N P_i \ \ (i = 1, 2) \quad \Gamma \vdash e : \circ\mathsf{nat}}{\Gamma \vdash_N [e = 0]P_1 + [e \neq 0]P_2} \quad (\text{SOut})\frac{\Gamma \vdash u : (\tilde{T})_M \quad \Gamma \vdash \tilde{e} : \tilde{T} \quad M \leq N}{\Gamma \vdash_N \overline{u}\langle\tilde{e}\rangle} \quad (\text{UOut})\frac{\Gamma \vdash u : (\tilde{T})_M \quad \Gamma \vdash \tilde{e} : [\tilde{T}]_\star \quad M < N}{\Gamma \vdash_N \overline{u}\langle\tilde{e}\rangle}$$

$$(\text{Serv})\frac{\Gamma \vdash_N P \quad \Gamma \vdash u : (\tilde{T})_N \quad \Gamma \vdash \tilde{y} : \tilde{T} \quad (1)\ \mathrm{out}_N(P;\Gamma) = \emptyset \text{ or; } (2)\ \mathrm{out}_N(P;\Gamma) = \{\overline{b}\langle\tilde{e}\rangle\} \text{ with } \Gamma(b) = \Gamma(u) \text{ and } (\tilde{e} \triangleleft \tilde{T}) < (\tilde{y} \triangleleft \tilde{T})}{\Gamma \vdash_\infty !u(\tilde{y}).P}$$

Figure 3. Typing Rules.

$$\begin{aligned}
\boldsymbol{A} &= !add(x,y,r).[x=0]\,\overline{r}\langle y\rangle + [x\neq 0]\,(\nu c)\,(\overline{add}\langle x-1,y,c\rangle \mid c(z).\overline{r}\langle z+1\rangle)\\
\boldsymbol{P} &= \boldsymbol{A} \mid !mult(x,y,r).[x=0]\,\overline{r}\langle 0\rangle + [x\neq 0]\,(\nu d_1,d_2)\,(\overline{mult}\langle x-1,y,d_1\rangle \mid d_1(res).\overline{add}\langle y,res,d_2\rangle \mid d_2(z).\overline{r}\langle z\rangle)\\
\boldsymbol{F} &= \boldsymbol{P} \mid !fact(x,r).[x=0]\,\overline{r}\langle 1\rangle + [x\neq 0]\,(\nu d_1,d_2)\,(\overline{fact}\langle x-1,d_1\rangle \mid d_1(res).\overline{mult}\langle x,res,d_2\rangle \mid d_2(z).\overline{r}\langle z\rangle)\\
\boldsymbol{C} &= \boldsymbol{P} \mid !cube(x,r).(\nu c,d)\,(\overline{mult}\langle x,x,c\rangle \mid c(y).\overline{mult}\langle x,y,d\rangle \mid d(z).\overline{r}\langle z\rangle)\\
\boldsymbol{L} &= !a.\overline{b} \mid !b.\overline{a} \qquad \boldsymbol{E} = !a(z).[z\neq 0](\overline{a}\langle z-1\rangle \mid u(x).\overline{x}\langle z-1\rangle \mid \overline{u}\langle a\rangle)\\
\boldsymbol{P'} &= \boldsymbol{A} \mid !mult(x,y,r).[x=0]\,\overline{r}\langle 0\rangle + [x\neq 0]\,(\nu d_2)\,(\overline{mult}\langle x-1,y,d_1\rangle \mid d_1(res).\overline{add}\langle y,res,d_2\rangle \mid d_2(z).\overline{r}\langle z\rangle)\\
\boldsymbol{H} &= \boldsymbol{A} \mid !sum(f,x,r)\,[x=0]\,\overline{r}\langle 0\rangle + [x\neq 0]\,(\nu d_1,d_2,d_3)\,(\overline{sum}\langle f,x-1,d_1\rangle \mid \overline{f}\langle x,d_2\rangle \mid d_1(y_1).d_2(y_2).\overline{add}\langle y_2,y_1,d_3\rangle \mid d_3(y_3).\overline{r}\langle y_3\rangle)
\end{aligned}$$

Figure 4. Examples of Services.

$$(\text{Serv})\frac{(\text{Choice})\dfrac{(\text{Out})\dfrac{\Gamma_1 \vdash r : (\mathsf{nat})\quad \Gamma_1 \vdash y : \mathsf{nat}}{\Gamma_1 \vdash_1 \overline{r}\langle y\rangle} \quad \Gamma_1 \vdash x : \mathsf{nat}_\star\ (\text{Res})\dfrac{(\text{Par})\dfrac{(\text{SOut})\dfrac{\Gamma_1 \vdash add : (\mathsf{nat}_\star, \mathsf{nat}, (\mathsf{nat}))_1 \quad \Gamma_1 \vdash x-1,\ y,\ c : \mathsf{nat}_\star, \mathsf{nat}, (\mathsf{nat})}{\Gamma_1 \vdash_1 \overline{add}\langle x-1,\ y,\ c\rangle} \quad (\text{In})\dfrac{(\text{Out})\dfrac{\Gamma_1 \vdash r,\ z+1 : (\mathsf{nat}), \mathsf{nat}}{\Gamma_1 \vdash_1 \overline{r}\langle z+1\rangle} \quad \Gamma_1 \vdash c,\ z : (\mathsf{nat}), \mathsf{nat}}{\Gamma_1 \vdash_1 c(z).\overline{r}\langle z+1\rangle}}{\Gamma_1 \vdash_1 \overline{add}\langle x-1,\ y,\ c\rangle) \mid c(z).\overline{r}\langle z+1\rangle}}{\Gamma_1 \vdash_1 (\nu c)\,(\overline{add}\langle x-1,\ y,\ c\rangle) \mid c(z).\overline{r}\langle z+1\rangle)}}{\Gamma_1 \vdash_1 [x=0]\,\overline{r}\langle y\rangle + [x\neq 0]\,(\nu c)\,(\overline{add}\langle x-1,\ y,\ c\rangle \mid c(z).\overline{r}\langle z+1\rangle)}}{\Gamma_1 \vdash_\infty !add(x,\ y,\ r).[x=0]\,\overline{r}\langle y\rangle + [x\neq 0]\,(\nu c)\,(\overline{add}\langle x-1,\ y,\ c\rangle \mid c(z).\overline{r}\langle z+1\rangle)}$$

$$\Gamma_1 \vdash add : (\mathsf{nat}_\star, \mathsf{nat}, (\mathsf{nat}))_1 \quad \Gamma_1 \vdash x,\ y,\ r : \mathsf{nat}_\star, \mathsf{nat}, (\mathsf{nat}) \quad \mathrm{out}_1(\ldots;\Gamma_1) = \{\overline{add}\langle x-1,\ y,\ c\rangle\} \wedge (x-1|y) < (x|y)$$

Figure 5. Typing Derivation for $\boldsymbol{A}$.

arguments: $(x-1, y;) < (x, y;)$. The continuation is typed at level 2 (the level of *mult*). The auxiliary call to *add*, of level 1, is typable using rule (SOut) (as the call is passed to a service at a strictly lower level). The remaining of the typing derivation is similar to the one of $\boldsymbol{A}$. The interesting point is that y has type $\mathsf{nat}_\star$ as it is used in a recursion position in the auxiliary call $\overline{add}\langle y, res, d_2\rangle$. Rule (SOut) and well-formedness of types force d_1, sent on a recursive call, to be a safe channel, thus *res* has type nat. Hence a call $\overline{add}\langle res, y, d_2\rangle$ (which would yield the same result, as the operation is commutative) is untypable: in this case, the recursion in *add* would be done on the result of the recursive call of *mult*, which is unpredicative recursion.

(3) Exponential service. Process $\boldsymbol{F}$ computes the factorial function and is not typable: for the same reason as the one invoked above, d_1 has to be a safe channel and *res* has to be of type nat. As a consequence, *res* cannot be used as any argument in output $\overline{mult}\langle x, res, d_2\rangle$ (or $\overline{mult}\langle res, x, d_2\rangle$) which requires two arguments typed by $\mathsf{nat}_\star$. Hence we reject $\boldsymbol{F}$.

(4) Unsafe calls in a polynomial service. Process $\boldsymbol{C}$ computes the cubic exponent of its argument. It is typable with a context containing $\{cube : (\mathsf{nat}_\star, (\mathsf{nat}))_3\}$. The service itself is not recursive but uses twice the channel *mult* to compute multiplications. Rule (Res), when typing the continuation at level 3, gives unsafe linear types to c and d, used in an unsafe auxiliary call to *mult*, typed by rule (UOut), thanks to level comparison $3 > 2$. Our system gives to x type $\mathsf{nat}_\star$ and to d type $(\mathsf{nat}_\star)$, which implies that y's type is $\mathsf{nat}_\star$, allowing to apply the rule (UOut). Note that d can be typed either (nat) or $(\mathsf{nat}_\star)$ leading to the use of either rule (UOut) or (SOut) for the second call to *mult*. In the latter case, a cast from $\mathsf{nat}_\star$ to nat is applied when typing $\overline{r}\langle z\rangle$.

(5) Diverging behaviour Process $\boldsymbol{L}$ describes a diverging behaviour between two services a and b. When trying to typecheck it, these names have to be given recursive channel types. It is not typable with any environment $\Gamma = \{a : ()_{N_1}, b : ()_{N_2}\}$ because the two applications of rule (Serv) are forcing both $N_1 > N_2$ and $N_2 > N_1$. Our type system enforces termination as in [15].

(6) Multiple recursive calls. Process $\boldsymbol{E}$ performs an exponential number of transitions on a since each call $\overline{a}\langle N\rangle$ spawns two recursive calls on $\overline{a}\langle N-1\rangle$. One call is directly visible and the other one is hidden under an interaction on u. Our type system rejects this process: if $\Gamma(a) = (\mathsf{nat}_\star)_N$ for some l, then the output $\overline{u}\langle a\rangle$ is typable only if $\Gamma(u) = ((\mathsf{nat}_\star)_N)$, and the input $u(x)$ forces x to be given same type $(\mathsf{nat}_\star)_N$. When typing the replicated input, predicates in rule (Serv) require that $\mathrm{out}_N(P;\Gamma)$ is a singleton or the empty set; as it is a pair $(\overline{a}\langle z-1\rangle, \overline{x}\langle z-1\rangle)$ here, we reject $\boldsymbol{E}$.

(7) Uncontrolled expression. Process $\boldsymbol{P'}$ is a copy of $\boldsymbol{P}$ except name d_1 is not private. It is typable, using a typing derivation close to the one for $\boldsymbol{P}$ (minus the (Res) rule, as there is one less restriction). Yet, the information flow analysis introduced later in § 5 rejects this process.

(8) Higher-order service. Process $\boldsymbol{H}$ offers a *higher-order* service on channel *sum* accepting requests containing name f and integer x. If service f computes function $\mathcal{F} : \mathbb{N} \to \mathbb{N}$, then $!sum(f, N, r)$ eventually produces output $\overline{r}\langle\sum_{1\leq k\leq N}\mathcal{F}(k)\rangle$. Process $\boldsymbol{H}$ is typable by our typing system.

Limitation of the Typing System. Our type system enforces termination and predicativity of recursion: the result obtained from a recursive call is never used in a recursion position, in the spirit of [5]. Therefore, one would expect that our system guarantees polynomial bounds for services, just as [5] characterises polytime functions. The following process $\boldsymbol{CE}$ is a counterexample:

$$
\begin{aligned}
&incr \;=\; d(z).\overline{d}\langle z+1\rangle\\
&\boldsymbol{CE} \;=\; !add(x,y,r).[x=0]\overline{r}\langle y\rangle + [x\neq 0](\nu c)\,(\overline{add}\langle x-1,y,c\rangle\\
&\qquad\qquad \mid\; c(z).\overline{r}\langle z+1\rangle \;\mid\; incr)\\
&\mid !mult(x,y,r).[x=0]\overline{r}\langle 0\rangle + [x\neq 0](\nu c1,c2)\,(\overline{mult}\langle x-1,y,c1\rangle\\
&\qquad\qquad \mid\; c1(z1).\overline{add}\langle y,z1,c2\rangle \;\mid\; c2(z2).\overline{r}\langle z2\rangle \;\mid\; incr)\\
&\mid !fact(x).[x=0]\overline{r}\langle 1\rangle + [x\neq 0](\nu c)\,(\overline{fact}\langle x-1\rangle\\
&\qquad\qquad \mid\; d(z).(\overline{mult}\langle z,x-1,c\rangle \;\mid\; \overline{d}\langle z+1\rangle \;\mid\; \overline{d}\langle 1\rangle))
\end{aligned}
$$

This process is similar to the one of Figure 1: the main difference is that addition and multiplication services include an incrementer module *incr* which receives value z on channel d and immediately sends $z+1$ on d. The factorial service from $\boldsymbol{CE}$ is different from the factorial service of Figure 1: it calls itself recursively, but instead of obtaining the result of the recursive call on a private answer channel sent along the call (which would be detected by the type system), it listens on free channel d and uses the value obtained to carry on computation. We can prove by recurrence that a single output $fact(N)$ can produce N recursive calls to *fact*, spawn $N!$ copies of *incr* and thus generate more than $N!$ reductions. Yet, $\boldsymbol{CE}$ is typable by the typing system in Figure 3 and predicativity of recursion is not violated: the process is not using the result of a recursive call in a recursion position. Here integer z received on d can be given an unsafe type (and d type $(\mathsf{nat}_\star)$), as it is not linked to the recursive call $\overline{fact}\langle x-1\rangle$.

In $\boldsymbol{CE}$, the message-passing power of the π-calculus is interfering with the type system: free name d is used to transfer information from different independent computations of services *mult* and *add* and to carry it to a *fact* computation. Indeed, it is not enough to actually enforce constraints on recursive calls; one needs to control the information flow between computations in order to ensure the origin of the values received inside computations is known, and to prevent usage of uncontrolled information. In order to achieve this goal, we introduce an information flow analysis in § 5 which

$$
\begin{aligned}
\boldsymbol{S}_0 &= !a(x).[x\neq 0]\overline{a}\langle x-1\rangle\\
\boldsymbol{S}_1 &= !a(x).([x\neq 0]\overline{a}\langle x-1\rangle + [x=0]\overline{c})\\
\boldsymbol{S}_2 &= c.!b(x).([x\neq 0]\overline{b}\langle x-1\rangle)\\
\boldsymbol{S}_3 &= !a(x).[x\neq 0]\;(c.\overline{a}\langle x-1\rangle \;\mid\; \overline{d})\\
\boldsymbol{S}_4 &= !b(x).[x\neq 0]\;(d.\overline{b}\langle x-1\rangle \;\mid\; \overline{c})\\
\boldsymbol{S}_5 &= !a(x).[x\neq 0]\;(c_2.\overline{a}\langle x-1\rangle \;\mid\; \overline{c_1}\langle d_2\rangle)\\
\boldsymbol{S}_6 &= !b(x).[x\neq 0]\;(d_2.\overline{a}\langle x-1\rangle \;\mid\; \overline{d_1}\langle c_2\rangle)\\
\boldsymbol{S}_7 &= !a(x).!b(y).[y\neq 0].\overline{b}\langle y-1\rangle
\end{aligned}
$$

Figure 6. Examples Guiding the Causality Definition.

supplements the type system and guarantees a polynomial bound on the number of reductions. In addition, it allows us to state a complexity result for *open systems*. To this end, we define a notion of service causality in § 4.

4 Causal Dependency

Instead of counting the number of reductions or the size of the evolving processes, our objective is to control the number of transitions *caused* by a request to a service. For this, we define a causality relation which is able to remember, for each action, the previous transitions which make it happened. The frameworks developed in [9–11] propose two kinds of causal dependencies in a computation (a sequence of transitions): *structural* dependency relates a transition to a previous transition which prefixed (guarded) it; and *binding* dependency [11] relates a transition to a previous output transition that extrudes one of its names. Since our analysis focuses on services (implemented through replications) [9], and we need to cut undesirable causality links.

Example 4.1. **(1) Independence of replications.** In [9], subsequent firings of the same replicated input are causally related. Consider an invocation $!a(10)$ to $\boldsymbol{S}_0$ of Figure 6 which spawns 10 messages. A subsequent independent invocation $0.!a(100)$ will produce another chain of 100 transitions that should be considered unrelated to the previous one, from a service usage perspective. Yet, first rule of Definition 2 in [9] makes $0.!a(100)$ causally dependent from $a(10)$.

(2) Guarded replications. In $(\boldsymbol{S}_1 \mid \boldsymbol{S}_2)$, the services proposed on a and b can be considered polynomial, as the number of transitions caused by initial request $0.!a(N)$ or $1.!b(N)$ is linear w.r.t. N. However if we build a definition of causal complexity based on [9], service b is of linear complexity but a is not: an input $0.!a(N)$ eventually produces an output $\overline{c}$, which is able to react with the guard c of $\boldsymbol{S}_2$ and makes b available. All further calls to b will be causally related to the initial request $!a(N)$ through this synchronisation on c, preventing service a to be bound. Messages fired from usages of b are *indirectly* related to the first request on a: they require additional inputs $b(M)$ to happen.

(3) Independent requests. The causality in [9] relates chains of transitions produced by independent requests. In $(\boldsymbol{S}_3 \mid \boldsymbol{S}_4)$, the two processes are blocking each other recursion, but overall behaviour of a and b can be considered linear. There exists an infinite computation from $(\boldsymbol{S}_3 \mid \boldsymbol{S}_4)$ containing an infinite sequence of external inputs $0.!a(5)$, $1.!b(10)$, $\theta_2.!a(10)$, $\theta_3.!b(10)$, ... In this computation, according to causality in [10], the transitions depending of request $0.!a(5)$ are interleaved with those depending of request $1.!b(10)$, linear communications on d and c unlocking further transitions. As a result, all requests of the sequence produce transitions related to the initial one and the set of transitions causally dependent from the first

request is infinite.

(4) Binding. The causality in [11] relates transitions through binding. Consider $\boldsymbol{S}_5$ and $\boldsymbol{S}_6$ which are variants of $\boldsymbol{S}_3$ and $\boldsymbol{S}_4$. In $(\nu c_2, d_2)(\boldsymbol{S}_5 \mid \boldsymbol{S}_6)$, an external output $(\nu d_2)\overline{c_1}\langle d_2\rangle$ can extrude name d_2, allowing an external linear input d_2 to be performed later. Causality in [11] includes binding causality and relates both transitions, linking interleaved computations performed on a and b, as in the example in **(3)**.

(5) Nested replications. The causality in [9] relates transitions from nested replications. Process $\boldsymbol{S}_7$ receives requests on a, but does not do anything except freeing new replications on b implementing a linear service. There is no guarantee on the number of transitions dependent from an external input $!a(10)$, because all usages of the freed replications on b are causally related to this input. Our type system introduced in § 3 prohibits nested replications.

In summary, if we do not propagate causality through linear communications and channel binding, and if we do not link different requests to the same replicated input, we obtain a causality relation relevant for our analysis: we can compute, for each transition, to which previous external request $\theta.!a(\tilde{v})$ is related (as formalised in Theorem 4.4). Informally, the definitions in [9–11] represent upward causality (l_i causes l_j when l_i is necessary for l_j to happen) whereas our causality goes downward (l_i causes l_j when l_j is a consequence of l_i alone).

Service Causality. Service causality defines the causality links between transitions of the same computation (Definition 4.2). As explained above, it weakens the structural causality of [9] and ignores binding causality of [11] to define a dependency relation ($\subseteq_{\mathsf{d}}$ below): we do not relate two different requests to the same replicated input; and ignore messages caused by external outputs and causality links from linear transition.

Definition 4.2 (Service Causality)**.**

1. A *causality relation* between labels ($l \subseteq l'$) is defined by the following rules:
   (1) $a(\tilde{v}) \subseteq_{\mathsf{d}} l$ (r1) $!a(\tilde{v}) \subseteq_{\mathsf{d}} 1.l$
   (2) $i.l \subseteq_{\mathsf{d}} i.l'$ if $l \subseteq_{\mathsf{d}} l'$
   (3) $\langle l_0, l_1\rangle \subseteq_{\mathsf{d}} \langle l_0', l_1'\rangle$ if $l_i \subseteq_{\mathsf{d}} l_j'$ for some i, j and $!a(\tilde{v}) \in \langle l_0, l_1\rangle$
   (4) $\langle l_0, l_1\rangle \subseteq_{\mathsf{d}} l'$ if $l_i \subseteq_{\mathsf{d}} l'$ for some i and $!a(\tilde{v}) \in \langle l_0, l_1\rangle$
   (5) $l \subseteq_{\mathsf{d}} \langle l_0', l_1'\rangle$ if $l \subseteq_{\mathsf{d}} l_j'$ for some j
2. A *computation* C from process P_0 is a finite or infinite sequence of transitions and processes $(l_k, P_k)_{k\in I\subseteq\mathbb{N}^*}$ s.t. for all i, $P_i \xrightarrow{l_i} P_{i+1}$. Transitions in a computation are uniquely identified by their labels.
3. Let C be a computation and $i, j \in \{0, .., n\}$ with $i < j$. We say that transition l_i *depends on* l_j in C, written by $l_i \subseteq_C l_j$ iff $l_i \subseteq_{\mathsf{d}} l_j$. We call the reflexive and transitive closure of $\subseteq_C$ *causal dependency* and write it $\subseteq$.

Rule (1) states transitions fired from the continuation of a standard input depend on this input, and rule (r1) defines transitions fired from a spawned replicated process depend on the input that created a copy of the replicated process. Different triggers of the same replicated inputs are not related. Rule (2) navigates through parallel compositions when computing dependencies. Rules (3) and (4) state that a *replicated* communication is responsible for the transitions depending of its matching actions (but linear communications do not propagate causality). Rule (5) relates an external input (as our calculus is asynchronous l cannot be an output) to any communications involving a prefix it guarded. The service causality does *not* include (i) relations where the left-hand side is an external output (since our calculus is asynchronous); and (ii) relations where a linear communication appears in the left-hand side.

Example 4.3. $\boldsymbol{S} = !a(x, y).[x \neq 0](\nu d)\ (\overline{a}\langle x-1, y\rangle \mid \overline{d} \mid d.\overline{b} \mid b.\overline{y})$ From $\boldsymbol{S}$, input $!a(10, r)$ can produce, in total, 10 recursive calls on a, 10 synchronisations on d, 10 outputs on r and 10 inputs/outputs/communications on b. If this initial input is followed by two other inputs $\theta.!a(10, r_1)$ and $\theta'.!a(5, r_2)$, we can distinguish, in the subsequent τ transitions on names d, between the ones caused by the first input and the ones caused by another input. Even if communications on name b can happen between two of these interleaved computations, their interferences are not taken into account when propagating causality by rules (3, 4) in Definition 4.2. Each replicated input $\theta.!a(N, r)$ will cause a number of transitions linear in N.

Theorem 4.4 characterises the effect of the transformations we apply to the causality definition from [9]: we ensure that, in a computation from a process without nested replications, any transition which is not a local communication is caused by at most one external replicated input. Thus, when considering the usage of a service we can identify the particular request that causes it (if it exists). Local communications are excluded because each side of a communication can be caused by a different replicated input, even if causality is not propagated further.

Theorem 4.4 (Unicity of Cause)**.** *Let P be a process which does not contain nested replications, S a computation from P and l_{i_1}, l_{i_2}, l_j three transitions of S s.t. $l_{i_1} \subseteq l_j$ and $l_{i_2} \subseteq l_j$. Then if* (a) *$l_{i_1} = \theta_1.!a_1(\tilde{v}_1)$ for some a_1, $\tilde{v}_1$, and* (b) *$l_{i_2} = \theta_2.!a_2(\tilde{v}_2)$ for some a_2, $\tilde{v}_2$, and* (c) *l_j is not a linear communication, then we have $i_1 = i_2$.*

5 Information Flow Analysis

This section introduces a set of constraints which guarantee polynomial bounds for typed processes which abide to them. In order to rule out process $\boldsymbol{CE}$ from § 3, our analysis needs to detect that some information is passed through different recursive calls, using channel d. Indeed, the main culprit in the case of $\boldsymbol{CE}$ not being polynomial is integer z received on d, which is able to act as if it was the result of recursive call $\overline{fact}\langle x-1\rangle$. As our type system is unable to identify z as a safe integer – as there is nothing which would force d to be a safe channel (such as d being carried on a replicated channel), it does not prevent its usage in a recursion position in $\overline{mult}\langle z, x-1, c\rangle$.

The goal of our information flow analysis is to identify the integers used in critical positions (arguments of recursive or auxiliary calls inside service definitions and results of computation sent on answer channels) and to check that their origin is controlled, i.e. integers have been received in a reliable way.

We first define *controlled expressions* as integer expressions that can be used trustfully: they are either (1) closed; or (2) they contain a single integer variable that has either be received from a request to a service; or (3) have been received on a linear name, but such linear names are local and have been extruded at most once inside calls. We denote the set of input and output prefixes of P whose subject (resp. object) is a by $\mathrm{sub}(a, P)$ (resp. $\mathrm{obj}(a, P)$).

Definition 5.1 (Controlled)**.** Let P be a typable process, Q a subprocess of P, and e an integer expression occurring in Q. We say that e is *controlled in Q of P*, denoted by $\mathrm{ok}(e, Q \in P)$, whenever either:

1. e does not contain any variable;

$\boldsymbol{S} = !a(x,r).(\nu c)\ (\overline{a}\langle x-1, c\rangle \mid c(z).\overline{r}\langle z\rangle \mid d(y).\overline{s}\langle y\rangle \mid \overline{s}\langle 8\rangle)$
$\boldsymbol{U_1} = !a(x,r).(\nu c_1, c_2)\ (\overline{a}\langle x-1, c_1\rangle \mid c_1(z).\overline{r}\langle z\rangle \mid d(y).\overline{b}\langle y, c_2\rangle)$
$\boldsymbol{U_2} = !a(x,r).(\nu c)\ (\overline{a}\langle x-1, c\rangle \mid c(z).\overline{r}\langle z\rangle \mid \overline{s}\langle r\rangle)$
$\boldsymbol{U_3} = !a(x,r).(\nu c)\ (\overline{a}\langle x-1, c\rangle \mid c(z).\overline{r}\langle z\rangle \mid d(y).\overline{r}\langle y\rangle)$

Figure 7. Examples of sound and unsound processes.

2. e contains variable x_i, bound in $!a(\tilde{x}).R \in Q$; or
3. e contains variable y_i, bound in $b(\tilde{y}).R \in Q$ with: (a) b is bound by restriction $(\nu b)\ R' \in Q$; (b) $\mathrm{sub}(b, R') = \{b(\tilde{y})\}$; and (c) $\mathrm{obj}(b, R') \subseteq \{\overline{d}\langle \ldots, b, \ldots\rangle\}$

We recall that integer expressions contain at most one variable. Definition 5.2 ensures that there exists a flow of controlled integers inside services: (1) all integers passed inside auxiliary and recursive calls are controlled; (2) all channels received with the initial call to the service ("answer channels") are (a) not passed as arguments and (b) outputs on them only contain controlled integers.
Consider process $\boldsymbol{S}$ from Figure 7 where x, z, y are integers, a is a replicated channel and r, c, d are linear channels. Integer expression $x-1$ is controlled, because it contains variable x which is bound in the replicated input (2). Expression z is controlled because it is bound by input $c(z)$ (3); c is bound by restriction (νc) inside the replication (3.a); there is only one prefix in which c appears as an subject of an input $c(z)$ (3.b); and there is only one prefix where c appears as an object, which is an output $\overline{a}\langle x-1, c\rangle$ (3.c). Expression y is not controlled because it is received from $d(y)$ (3) but d is not bound inside the replication continuation (3.1). Expression 8 is controlled, as it contains no variable.

Definition 5.2 (Sound Process)**.** Let P be a process s.t. $\Gamma \vdash_N P$ for some Γ, N. We write $\mathrm{sound}(P)$ whenever, for each subprocess $!a(\tilde{x}).Q \in P$:

1. in all $\overline{c}\langle\tilde{e}\rangle \in \mathrm{calls}(Q;\Gamma)$, for all e_i, if $\Gamma \vdash e_i : \circ\mathsf{nat}$, then $\mathrm{ok}(e_i, !a(\tilde{x}).Q \in P)$.
2. for all $z \in \tilde{x}$, (a) $\mathrm{obj}(z, Q) = \emptyset$; and (b) if $\overline{z}\langle\tilde{e}\rangle \in Q$, then for all e_i s.t. $\Gamma \vdash e_i : \circ\mathsf{nat}$, $\mathrm{ok}(e_i, !a(\tilde{x}).Q \in P)$.

Example 5.3 (Control and Soundness)**.** We can check that $\boldsymbol{S}$ is sound; first all integer expressions inside calls are controlled (1); moreover, for the channel r, received in the replicated input, there is no prefix in the continuation where r appears as an object; and in all prefixes where it appears as an output the integers expression are controlled (there is only one, $\overline{r}\langle z\rangle$ and z is controlled) (2).

Process $\boldsymbol{U_1}$ similar to $\boldsymbol{S}$ with replicated channel b and linear channels c_1, c_2 is unsound as call $\overline{b}\langle y, c_2\rangle$ carries an uncontrolled integer y, which violates (1). Process $\boldsymbol{U_2}$ is unsound as there is an output $\overline{s}\langle r\rangle$ which carries name r, received via the replicated input, which violates (2.a). Finally, process $\boldsymbol{U_3}$ is unsound as there is an output $\overline{r}\langle y\rangle$ on a name r received on the replicated input which contains an uncontrolled integer y, which violates (2.b).

We show that process $\boldsymbol{CE}$ is unsound: expression z used in the call to *mult* must be controlled in *fact* service definition, according to rule (1) of Definition 5.2. By Definition 5.1, z is either received on *fact*, which is not the case, or received on a linear channel d. Yet d is not bound by a restriction inside the service definition. Hence it does not meet Definition 5.1(3.a), which forbids a usage in a critical position for integers received on free channels free inside the service definitions.

Example 5.4 (Services from Figure 4)**.** As explained in § 3.2, processes $\boldsymbol{A}$, $\boldsymbol{P}$, $\boldsymbol{C}$ and $\boldsymbol{P'}$ are typable. Process $\boldsymbol{A}$ is sound: notice that expressions $x-1$ and y in the recursive call are controlled because they use variables bound in the replicated input. An external input of $!add(N, M, d)$ spawns $\Theta(N)$ transitions. Process $\boldsymbol{P}$ is sound. Indeed, integer expressions *res* and z are controlled because they are received on d_1 and d_2 which are created locally and abide to conditions (3.a) and (3.b) of Definition 5.1. An external input $mult(N, M, c)$ spawns $\Theta(N * M)$ transitions. Process $\boldsymbol{C}$ is sound: all integer expressions are controlled: x is bound by the replicated input and y and z are bound by c and d which abides to condition (3) of Definition 5.1. An external input $!cube(N, c)$ spawns $\Theta(N^3)$ transitions. $\boldsymbol{H}$ is sound and exhibits a polynomial behaviour, provided service f is implemented by sound process. Processes $\boldsymbol{L}$ and $\boldsymbol{E}$ are rejected by the type system. An external input $a(N)$ from process $\boldsymbol{E}$ spawns $\Theta(2^N)$ transitions.

$\boldsymbol{P'}$ is typable. Yet, consider a request $!mult(3, 3, r)$ which causes a recursive subrequest $\overline{mult}\langle 2, 3, d_1\rangle$; an input $d_1(1000)$ can happen on free name d_1 and make the computation carrying on with number 1000 and produce a final output result $\overline{r}\langle 1003\rangle$. Interestingly, $\boldsymbol{P'}$ itself is polynomial w.r.t. service causality: a request $!mult\langle N, M, r\rangle$ causes $\Theta(N \times M)$ transitions, even with the interferences. However, if we replace $\boldsymbol{P}$ by $\boldsymbol{P'}$ in $\boldsymbol{C}$, the service offered on channel *cube* is no longer polynomial: a request $!cube(3, r)$ causes a subrequest $\overline{mult}\langle 3, 3, d\rangle$ which can return result 1003 on d, because of the interference invoked above. The second subrequest $\overline{mult}\langle 3, 1000, d\rangle$ will cause more than 3000 transitions. There is no longer a bound of the number of transitions a request $!cube(3, r)$ causes, as an arbitrary large integer can be received on d_1. $\boldsymbol{P'}$ is unsound since constraints from Definitions 5.2 and 5.1 require expression *res*, received on d_1 and passed on a call to *add*, to be controlled, meaning that d_1 has to be restricted locally. $\boldsymbol{P'}$ can be seen as an open environment version of $\boldsymbol{CE}$. If arbitrary external inputs from an environment are allowed, no *incr* module is needed in order to build typable processes which are not polynomial as dangerous integers can be directly received from the outside.

6 Soundness, Completeness and Decidability

Our framework is composed of a type system, which enforces termination and predicativity of service recursion in the asynchronous π-calculus and a flow analysis, which prevents the message-passing layer from interfering with the type system. The structure of the soundness proof is reminiscent of the one of [5]. One noticeable change is that information flow constraints are used inside the proofs to guarantee the link between answers and requests, introducing additional technicalities.

The induction on the function term from [5] is replaced by an induction on levels: indeed, levels are used to stratify service usage; a service can only call auxiliary services of lower level, mimicking the tree structure of the recursive function terms. The main lemma of [5], which gives a bound on the *result* of a function call has its counterpart in Lemma 6.6, which bounds the contents of messages emitted by a service w.r.t its parameters. Flow analysis is crucial there, as it allows to relate the content of a message with either the parameters, or a result of a recursive or auxiliary call. As a single process can host many different service definitions (but only a finite number of them), the polynomial bound of a *process* is given as an upper bound of all polynomial bounds of the service definition it contains. We start from several definitions. We first restrict the valid

transitions in Definition 6.1 for two reasons: the inputs performed by the system must abide to the typing discipline; and controlled names should not be extruded.

Definition 6.1 (Typable Transition)**.** When $\Gamma \vdash_N P$, we say that $P \xrightarrow{l} P'$ is a *typable transition* whenever (1) If $a(\tilde{v}) \in l$, $\Gamma \vdash_{N'} \overline{a}\langle\tilde{v}\rangle$ for some N'; and (2) If $(\nu\tilde{c})\ \overline{a}\langle\tilde{v}\rangle \in l$, and $\tilde{c}$ is not empty, then $\Gamma(a) \neq (\tilde{T})_{N'}$.

(2) prevents the system to send calls containing restricted names to the outside. For example, if a call $\overline{a}\langle 3, r\rangle$ is sent to the environment, extruding r, any answer carrying an arbitrarily large value, for example $r(10^{10})$, would be accepted and its usage in remaining computations would breaks complexity bounds. An extrusion *through* linear outputs is permitted, though.

Lemma 6.2. 1. (Substitution Lemma) *If* $\Gamma \vdash_N P$, $\mathsf{sound}(P)$, *and* $\Gamma(\tilde{x}) = \Gamma(\tilde{v})$, *then* $\Gamma \vdash_N P[\tilde{v}/\tilde{x}]$ *and* $\mathsf{sound}(P[\tilde{v}/\tilde{x}])$.

2. (Subject Transition) *If* $\Gamma \vdash_N P$, $\mathsf{sound}(P)$ *and* $P \xrightarrow{l} P'$ *with a typable transition, then* $\Gamma' \vdash_N P'$ *for some* Γ' *and* $\mathsf{sound}(P')$.

Definition 6.3 (Typable Computation)**.** A *typable computation* (sometimes referred to as a typable computation *from* P_0) $(\Gamma_k, P_k, l_k)_{k\in I\subseteq\mathbb{N}}$ is a sequence s.t. (i) $\forall k, \Gamma_k \vdash_{N_k} P_k$ for some N_k; (ii) $\forall k, \mathsf{sound}(P_k)$; and (iii) $\forall k, P_k \xrightarrow{l_k} P_{k+1}$ is a typable transition.

In a computation, the depending set of transition l_i is the set of all transitions which depend on l_i. It allows us to define complexity bounds (Definition 6.5), as bounds on the depending sets of all external inputs in all computations from a given process.

Definition 6.4 (Depending Set)**.** Let S be a typable computation $(\Gamma_k, P_k, l_k)_{k\in I\subseteq\mathbb{N}}$. The *depending set* of l_m in S, denoted by $\mathbf{D}(l_m)_S$, is the set $\{l_k | l_m \subseteq l_k\}$.

Definition 6.5 (Complexity Bound)**.** We say that a process P is *bounded by function* $\mathcal{F} : \mathbb{N} \to \mathbb{N}$ whenever for every typable computation $S = (\Gamma_k, P_k, l_k)_{k\in I\subseteq\mathbb{N}}$ from P, for any m, if $l_m = \theta.!a(\tilde{v})$ then $|\mathbf{D}(l_m)_S| \leq \mathcal{F}(|\tilde{v}|)$. We say that P is bounded by a polynomial when there exists a polynomial $\mathcal{F}$ such that P is bounded by $\mathcal{F}$.

For instance, process $\boldsymbol{A}$ from Figure 4 is bounded by $\mathcal{F}_1 : n \mapsto 3 * n$. In any typable computation from $\boldsymbol{A}$, for each transition $\theta.a(x, y, r)$ there is at most $3 * (x + y)$ other transitions which depends from it. Similarly, $\boldsymbol{P}$ is bounded by $\mathcal{F}_2 : n \mapsto 3 * \frac{n.(n+1)}{2} + 5 * n$ and $\boldsymbol{F}$ is bounded by a function asymptotically equivalent to $n \mapsto n!$.

Lemma 6.6, a crucial prerequisite of Theorem 6.7, bounds the content of messages inside outputs directly depending of a service request by an expression composed of a polynomial of its recursive arguments and the sum of its safe arguments. It is the process counterpart to Lemma 4.1 in [5]. Case 3.(i) treats auxiliary calls and Case 3.(ii) treats answers from the service. Lemma 6.6 is used for proving Theorem 6.7 to ensure that calls to auxiliary services are performed on arguments polynomial in $|\tilde{v}|$.

Lemma 6.6 (Output Control)**.** *Assume* $\mathsf{sound}(P)$ *and* $N \in \mathbb{N}$. *Then there exists a monotone polynomial* $\mathcal{F}_N$ *such that for all typable computations* $S = (\Gamma_k, P_k, l_k)_{k\in I\subseteq\mathbb{N}}$ *from* P *and for all pairs of transitions* l_i *and* l_j *of* S *satisfying:* (1) $l_i \subseteq_{\mathsf{d}} l_j$; (2) l_i *contains* $!a(\tilde{v})$ *with* $\Gamma_i(a) = (\tilde{T})_N$ *and* $(\tilde{v} \triangleleft \tilde{T}) = (\tilde{v}_r, \tilde{v}_s)$; *and* (3) l_j *contains* $\overline{b}\langle\tilde{v}'\rangle$ *with either* (i) $\Gamma_j(b) = (\tilde{U})_{N'}$ *for some* N'; *or* (ii) $b \in \tilde{v}$, *we have* $v'_j \leq \mathcal{F}_N(|\tilde{v}_r|) + |\tilde{v}_s|$.

Theorem 6.7 (Soundness)**.** *If* $\mathsf{sound}(P)$ *then* P *is bounded by a polynomial.*

Completeness (Theorem 6.9) is stated w.r.t the set of recursive polytime functions (see [5] for instance): we can compute all recursive polytime functions with typable processes. It is proved by building a process from the polytime characterisation of [5].

Definition 6.8 (Computing Function)**.** We say that typable process P *computes function* $\mathcal{F} : \mathbb{N}^k \to \mathbb{N}$ at address a, whenever $P \xrightarrow{\theta.a(\tilde{n},c)} P'$, there is a typable computation $(\Gamma_k, l_k, P_k)_{k\leq m}$ from P' such that $l_m = \theta'.\overline{c}\langle\mathcal{F}(\tilde{n})\rangle$. We say P *computes* $\mathcal{F}$ if there exists a name a s.t. P computes function $\mathcal{F}$ at address a.

For instance, process $\boldsymbol{A}$ of Figure 4 computes $(n, m) \mapsto n + m$ on *add*. Indeed, process $\boldsymbol{A}$ is able to perform input $add(n, m, r)$ and later produce output $\overline{r}\langle n + m\rangle$. Similarly, $\boldsymbol{P}$ computes $(n, m) \mapsto n * m$ on *mult* and $\boldsymbol{F}$ computes $n \mapsto n!$ on *fact*.

Theorem 6.9 (Completeness)**.** *If* F *is a function computable in polynomial time, then there exists* P *which computes* F *s.t.* $\mathsf{sound}(P)$.

Below the size of P is defined as the number of prefixes in P. Complexity for the type inference is similar to the one in [13].

Proposition 6.10 (Deciding Typability and Soundness)**.**

1. *Deciding whether there exist* Γ, N *for a process* P *such that* $\Gamma \vdash_N P$ *can be done in time polynomial w.r.t the size of* P.
2. *Deciding if a typable process* P *is such that* $\mathsf{sound}(P)$ *is quadratic in the size of* P.

We define a constrained shape of processes which structurally enforces information flow constraints from Definition 5.2. The use of simple processes is an alternative to the information flow analysis.

Definition 6.11 (Simple Process)**.** We say typable P is *simple* whenever every replication in P has the form:

$$!a(\tilde{x}, y).[n = 0]\overline{y}\langle e(\tilde{x})\rangle + [n \neq 0]$$
$$(\nu d_1, \ldots, d_m)\ (\overline{a_1}\langle e_1(\tilde{x}), \ldots, e_{k_1}(\tilde{x}), d_1\rangle$$
$$\mid\ d_1(z_1).(\overline{a_2}\langle e_1(\tilde{x}, z_1), \ldots, e_{k_2}(\tilde{x}, z_1), d_2\rangle \mid d_2(z_2).(\ldots$$
$$\mid\ d_m(z_m)(\overline{a_m}\langle e_1(\tilde{x}, z_1, \ldots, z_m), \ldots, e_{k_m}(\tilde{x}, z_1, \ldots, z_m), d_{m+1}\rangle$$
$$\mid\ d_{m+1}(z_{m+1}).\overline{y}\langle e(\tilde{x}, \tilde{z})\rangle)))\ldots))$$

where $e(\tilde{y})$ are expressions with no variable or one variable from $\tilde{y}$.

Simple processes organise their auxiliary calls in a chain $a_1, \ldots, a_m$ and integer expressions inside calls can only refer to values received by the initial replicated inputs, or to values received by channels $d_1, \ldots, d_{m+1}$, restricted locally. Moreover, channel y is only used once, in subject of the final output. By structure, simple processes abide to Definition 5.2. Notice that processes $\boldsymbol{A}$, $\boldsymbol{P}$, $\boldsymbol{F}$ and $\boldsymbol{C}$ are equivalent to simple processes. Indeed, $\boldsymbol{A}$ is simple and $\boldsymbol{P}$ can be rewritten as simple process $\boldsymbol{P_s}$ by replacing its second branch with $d_1(res).(\overline{add}\langle y, res, d_2\rangle \mid d_2(z).\overline{r}\langle z\rangle))$. Theorem 6.12 states that the flow analysis is not needed on simple processes.

Theorem 6.12 (Simple Soundness and Completeness)**.** (1) *If* P *is simple and typable, then* $\mathsf{sound}(P)$*; and* (2) *If* F *is a function computable in polynomial time, then there exists a simple* P *which computes* F *such that* $\mathsf{sound}(P)$.

7 Related Works

Implicit Computational Complexity. ICC has been developed in many different contexts, e.g. using structural constraints [5, 21] and

type systems based on linear logics [4, 18]. Our system develops ICC into a process algebra framework inspired by one of the classical systems [5]. The latter gives a characterisation of polytime recursive function through control of predicative recursion in a recursive framework. Instead of recursive function definitions, our system controls replications in the π-calculus but has to handle interleaving of computations, mobility and hidden name passing. Because several different computations can be interleaved, we define complexity relying on causal relations extended from [11]: instead of counting the computation steps as in [5], we introduce an involved notion of service causality to identify messages which depend on a given external input. Translating the characterisation of polytime recursive functions into processes is challenging because mobility allows parametrisation of functions, arbitrary interferences on free names, interferences between different instances of function calls and diverging behaviours by fresh name-passing. Our flow analysis handles these issues by ordering channels with levels and statically checking different usages of bound names to prevent spurious exponential or infinite causal chains.

Complexity in Process Algebra. We use a *level* system inspired by [14, 15] and by *regions* from [2]. These systems decorate types with integer levels (or abstract regions), and checks that no loop arises between them (inside π-replications for [14, 15] or λ-references for [2]), ensuring termination. Ours aims to guarantee a bound on complexity on recursive functions and thus allows controlled recursive calls. Our proof technique, based on analysis of causality chains, is different from the logical relations used in [2]. The second type system presented in [15] allows recursive calls in the same region while controlling their payloads w.r.t. the initial parameters. Yet, it only considers termination and not complexity. The work by [3] studies complexity in a synchronous π-calculus, reminiscent of the Esterel model [6]: multiple processes engage during an *instant* in interleaved computations until all reach an explicit state of cooperation; then the instant is terminated, and a computation resumes in a new instant. The target applications (synchronous programs) differ from distributed services studied in this paper. In addition, their definition of complexity is different: they count the number of reductions between two instants w.r.t. the size of the whole process at the first instant. Our work counts the number of transitions caused by an external input. Our analysis relies on a type system controlling replications as opposed to a usage of annotations to control recursive π-expression in [3]. The work in [20] defines a type system based on soft linear logics [18] to control the reduction complexity of HOπ processes. Complexity is defined by the number of reductions made by a process w.r.t. its size. The calculus does not include name passing: the function-passing structure of HOπ straightforwardly allows a direct transposition of the initial system from [18]. This differs from our treatment of the π-calculus with full constructs, i.e. mobility, channel-passing and replications. The work in [19] presents a session type system which controls complexity of π-processes, defined as the number of reductions w.r.t. the initial size of the process. Because of the close correspondence between sessions and linear λ-calculi [7, 23], they are able to lightly modify the session system in [7] in order to capture polynomial behaviours. However, the expressiveness of the typable processes is heavily constrained by the linear logic based session types. Our technique of making explicit decreasings in recursive calls can be related to the use of sized types [1, 17] which use size annotation for control of recursion. In our case, the use of levels and expression comparison is more tailored for process verification.

Causality. The existing works about complexity of processes rely on counting the number of reductions a process performs w.r.t. its size [3, 19, 20]. These coarse-grained approaches make difficult any attempt at an interesting completeness result whereas our framework, which focuses on the counting of actions causally dependent from an initial request w.r.t. the content of this request, allows the definition of recursive polynomial function as input-output behaviours, bringing completeness result Theorem 6.9. We restrict Degano and Priami's definition of causality [11], defining causally dependent paths from processes via the standard LTS. This makes the definition of complexity bounds clear and straightforward and simplifies the presentation of the proofs, as transitions are compared thanks to their labels. Our analysis works with other definitions of causality, such as the ones in [8], as long as one weakens them to break the causality links from linear communications, as explained in § 4.

References

[1] Andreas Abel and Brigitte Pientka. Well-founded recursion with copatterns and sized types. *J. Funct. Program.*, 26:e2, 2016.

[2] Roberto M. Amadio. On stratified regions. In *APLAS*, volume 5904 of *LNCS*, pages 210–225. Springer, 2009.

[3] Roberto M. Amadio and Frédéric Dabrowski. Feasible reactivity in a synchronous pi-calculus. In *PPDP*, pages 221–230. ACM, 2007.

[4] Patrick Baillot and Kazushige Terui. Light types for polynomial time computation in lambda-calculus. In *LICS*, pages 266–275. IEEE, 2004.

[5] Stephen Bellantoni and Stephen A. Cook. A new recursion-theoretic characterization of the polytime functions. In *STOC*, pages 283–293. ACM, 1992.

[6] Gérard Berry and Laurent Cosserat. The ESTEREL synchronous programming language and its mathematical semantics. In *CONCUR*, volume 197 of *LNCS*, pages 389–448. Springer, 1984.

[7] Luís Caires and Frank Pfenning. Session types as intuitionistic linear propositions. In *CONCUR*, volume 6269 of *LNCS*, pages 222–236. Springer, 2010.

[8] Ioana Cristescu, Jean Krivine, and Daniele Varacca. A compositional semantics for the reversible pi-calculus. In *LICS*, pages 388–397. IEEE, 2013.

[9] Pierpaolo Degano, Fabio Gadducci, and Corrado Priami. A causal semantics for CCS via rewriting logic. *Theor. Comput. Sci.*, 275(1-2):259–282, 2002.

[10] Pierpaolo Degano, Fabio Gadducci, and Corrado Priami. Causality and replication in concurrent processes. In *PSI*, volume 2890 of *LNCS*, pages 307–318, 2003.

[11] Pierpaolo Degano and Corrado Priami. Causality for mobile processes. In *ICALP*, volume 944 of *LNCS*, pages 660–671, 1995.

[12] Romain Demangeon. *Termination of Distributed Systems*. PhD thesis, ENS Lyon / Università di Bologna, 2010.

[13] Romain Demangeon, Daniel Hirschkoff, Naoki Kobayashi, and Davide Sangiorgi. On the complexity of termination inference for processes. In *TGC*, volume 4912 of *LNCS*, pages 140–155. Springer, 2007.

[14] Romain Demangeon, Daniel Hirschkoff, and Davide Sangiorgi. Termination in impure concurrent languages. In *CONCUR*, volume 6269 of *LNCS*, pages 328–342. Springer, 2010.

[15] Yuxin Deng and Davide Sangiorgi. Ensuring termination by typability. *Inf. Comput.*, 204(7):1045–1082, 2006.

[16] Kohei Honda and Mario Tokoro. An object calculus for asynchronous communication. In *ECOOP'91*, volume 512 of *LNCS*, pages 133–147, 1991.

[17] John Hughes, Lars Pareto, and Amr Sabry. Proving the correctness of reactive systems using sized types. In *POPL*, pages 410–423, 1996.

[18] Yves Lafont. Soft linear logic and polynomial time. *Theor. Comput. Sci.*, 318(1-2):163–180, 2004.

[19] Ugo Dal Lago and Paolo Di Giamberardino. On session types and polynomial time. *MSCS*, 26(8):1433–1458, 2016.

[20] Ugo Dal Lago, Simone Martini, and Davide Sangiorgi. Light logics and higher-order processes. *MSCS*, 26(6):969–992, 2016.

[21] Daniel Leivant and Jean-Yves Marion. Lambda calculus characterizations of poly-time. *Fundam. Inform.*, 19(1/2):167–184, 1993.

[22] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I. *Inf. Comput.*, 100(1):1–40, 1992.

[23] Philip Wadler. Propositions as sessions. *JFP*, 24(2-3):384–418, 2014.