Information and Computation 290 (2023) 104998

Contents lists available at ScienceDirect

Information and Computation

journal homepage: www.elsevier.com/locate/yinco

Causal computational complexity of distributed processes

Romain Demangeon [a,*], Nobuko Yoshida [b,*]

[a] Sorbonne Université, France
[b] University of Oxford, United Kingdom

ARTICLE INFO

Article history:
Received 14 February 2021
Received in revised form 27 November 2022
Accepted 27 November 2022
Available online 1 December 2022

Keywords:
Concurrency
Process calculi
Complexity
Services

ABSTRACT

This article 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 polytime computable 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 necessary syntactic flow analysis to typable processes to ensure a computational bound on the number of distributed messages. We prove that our analysis is *decidable*; *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*, meaning that each polynomial recursive function can be computed by a typable process.

1. Introduction

Complexity of distributed services. A common requirement for large distributed systems, such as web applications involving a series of HTTP requests and/or Remote Procedure Calls, is to ensure the final answer to an initial 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.

To develop this theory, we use the π-calculus [28] and define a sound notion of complexity for processes. Previous works [3,13,12] 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 [16] which identifies dependency links between service calls. Causality is required to define a notion of computational complexity we can apply to open systems modelling service interactions thanks to a transition semantics. The introduction of causality is not a precondition for stating a soundness result: our analysis guarantees polynomial bounds for a reduction semantics.

* Corresponding authors.
E-mail address: nobuko.yoshida@cs.ox.ac.uk (N. Yoshida).

https://doi.org/10.1016/j.ic.2022.104998
0890-5401/© 2022 The Authors. Published by Elsevier Inc. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

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

Fig. 1. Example of arithmetic services.

Implicit computational complexity (ICC) for processes. ICC is an area which studies the design of *a priori* complexity analyses based on type systems (and linear logics) and syntactical characterisations [7,25,6,24]: constraints guarantee that any accepted program belongs to a given complexity class. In recent works [4,5], sized types and linear logic have been combined to obtain characterisations of polynomial-time (polytime) and exponential-time classes of functions. A classical ICC analysis [7] 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. In this article, we adapt this framework into an analysis for the asynchronous π-calculus containing unary integers (this lightly differs from the use of binary integers in Bellantoni and Cook [7]) 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.

Implicit complexity in the π-calculus. In order to illustrate which "safe" polytime behaviours we guarantee, we introduce an example modelling three arithmetic services in Fig. 1. This process is a parallel composition of three "servers" and can receive requests on channel *add* and performs recursively the addition of the two integer parameters carried by the request. A single request $add(n,m,c)$ spawns $\Theta(n)$ transitions. Requests of the form $mult(n,m,c)$ are also accepted by this process, triggering a recursive computation of the multiplication $n*m$, using *add* as an auxiliary service. A single request $mult(n,m,c)$ spawns $\Theta(n*m)$ transitions. Service *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 polytime w.r.t. a causal definition of complexity (defined in Section 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 toward the complexity of the first one.

Our analysis is divided into two steps. First, we propose a *type system* (Section 3) which checks that these services are terminating (using techniques from Deng and Sangiorgi [20]) and enforces the predicativity of recursion [7]: 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* (Section 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 Section 5 (***A***, ***P*** and ***F*** of Fig. 4).

Contributions. This article is a substantially extended version of Demangeon and Yoshida [19]: it improves the initial paper by giving complete proofs and adding a study of how the type system can be extended to high-level data structures. The materials which have not been published in Demangeon and Yoshida [19] are: the full proofs of the results (Section 6), an in depth-analysis of different counter-examples (Section 7), the discussion about completeness (Section 6) and an adaptation of the type system handling lists as values (Section 8). The related works are also extended. The structure of this article is as follows:

Section 2: a presentation of an asynchronous π-calculus, adapted to the definition of complexity bounds through the use of a labelled transition semantics with paths;

Section 3: a type system ensuring complexity bounds by restricting how the result of a recursive call can be used in further computations;

Section 4: a notion of *service causality* inspired from works on π-causality by Degano and Priami [16], Degano et al. [14, 15] which identifies the messages dependent from service calls;

Section 5: a static flow analysis controlling information flow, which complements the type system;

Section 6: the theorems stating type soundness (every accepted process is polytime) and completeness (every polytime recursive function can be computed by an accepted process), and decidability results of the analyses, together with proofs, and a discussion about completeness relative to existing complexity classes;

Section 7: a thorough exploration of different counter-examples, aimed at giving the reader intuition about how the type systems and the analysis operate;

Section 8: a detailed description of how the type system can be extended to typecheck higher-order structures such as lists; and

Section 9: related works and possible extensions.

2. The π-calculus and labelled transition system

Syntax. We introduce a variant of the asynchronous π-calculus [22] used for our analysis. We consider infinite sets of *channels* $a, b, c, \ldots$; natural numbers $0, 1, \ldots$; *variables* $x, y, \ldots$ (variables can be instantiated, in the semantics, by values, see below); 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:

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

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 $e-1 < e$ and $e < e+1$ together with 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. Our calculus is output-asynchronous: it has little incidence on the final result but makes the presentation of proofs easier. 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 $(\boldsymbol{\nu} c)P$ 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 use $\mathsf{fn}(P)$ / $\mathsf{bn}(P)$ for the set of ***free / bound names*** in P and $\equiv_\alpha$ denotes α-equivalence. $P[\tilde{v}/\tilde{x}]$ denotes substitution of variables $\tilde{x}$ by values $\tilde{v}$ in P. We write $P \subseteq Q$ when P is a subprocess of Q.

A *subprocess of* P is a process Q "which appears in P", that is, a subterm of P. We use $Q \subseteq P$ to denote that Q is a subprocess of P.

Remark. The semantics we use (inspired from Degano and Priami [16]) does not use structural congruence, as it needs to maintain "places in the process" where prefixes are fired, using *paths*. As a result, parallel composition is neither associative nor commutative.

In the following, when referring to processes in a context which does not directly involve the transition semantics, we allow us to write the process up-to commutativity, associativity and neutrality of parallel composition, e.g. to write $!a(x).\overline{r}\langle x\rangle \mid \overline{a}\langle 2\rangle \mid \overline{a}\langle 3\rangle$ instead of $\overline{a}\langle 3\rangle \mid ((0 \mid \overline{a}\langle 2\rangle) \mid !a(x).\overline{r}\langle x\rangle)$.

Labelled transition system with paths. Transition labels contain *paths* θ as in Degano and Priami [16] (a standard definition to identify where in processes actions are played). We define the grammar of ***actions*** $(\alpha, \beta, \ldots)$, ***paths*** $(\theta, \theta', \ldots)$ and ***labels*** $(l, l', \ldots)$ as follows:

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

The set of names of label l, denoted by $\mathsf{n}(l)$, 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 $\mathsf{bn}(l)$ is the elements of $\tilde{c}$ if $l = (\boldsymbol{\nu}\tilde{c})\ \overline{a}\langle\tilde{v}\rangle$ 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 $(\boldsymbol{\nu}\tilde{c})\ \overline{a}\langle\tilde{v}\rangle$ 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 composition. Labels l are composed of a path θ either 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 $\theta.\langle\theta_1.\alpha_1, \theta_2.\alpha_2\rangle$ 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.

Paths allow us to describe exactly where an action takes place. Causality, which is central to our results, is built upon paths: one of its main principles is that an action of path θ causally precedes actions of paths $\theta\theta'$. Intuitively, the path θ needs to be "cleared" in order for actions further down θ to be fired.

The Labelled Transition System (LTS) is defined in Fig. 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 (RCom0), (RCom1), (Com0) and (Com1) describe (respectively *replicated* and *linear*, with two rules each, covering the two possible respective positions for input and output) communications between two matching actions, at positions in the process given by paths θ_1 and θ_2. Other rules are standard.

$$\text{(Alpha)}\ \frac{P' \xrightarrow{l} Q \quad P \equiv_\alpha P'}{P \xrightarrow{l} Q} \qquad \text{(Rep)}\ \frac{}{!a(\tilde{y}).P \xrightarrow{!a(\tilde{v})} (!a(\tilde{y}).P \ \mid\ P[\tilde{v}/\tilde{y}])} \qquad \text{(In)}\ \frac{}{a(\tilde{x}).P \xrightarrow{a(\tilde{v})} P[\tilde{v}/\tilde{x}]}$$

$$\text{(Out)}\ \frac{\forall i, e_i \Downarrow v_i}{\overline{a}\langle\tilde{e}\rangle \xrightarrow{\overline{a}\langle\tilde{v}\rangle} \mathbf{0}} \qquad \text{(Par0)}\ \frac{P \xrightarrow{l} P' \quad \mathsf{bn}(l) \cap \mathsf{fn}(Q) = \emptyset}{P \ \mid\ Q \xrightarrow{0.l} P' \ \mid\ Q} \qquad \text{(Par1)}\ \frac{P \xrightarrow{l} P' \quad \mathsf{bn}(l) \cap \mathsf{fn}(Q) = \emptyset}{Q \ \mid\ P \xrightarrow{1.l} Q \ \mid\ P'}$$

$$\text{(RCom0)}\ \frac{P \xrightarrow{\theta_1.!a(\tilde{v})} P' \quad Q \xrightarrow{\theta_2.(\boldsymbol{\nu}\tilde{c})\overline{a}\langle\tilde{v}\rangle} Q' \quad \forall i, c_i \notin \mathsf{fn}(P)}{P \ \mid\ Q \xrightarrow{\langle\theta_1.!a(\tilde{v}),\theta_2.(\boldsymbol{\nu}\tilde{c})\overline{a}\langle\tilde{v}\rangle\rangle} (\boldsymbol{\nu}\tilde{c})(P' \ \mid\ Q')}$$

$$\text{(RCom1)}\ \frac{P \xrightarrow{\theta_1.(\boldsymbol{\nu}\tilde{c})\overline{a}\langle\tilde{v}\rangle} P' \quad Q \xrightarrow{\theta_2.!a(\tilde{v})} Q' \quad \forall i, c_i \notin \mathsf{fn}(P)}{P \ \mid\ Q \xrightarrow{\langle\theta_1.(\boldsymbol{\nu}\tilde{c})\overline{a}\langle\tilde{v}\rangle,\theta_2.!a(\tilde{v})\rangle} (\boldsymbol{\nu}\tilde{c})(P' \ \mid\ Q')}$$

$$\text{(Com0)}\ \frac{P \xrightarrow{\theta_1.a(\tilde{v})} P' \quad Q \xrightarrow{\theta_2.(\boldsymbol{\nu}\tilde{c})\ \overline{a}\langle\tilde{v}\rangle} Q' \quad \forall i, c_i \notin \mathsf{fn}(P)}{P \ \mid\ Q \xrightarrow{\langle\theta_1.a(\tilde{v}),\theta_2.(\boldsymbol{\nu}\tilde{c})\ \overline{a}\langle\tilde{v}\rangle\rangle} (\boldsymbol{\nu}\tilde{c})\ (P' \ \mid\ Q')}$$

$$\text{(Com1)}\ \frac{P \xrightarrow{\theta_1.(\boldsymbol{\nu}\tilde{c})\ \overline{a}\langle\tilde{v}\rangle} P' \quad Q \xrightarrow{\theta_2.a(\tilde{v})} Q' \quad \forall i, c_i \notin \mathsf{fn}(P)}{P \ \mid\ Q \xrightarrow{\theta_1.(\boldsymbol{\nu}\tilde{c})\ \overline{a}\langle\tilde{v}\rangle,\langle\theta_2.a(\tilde{v})\rangle} (\boldsymbol{\nu}\tilde{c})\ (P' \ \mid\ Q')} \qquad \text{(Res)}\ \frac{P \xrightarrow{l} P' \quad a \notin \mathsf{n}(l)}{(\boldsymbol{\nu}a)\ P \xrightarrow{l} (\boldsymbol{\nu}a)\ P'}$$

$$\text{(Open)}\ \frac{P \xrightarrow{\theta.(\boldsymbol{\nu}\tilde{c})\ \overline{a}\langle\tilde{v}\rangle} P' \quad b \in \tilde{v} - \tilde{c} \quad b \neq a}{(\boldsymbol{\nu}b)\ P \xrightarrow{\theta.(\boldsymbol{\nu}b,\tilde{c})\ \overline{a}\langle\tilde{v}\rangle} P'}$$

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

Fig. 2. Labelled Transition System.

Definition 1. (Computations). A *computation* C from process P_0 is a finite or infinite sequence of transitions and processes $(l_k, P_{k+1})_{k \in I \subseteq \mathbb{N}}$ s.t. for all $k \in I$, $P_k \xrightarrow{l_k} P_{k+1}$.

Note that transitions in a computation are uniquely identified by their labels. The ultimate result (soundness) of the paper is stated for computations: we guarantee that a in all computations from a well-typed process, the number of transitions depending on an initial "function call" will be bound by a polynomial of the arguments of the call.

3. Types and typing system

Types. To control potential computational explosions and infinite behaviours, we decorate the types of names used in replicated inputs with natural numbers ***levels*** N, M, similar to the ones from Deng and Sangiorgi [20], 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 Bellantoni and Cook [7]: 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 ($N \geq 0$):

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

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.

We will use *replicated types* (resp. *linear types*) to denote the types of replicated (resp. linear) names.

In a given environment, a channel is either linear or replicated. Replicated channels are used to trigger replications, which mimics function calls. Linear channels are used to get back results from function calls. In the examples, a replicated channel often carries the arguments of a function call and a linear channel, which will be used to retrieve the result of the call.

In the following, we use terms which hint at the computational aspect of the processes we analyse, identifying the subprocesses playing the roles of computing services.

Definition 2 *(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.

Whenever $\Gamma \vdash_M P$ for some M, we denote by $\mathsf{out}_N(P;\Gamma)$ the multiset of all outputs of P which are given level N by Γ, reminiscent of the output set $\mathsf{os}(P)$ from rule (T-Rep) in Deng and Sangiorgi [20]. (we recall $\uplus$ denotes multiset union):

$$\mathsf{out}_N(0;\Gamma) = \mathsf{out}_N(!a(\tilde{x}).P;\Gamma) \overset{\text{def}}{=} \emptyset \qquad \mathsf{out}_N(\overline{a}\langle\tilde{v}\rangle;\Gamma) \overset{\text{def}}{=} \begin{cases} \emptyset & \text{if } \Gamma(a) \neq (\tilde{T})_N \\ \{\overline{a}\langle\tilde{v}\rangle\} & \text{otherwise} \end{cases}$$

$$\mathsf{out}_N((\boldsymbol{\nu} c)\ P;\Gamma) = \mathsf{out}_N(a(\tilde{x}).P;\Gamma) \overset{\text{def}}{=} \mathsf{out}_N(P;\Gamma)$$

$$\mathsf{out}_N(P_1|P_2;\Gamma) = \mathsf{out}_N(([e=0]P_1 + [e\neq 0]P_2;\Gamma) \overset{\text{def}}{=} \mathsf{out}_N(P_1;\Gamma) \uplus \mathsf{out}_N(P_2;\Gamma)$$

Operator $\mathsf{out}_N(P;\Gamma)$ recursively goes down inside the structure of a typed process and collects the multiset of all outputs at level N. It is used, as in Deng and Sangiorgi [20], in rule (Serv) for service definitions, in order to compare the replicated input with the outputs inside the replication.

The multiset $\mathsf{calls}(P;\Gamma)$ of *calls* of typable process P, w.r.t. a context Γ, is defined as $\bigcup_{N\in\mathbb{N}} \mathsf{out}_N(P;\Gamma)$.

Example 3 *(Output multisets).* For instance, consider $Q = (\boldsymbol{\nu} c)\ (\overline{c}\ \mid\ \overline{b}\ \mid\ \overline{a}\langle x-1\rangle\ \mid\ b.\overline{c})$. Replication $!a(x).Q$ is typed with $\Gamma = a : (\mathsf{nat}_\star)_3, b : (), c : ()_1, x : \mathsf{nat}_\star$, and $\mathsf{out}_3(Q;\Gamma) = \{\overline{a}\langle x-1\rangle\}$, $\mathsf{out}_1(Q;\Gamma) = \{\overline{c},\overline{c}\}$ and $\mathsf{out}_2(Q;\Gamma) = \emptyset$.

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, containing 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 $\forall i, \mathsf{nat}_\star \neq T_i$ (resp. $\forall i, \mathsf{nat} \neq T_i$). 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;
3. or $T = (\tilde{S})_N$, all $\tilde{S}$ are well-formed and each S_j which is a linear type is 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.

Remark (Safety of linear channels). In Bellantoni and Cook [7], functions are operating on integers, and each argument of a function call is either a safe integer or an unsafe integer. It is still the case in our framework that integers are either safe of unsafe. However, our system include types for channels (because channels can be passed as "arguments" to our service) and if we identify some channels as safe and some as unsafe, some linear channels can be constructed as neither, or both, such as $(\mathsf{nat},\mathsf{nat}_\star)$.
Yet, the well-formedness conditions ensure that all linear channel types appearing in well-formed typing contexts are either safe or unsafe. The reason is that the return channel passed in a call to a service has to be safe (because the value it receives must be "marked", so that it will not be used for recursion), which forces all linear channels to appear directly inside a replicated type to be linear. Moreover, services can call auxiliary services on lower level without risk of breaking

complexity bounds, in this case they use unsafe channels inside calls (see rule (UOut) below). As a result, only "fully safe" linear channels and "fully unsafe" linear channels are used in the computation flow.

Example 4 *(Well-formed and ill-formed types).* For instance $T_1 = (\mathsf{nat}_\star, \mathsf{nat}_\star, (\mathsf{nat}))_1$ is well-formed as the inside channel type is safe. The typing rules for output (in Fig. 3) 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 $T_2 = (\mathsf{nat}_\star, \mathsf{nat}_\star, (\mathsf{nat}_\star))_1$ 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 Demangeon [17], 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 $N \in \mathbb{N} \cup \infty$. 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.

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; otherwise $[T]_\star = T$.

The notion of lifting is somehow reminiscent of the *declassification* in Marion [27] where neutral (not size-increasing) operations on variable X inside a loop controlled by X are allowed. However, in our system, lifting allows a name received by a higher-level service to be used freely by low-level service, whereas in Marion [27] the variable X might be given a lower level if the auxiliary operations are neutral.

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 $e_i \in \tilde{e}_r$ s.t. $T_i = \mathsf{nat}_\star$ and *safe* arguments $e_j \in \tilde{e}_s$ s.t. $T_j = \mathsf{nat}$. Comparisons between tuples of integer expressions use the product composition of the ordering given in Section 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.

Example 5 *(Comparisons).* 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 $\overline{a}\langle x-1, y, z, d\rangle \subseteq P$. 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 $(x-1, z; y) < (x, z; y)$. This ensures that recursive calls of well-type services are done on an argument strictly smaller (than the initial call) for a well-founded ordering.

Typing rules. The typing of values is defined in the first line of Fig. 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 Fig. 3. (Nil) states that **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 used 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 $\mathsf{out}_N(P; \Gamma)$ 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 8 explains the reason).

The syntactic check in rule (Serv) could have been done in a separated analysis, but we choose to integrate it inside the type system as it is a part of the translation of the analysis from Bellantoni and Cook [7]. We believe the syntactic presentation (as multiset of calls) is more clear than adding another layer in the typing context.

Value Typing

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

$$\frac{\Gamma\vdash e:\mathsf{nat}}{\Gamma\vdash e-1:\mathsf{nat}}\qquad \frac{\Gamma\vdash e:\mathsf{nat}_\star}{\Gamma\vdash e-1:\mathsf{nat}_\star}\qquad \frac{\Gamma\vdash e:\mathsf{nat}}{\Gamma\vdash e+1:\mathsf{nat}}\qquad \frac{\Gamma\vdash e:\mathsf{nat}_\star}{\Gamma\vdash e+1:\mathsf{nat}_\star}$$

Process Typing

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

$$(\mathsf{Res})\,\frac{\Gamma\vdash_N P}{\Gamma\vdash_N (\boldsymbol{\nu}c)\ P}\qquad (\mathsf{Cond})\,\frac{\Gamma\vdash_N P_i\quad (i=1,2)\quad \Gamma\vdash e:\circ\mathsf{nat}}{\Gamma\vdash_N [e=0]P_1+[e\neq 0]P_2}\qquad (\mathsf{SOut})\,\frac{\Gamma\vdash u:(\tilde{T})_M\quad \Gamma\vdash\tilde{e}:\tilde{T}\quad M\le N}{\Gamma\vdash_N \overline{u}\langle\tilde{e}\rangle}$$

$$(\mathsf{UOut})\,\frac{\Gamma\vdash u:(\tilde{T})_M\quad \Gamma\vdash\tilde{e}:[\tilde{T}]_\star\quad M<N}{\Gamma\vdash_N \overline{u}\langle\tilde{e}\rangle}$$

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

Fig. 3. Typing rules.

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

Fig. 4. Examples of services.

Example 6 *(Typing replications).* For instance, consider the example above $P = !a(x).(\boldsymbol{\nu}c)\ (\overline{c}\ \mid\ \overline{b}\ \mid\ \overline{a}\langle x-1\rangle\ \mid\ b.\overline{c})$ with $\Gamma = a:(\mathsf{nat}_\star)_3, b:(), c:()_1, x:\mathsf{nat}_\star$. We obtain directly premises $\Gamma\vdash a:(\mathsf{nat}_\star)_3$ 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 $\Gamma\vdash_3 (\boldsymbol{\nu}c)\ (\overline{c}\ \mid\ \overline{b}\ \mid\ \overline{a}\langle x-1\rangle\ \mid\ b.\overline{c})$. We have computed above $\mathsf{out}_P(\Gamma;3)=\{\overline{a}\langle x-1\rangle\}$, 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 $\Gamma\vdash_\infty !a(x).(\boldsymbol{\nu}c)\ (\overline{c}\ \mid\ \overline{b}\ \mid\ \overline{a}\langle x-1\rangle\ \mid\ b.\overline{c})$; any process containing P will be typed at level ∞, preventing it to occur inside another replication, as replicated names have finite levels.

Example 7 *(Simple examples of typing).* We explain here how the type system validates predicativity of recursion on distributed services using the examples of Fig. 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:

$$\begin{array}{rcl|rcl|rcl}
\Gamma_1(add) & = & (\mathsf{nat}_\star,\mathsf{nat},(\mathsf{nat}))_1 & \Gamma_1(x) & = & \mathsf{nat}_\star & \Gamma_1(y) & = & \mathsf{nat}\\
\Gamma_1(r) & = & (\mathsf{nat}) & \Gamma_1(c) & = & (\mathsf{nat}) & \Gamma_1(z) & = & \mathsf{nat}
\end{array}$$

The typing derivation is presented in Fig. 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:

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

$$
\text{(Serv)}\;\dfrac{
\dfrac{\text{(SOut)}\;\dfrac{\Gamma_2 \vdash mult : (\mathsf{nat}_\star, \mathsf{nat}_\star, (\mathsf{nat}))_2 \quad \Gamma_2 \vdash x-1, y, d_1 : \mathsf{nat}_\star, \mathsf{nat}_\star, (\mathsf{nat})}{\overline{mult}\langle x-1, y, d_1\rangle}}{\ldots}
\quad
\dfrac{\text{(SOut)}\;\dfrac{\Gamma_2 \vdash add : (\mathsf{nat}_\star, \mathsf{nat}, (\mathsf{nat}))_1 \quad 1<2}{\Gamma_2 \vdash_2 \overline{add}\langle y, res, d_2\rangle}}{\ldots}
\quad
\begin{array}{l}\mathsf{out}_2(\ldots;\Gamma_2) = \{\overline{mult}\langle x-1, y, d_1\rangle\} \\ \wedge (x-1, y;\,) < (x, y;\,)\end{array}
}{\Gamma_2 \vdash_\infty !mult(x, y, r).[x=0]\;\overline{r}\langle 0\rangle\; + [x\neq 0]\;\;(\boldsymbol{\nu} d_1, d_2)\;\;(\overline{mult}\langle x-1, y, d_1\rangle \;\mid\; d_1(res).\overline{add}\langle y, res, d_2\rangle \;\mid\; d_2(z).\overline{r}\langle z\rangle)}
$$

Fig. 5. Typing derivation for ***A*** and ***M***.

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

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 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 ***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 ***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 ***F***.

(4) Unsafe calls in a polytime service. Process ***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 ***L*** describes a diverging behaviour between two services a and b. When trying to type-check it, these names have to be given recursive channel types. It is not typable with any environment $\Gamma = \{a : ()_{N_1}, b : ()_{N_2}\}$.

Rule (SOut) for $\overline{b}$ inside $!a.\overline{b}$ forces $N_2 \leq N_1$ and Rule (SOut) for $\overline{a}$ inside $!b.\overline{a}$ forces $N_1 \leq N_2$.

So $N_1 = N_2$ but rule (Serv) forces $\overline{b}$ to be treated as a recursive call in $!a.\overline{b}$, and, as result, to exhibit a decreasing in its arguments. As a and b do not carry any value, the process cannot be typed.

Our type system enforces termination as in Deng and Sangiorgi [20].

(6) Multiple recursive calls. Process ***E*** performs an exponential number of transitions on a since each call $\overline{a}\langle N\rangle$ spawns two recursive calls $\overline{a}\langle N-1\rangle$. 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 $\mathsf{out}_N(P;\Gamma)$ 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 ***E***.

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

(8) Higher-order service. Process ***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 ***H*** is typable by our typing system.

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

Fig. 6. Examples guiding the causality definition.

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 Bellantoni and Cook [7]. Therefore, one would expect that our system guarantees polynomial bounds for services, just as Bellantoni and Cook [7] characterises polytime functions. The following process $\boldsymbol{CE}$ is a counterexample:

$$\begin{aligned}
&inc = d(z).\overline{d}\langle z+1\rangle\\
&\boldsymbol{CE} = !add(x,y,r).[x=0]\overline{r}\langle y\rangle + [x \neq 0](\boldsymbol{\nu} c)\ (\overline{add}\langle x-1,y,c\rangle \mid c(z).\overline{r}\langle z+1\rangle \mid inc)\\
&\quad \mid !mult(x,y,r).[x=0]\overline{r}\langle 0\rangle + [x \neq 0](\boldsymbol{\nu} c_1,c_2)\ (\overline{mult}\langle x-1,y,c_1\rangle \mid c_1(z_1).\overline{add}\langle y,z_1,c_2\rangle \mid c_2(z_2).\overline{r}\langle z_2\rangle \mid inc)\\
&\quad \mid !fact(x).[x=0]\overline{r}\langle 1\rangle + [x \neq 0](\boldsymbol{\nu} c)\ (\overline{fact}\langle x-1\rangle \mid d(z).(\overline{mult}\langle z,x-1,c\rangle \mid \overline{d}\langle z+1\rangle \mid \overline{d}\langle 1\rangle))
\end{aligned}$$

This process is similar to the one of Fig. 1: the main difference is that addition and multiplication services include an incrementer module *inc* 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 Fig. 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 $\overline{fact}(N)$ can produce N recursive calls to *fact*, spawn $N!$ copies of *inc* and thus generate more than $N!$ reductions. Yet, $\boldsymbol{CE}$ is typable by the typing system in Fig. 3 and predicativity of recursion is not violated: the process is not using the result of a recursive call in a recursion position. Indeed, 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 Section 5 which 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 Section 4.

Remark. Note that channel d can be seen as a shared memory, and that complexity analyses focusing on multi-threaded processes manipulating shared-memory, such as Hainry et al. [21] could probably be adapted to control this behaviour.

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 Degano and Priami [16], Degano et al. [15,14] 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 [16] relates a transition to a previous output transition that extrudes one of its names. Since our analysis focuses on services (implemented through replications) [14], we need to cut undesirable causality links.

Example 8. (1) Independence of replications. In Degano et al. [14], subsequent firings of the same replicated input are causally related. Consider a replicated input $!a(10)$ to $\boldsymbol{S}_0$ of Fig. 6 which spawns 10 messages. A subsequent independent replicated input $0.!a(100)$ 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 Degano et al. [14] 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 Degano et al. [14], 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 Degano et al. [14] 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 Degano et al. [15], the transitions depending on request $0.!a(5)$ are interleaved with those depending on request $1.!b(10)$, and communications on linear names (called, in the following, *linear communication*) 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 Degano and Priami [16] 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 Degano and Priami [16] 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 Degano et al. [14] 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 Section 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 11). Informally, the definitions in Degano and Priami [16], Degano et al. [15,14] 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 9). As explained above, it weakens the structural causality of Degano et al. [14] and ignores binding causality of Degano and Priami [16] to define a dependency relation ($\subseteq_{\mathrm{d}}$ 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 9 *(Service causality).*

1. A *causality relation* between labels ($l \subseteq l'$) is defined by the following rules:

   (1) $a(\tilde{v}) \subseteq_{\mathrm{d}} l$ (r1) $!a(\tilde{v}) \subseteq_{\mathrm{d}} 1.l$
   (2) $i.l \subseteq_{\mathrm{d}} i.l'$ if $l \subseteq_{\mathrm{d}} l'$
   (3) $\langle l_0, l_1\rangle \subseteq_{\mathrm{d}} \langle l'_0, l'_1\rangle$ if $l_i \subseteq_{\mathrm{d}} l'_j$ for some i, j and $!a(\tilde{v}) \in \langle l_0, l_1\rangle$
   (4) $\langle l_0, l_1\rangle \subseteq_{\mathrm{d}} l'$ if $l_i \subseteq_{\mathrm{d}} l'$ for some i and $!a(\tilde{v}) \in \langle l_0, l_1\rangle$
   (5) $l \subseteq_{\mathrm{d}} \langle l'_0, l'_1\rangle$ if $l \subseteq_{\mathrm{d}} l'_j$ for some j

2. Let C be a computation and $(i, j) \in \mathbb{N}^2$ two indices of C with $i < j$. We say that transition l_j *depends on* l_i in C, written by $l_j \subseteq_C l_i$ iff $l_j \subseteq_{\mathrm{d}} l_i$. 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 on 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 (since we do not need them to identify the initial request associated to a call).

Example 10 *(Separating computation).* Suppose $\boldsymbol{S} = !a(x, y).[x \neq 0](\nu d)\ (\overline{a}\langle x-1, y\rangle \mid \overline{d} \mid d.\overline{b} \mid b.\overline{y})$.
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 9. Each replicated input $\theta.!a(N, r)$ will cause a number of transitions linear in N.

Theorem 11 characterises the effect of the transformations we apply to the causality definition from Degano et al. [14]: 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: that is, each internal step of computation is caused by a unique "initial request" from the environment. 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 11 *(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 $i_1 = i_2$.*

Proof. Without loss of generality we suppose $i_1 \leq i_2$. According to Definition 9 of $\subseteq$, there exist two chains of relation $\subseteq_{\mathtt{d}}$ from l_{i_1} to l_j and from l_{i_2} to l_j. We notice $l_{i_1} \not\subseteq_{\mathtt{d}} l_{i_2}$ as P (and the process obtained by computations from P) does not contain nested replication. We pose j_1, j_2, j' elements of the $j_1 \leq j_2$, $l_{j_1} \subseteq_{\mathtt{d}} l_{j'}$ and $l_{j_2} \subseteq_{\mathtt{d}} l_{j'}$, $l_{i_1} \subseteq l_{j_1}$ (through a chain of relations C_1), $l_{i_2} \subseteq l_{j_2}$ (through C_2), $l'_j \subseteq l_j$ and $l_{j_1} \not\subseteq_{\mathtt{d}} l_{j_2}$. Notice l'_j is not a linear communication, thanks to the hypothesis and side-conditions of Rules (3, 4). We proceed by induction on C_1, C_2. If both are of size 1, it means l_{j_1} and l_{j_2} are two replicated inputs guarding $l_{j'}$ we conclude from the absence of nested replications. Otherwise, we discuss the nature of transitions (l_{j_1}, l_{j_2}):

1. none are linear communications, because of side conditions of Rules (3, 4).
2. none is an output because of asynchrony.
3. if one is a linear input, for instance l_{j_1} is $\theta_1.c_1(x_1)$. We consider $l_{j'_1}$ the name directly behind l_{j_1} in C_1. That is, $l_{j'_1} \subseteq_{\mathtt{d}} l_{j_1}$. We notice that $l_{j'_1} \subseteq_{\mathtt{d}} l_{j'}$, as $l_{j'_1}$ prefixes l_{j_1} which prefixes an action inside $l_{j'}$. We use the induction hypothesis on C'_1, C_2 with C'_1 the chain C_1 without its last element.
4. if both involve replicated inputs (that is, each one is either a replicated input or a replicated communication), $l_{j'}$ is related with $\subseteq_{\mathtt{d}}$ to both replicated inputs (it cannot be related to the output side of a replicated communication because of asynchrony). If a single label of $l_{j'}$ is related to both replicated inputs, we conclude, thanks to the absence of nested replication (one prefix can only be guarded by a single replication). Otherwise it means $l_{j'}$ is a replicated communication (as it cannot be a linear communication) $\theta'\langle\theta'_1.!d_1(\tilde{v}_1), \theta'_1.\overline{d_2}\langle\tilde{v}_1\rangle\rangle$ with the output $\overline{d_2}$ guarded by one replicated input and $!d_1$ guarded by the other. As there is no nested replication, we obtain a contradiction.

On Nested Replications. Our definition of causality requires the initial process not to contain any nested replication. The reason for this restriction is that it makes the process of finding the "initial cause" of given transition easier, because it guarantees that one transition has at most one "initial cause".

The effect this constraint has on expressiveness is that services cannot be created dynamically.

Our system can be accommodated to lift this constraint at the cost of a more technical (and more specific) definition for causality. In the case of a nested replication $!s_1(\ldots).C[!s_2(\ldots).P]$, the new definition of the causality relation should relate a transition of a prefix appearing in a copy of C (not appearing in another nested replication) with the transition triggering s_1, and every transition fired from a prefix in a copy of P by the transition triggering s_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 ***CE*** from Section 3, our analysis needs to detect that, in this particular example, some information is passed through different recursive calls, using channel d. Indeed, the main culprit in the case of ***CE*** not being polytime 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, and thus controlled by types), 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 been 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 $\mathsf{sub}(a, P)$ (resp. $\mathsf{obj}(a, P)$).

Definition 12 *(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 $\mathsf{ok}(e, Q \subseteq P)$, whenever either:

1. e does not contain any variable;
2. e contains variable x_i, bound in $!a(\tilde{x}).R \subseteq Q$; or
3. e contains variable y_i, bound in $b(\tilde{y}).R \subseteq Q$ with:

   (a) b is bound by restriction $(\boldsymbol{\nu} b)\ R' \subseteq Q$;
   (b) $\mathsf{sub}(b, R') = \{b(\tilde{y})\}$; and
   (c) $\mathsf{obj}(b, R') \subseteq \{\overline{d}\langle\ldots, b, \ldots\rangle\}$

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

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

$$\boldsymbol{U_2} = !a(x,r).(\boldsymbol{\nu} c)\ (\overline{a}\langle x-1, c\rangle \mid c(z).\overline{r}\langle z\rangle \mid \overline{s}\langle r\rangle)$$

$$\boldsymbol{U_3} = !a(x,r).(\boldsymbol{\nu} c)\ (\overline{a}\langle x-1, c\rangle \mid c(z).\overline{r}\langle z\rangle \mid d(y).\overline{r}\langle y\rangle)$$

Fig. 7. Examples of sound and unsound processes.

When process Q and P are clear from context (especially when Q is a "service" defined as a replicated input), we write that *e is controlled* for "e is controlled in $Q \subseteq P$".

We recall that integer expressions contain at most one variable. Definition 14 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") (a) are not passed as arguments and (b) outputs on them only contain controlled integers.

The use of sets (and not multisets) for subject and object prefixes is innocuous: $b(\tilde{y})$ can only appear once in the process as $\tilde{y}$ is fixed and multiple calls $\overline{d}$ can only appear in the case of unsafe calls.

Example 13 *(First idea of control flow).* Consider process $\boldsymbol{S}$ from Fig. 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 in $\boldsymbol{S}$**, 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 $(\boldsymbol{\nu} c)$ 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.

When every crucial expression is controlled in a process, we describe this process as ***sound***.

Definition 14 *(Sound process).* Let P be a process s.t. $\Gamma \vdash_N P$ for some Γ, N. We write $\mathsf{sound}(P)$ whenever, for each subprocess $!a(\tilde{x}).Q \in P$:

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

Example 15 *(Control and soundness).* We can check that $\boldsymbol{S}$ from Example 13 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 integer expressions 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 $\boldsymbol{CE}$ from Section 3 is unsound: expression z used in the call to *mult* must be controlled in *fact* service definition, according to rule (1) of Definition 14. By Definition 12, 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 12(3.a), which forbids a usage in a critical position for integers received on free channels free inside the service definitions.

Example 16 *(Sound and unsound processes).* We describe here how the flow analysis proceeds for each example from Fig. 4. Section 7 will provide additional details on how both the type system and the flow analysis work on several counter-examples.

- 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.

  The soundness analysis for $\boldsymbol{A}$ is done as follows:

  1. First, we have to check that each integer appearing in outputs inside the service definition is controlled. The only call is the recursive output on *add*, its integer expressions are $x-1$ and y and they are bound in the replication itself (item 2. in Definition 12).
  2. Then, we need to check the usage of the channel appearing in the replicated input, which is r. It never appears in object (item 2.a) and when it appears in subject, in $\overline{r}\langle 0\rangle$ and $\overline{r}\langle z+1\rangle$, its integer argument 0 and $z+1$ are controlled. Indeed 0 is a free expression and z is bound in $c(z).\overline{r}\langle z+1\rangle$. In the latter case, we follow item 3 of Definition 12: c is bound in a restriction appearing in the continuation (item 3.a), there is only one input on c (input $c(z)$) (item 3.b), and there is only one prefix in which c appears in object position (the recursive call $\overline{add}\langle x-1, y, c\rangle$) (item 3.c).

- 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 12. An external input $mult(N, M, c)$ spawns $\Theta(N*M)$ transitions.

  The flow analysis of $\boldsymbol{P}$ is done as follows:

  1. Everything related to the α-conversion of $\boldsymbol{A}$ is done as above.
  2. In the service definition of *mult*, we have to check that integers appearing in calls are controlled. These expressions are x, y, res. Reasoning for x and y is done as above. For *res*, we use item 3 of Definition 12: it is bound in an input $d_1(res)$. Name d_1 is bound in a restriction, there is only one prefix whose subject is d_1 (input $d_1(res)$). Moreover, d_1 appears only once in object position, in $\overline{mult}\langle x-1, y, d_1\rangle$.
  3. The channel r passed on *mult*, is used as the channel r in the previous example, and the reasoning is similar.

- 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 12. An external input $!cube(N, c)$ spawns $\Theta(N^3)$ transitions.

  We do not give the detailed reasoning for typability and soundness of $\boldsymbol{C}$, as it is similar to $\boldsymbol{A}$ and $\boldsymbol{P}$. Instead, we regroup some interesting remarks:

  1. *cube* has to be given level 3 (or greater) in order for the auxiliary calls to *mult* of level 2 to be typed. Its type is $(\mathsf{nat}_\star, (\mathsf{nat}))_3$.
  2. as *mult* is on a strictly lower level that 3, several calls to *mult* are possible.
  3. as *mult* is of type $(\mathsf{nat}_\star, \mathsf{nat}_\star, (\mathsf{nat}))_2$, its second argument has to be an unsafe integer. Thus y, in the second call to *mult* as to be of type $\mathsf{nat}_\star$, meaning that c has type $(\mathsf{nat}_\star)$.
  4. channel d is given type (nat) (the value received on d need not be unsafe).
  5. the flow analysis ensures that z is bound in $d(z).\overline{r}\langle z\rangle$, that d abides to Definition 12.3, that y is bound in $c(y).\overline{mult}\langle x, y, d\rangle$ and that c abides to Definition 12.3 as well.

- Process $\boldsymbol{H}$ is sound and exhibits a polytime 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 polytime 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 polytime: 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 14 and 12 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 polytime as dangerous integers can be directly received from the outside.

6. Soundness, completeness and decidability

Soundness of the analysis (Theorem 29) guarantees a polynomial bound on sound processes, and requires Subject Transition (Theorem 19) which states that soundness is stable through typable transitions. Completeness (Theorem 31) states that polytime functions can be computed by typable processes.

6.1. Subject transition theorem

When there is no ambiguity, we use the label l of a transition to denote the transition itself.

Lemma 17 *(Subject substitution). Soundness of processes is preserved by substitution, that is:*
$\mathsf{sound}(P)$ *with* $\Gamma \vdash_N P$ *and* $\Gamma(\tilde{v}) = \Gamma(\tilde{x}) = \tilde{T}$, *then* $\mathsf{sound}(P[\tilde{v}/\tilde{x}])$ *with* $\Gamma \vdash_N P$.

Proof. We easily prove that flow analysis rules are preserved by substitutions, as all the names involved are bound.
We prove typability by induction on the typing derivation. As $\tilde{v}$ and $\tilde{T}$ have same type, replacement is easy. □

In order to prove subject transition, we need to formally define the notion of typable transition: a transition is typable whenever it is an internal communication, an output which does not extrude linear channels, or an input whose objects are either typable or fresh. Extruding linear channels would be dangerous as they could be used to introduce unbound integers from the outside, and this condition is used in the main proof. Note that it prevents a process with a service definition on a to send a recursive call to an external (and potentially untyped) service.

Definition 18 *(Typable transitions).* If $\Gamma \vdash_N P$, and $P \xrightarrow{l} P'$, we say that l is a *typable* transition of P w.r.t. Γ when either:

1. $l = \theta.\tau$; or
2. $l = \theta.(\nu\tilde{c})\ \overline{a}\langle\tilde{v}\rangle$ and if $\Gamma \vdash a : (\tilde{T})_N$ and $T_j = (\tilde{U}_j)$, then $v_j \notin \tilde{c}$;
3. $l = \theta.a(\tilde{v})$ (resp. $l = \theta.!a(\tilde{v})$), $\Gamma \vdash a : (\tilde{T})$ (resp. $\Gamma \vdash a : (\tilde{T})_N$) and for all $v_k \in \Gamma$, $\Gamma \vdash v_k : T_k$.

We omit Γ when it is clear from context. A *typable computation from* P_0, $C = (k, \Gamma_{k+1}, P_{k+1})_k$ is a computation from P_0 s.t. for all k, $\Gamma_k \subseteq \Gamma_{k+1}$, $\Gamma_k \vdash_{N_k} P_k$ for some N_k and transition l_k typable w.r.t. Γ_k.

Note that $\Gamma_k \subseteq \Gamma_{k+1}$ because fresh names might have been received from the outside.

Theorem 19 *(Subject transition). Soundness of processes is preserved by transition, that is, if* $\mathsf{sound}(P)$ *with* $\Gamma \vdash_N P$, $P \xrightarrow{l} P'$ *and* l *typed w.r.t.* Γ, *then* $\mathsf{sound}(P')$.

Proof. Because there is no extrusion of linear names (which would break rule 3.a), we easily prove that flow analysis rules are preserved by typable transitions, as they only concern replicated subprocess of P, persistent by transitions. We apply Lemma 17 to the replicated process.
By induction on the transition derivation.

1. Case (Out). We use $\Gamma \vdash_N 0$ for any N.
2. Case (In). We use the fact that $l = a(\tilde{v})$ is typable with $a : (\tilde{T})$. By Definition 18 of typable transition Γ types received values of $\tilde{v}$ according to $(\tilde{T})$. We apply Lemma 17 to the premise of Rule (In) typing P and conclude.
3. Case (Rep). We use the fact that $l = !a(\tilde{v})$ is typable with $a : (\tilde{T})_{N'}$. By definition of typable transitions, Γ types received values of $\tilde{v}$, according to $(\tilde{T})$. Replicated process is trivially typable. For the spawned process, we apply Lemma 17 to the premise of Rule (Serv) typing P. We build Γ' by adding to Γ new type assignment for new bound variable of the spawned process, using Lemma 25.
   An application of Rule (Par) is needed to conclude.
4. Case (Comm) is done using induction hypothesis.
5. Case (Res): direct.
6. Case (Open): direct.
7. Other cases are either similar to the previous ones, or trivial. □

A corollary of Theorem 19 is that computations are typable whenever the components of the external actions performed during the computation are matching the expected types in the initial context Γ; that is, whenever a process P_k performs input $a(\tilde{v})$, the content $\tilde{v}$ has been assigned types in Γ which match the type of the objects of a.

6.2. Analysis soundness

We say that channel a *instantiates* name u in a computation S when the substitution $[a/u]$ is applied in one transition of S or when $a = u$.
In a computation, the *depending set* of transition l is the set of all posterior transitions causally related to l.

Definition 20 *(Dependending set).* Let S be a computation $(P_k, l_k)_{k \in I \subseteq \mathbb{N}}$ from P. The *depending set* of transition l_m in S, denoted by $\mathbf{D}(l_m)_S$, is the set $\{l_k | l_m \subseteq l_k\}$ of all transitions of S which depend from l_m. We write $\mathbf{D}(l_m)$ when S is clear from context.

If $\tilde{e}$ is a tuple of expressions, we write $|\tilde{e}|$ (the size of $\tilde{e}$) as the sum of all integer expressions in $\tilde{e}$. A process P is bound by F when in all typable computations from P, an external input $!a(\tilde{v})$ does not generate more than $F(|\tilde{v}|)$ causally dependent actions, that is, F computed on the size of $\tilde{v}$.

We use $\#(E)$ to denote the cardinal of set E.

Definition 21 *(Complexity bound).* We say that a typable process P is *bound by function* $F : \mathbb{N} \to \mathbb{N}$ whenever for computation $S = (P_k, l_k)_{k \in I \subseteq \mathbb{N}}$ from P, for any m, if $l_m = \theta.!a(\tilde{v})$ then $\#(\mathbf{D}(l_m)_S) \le F(|\tilde{v}|)$. We say that P is bound by a polynomial when there exists a polynomial $\mathcal{P}$ s.t. P is bound by function $\mathcal{P}$.

In a computation, one transition labelled l_2 is a *consequence* of transition l_1 whenever $l_1 \subseteq_{\mathrm{d}} l_2$. It denotes that one path in l_1 is a prefix of one path in l_2, according to the rules of Definition 9, Lemma 22 states that the depending set of an input or a replicated communication can be computed as the union of the depending sets of the consequences, and that the depending set of a linear communication or an output is empty. We prove it by exploring $\mathbf{D}(l_m)$, following directly the rules from Definition 9 and the definition of $\subseteq$ as their transitive and reflexive closure.

Lemma 22 *(Depending set characterisation). Let $\Gamma \vdash_N P$ and $S = (l_k, \Gamma_{k+1}, P_{k+1})_{k \in I \subseteq \mathbb{N}}$ a typable computation from P. Then we have:*

1. *if $l_0 = \theta.(\boldsymbol{\nu}\tilde{c})\ \overline{a}\langle\tilde{v}\rangle$ or $l_0 = \theta_0.\langle\theta_1.a(\tilde{v}), \theta_2\rangle$, then $\mathbf{D}(l_0) = \{l_0\}$.*
2. *if $l_0 = \theta.c(\tilde{v})$, $l_0 = \theta.!c(\tilde{v})$ or $l_0 = \theta_0.\langle\theta_1.!a(\tilde{v}), \theta_2\rangle$, then $\mathbf{D}(l_0) = \{l_0\} \cup \bigcup_{l_0 \subseteq_{\mathrm{d}} l_k} \mathbf{D}(l_k)$.*

Proof. By examining the rules of Definition 9, we can distinguish between the direct consequences of transitions (the ones given by the rules) and the indirect causally related transitions (given by the transitivity of the relation). It turns out that every transitivity chain ends up with one consequence of l_0. Moreover, outputs and linear inputs, thanks to our definition, do not have any direct consequences (nor any causally related further transitions). □

Bound consequences. Lemma 23 bounds the consequences of a replicated input by a constant (depending on the name it instantiates) and its proof uses the absence of nested replications. The proof uses the fact that P is free of nested replications to bound the number of consequences a transition has. It is used in the following proof.

Lemma 23 *(Bound consequences). If* $\mathsf{sound}(P)$*, then for any name u and any typable computation $S = (l_k, \Gamma_{k+1}, P_{k+1})_{k \in I \subseteq \mathbb{N}}$ from P there exists an integer bound K such that $\#(\{l_k | l_m \subseteq_{\mathrm{d}} l_k\}) \le K$ if $l_m = \theta.!a(\tilde{v})$ where name a instantiates u.*

Proof. Rule (Serv) prevents nested replications, so no transition prefixed by $\theta.1$ can be an internal replicated input. As a consequence, the size of subprocess (in number of prefixes) at path $\theta.1$ is bound by the size of the subprocess $!a(\tilde{x}).Q$ at path θ in P_m (1) which is bound by the size of initial process P. In all computations S from P, the size of a replicated subprocess $!a(x).Q$ is bound by $\mathbf{s}$ the maximum size of a replicated subprocess in P (2), as typable computations cannot make a process grow in size under a "!" and cannot create new replicated subprocesses. From (1) and (2) we define K as $\mathbf{s}$. □

Lemma 24 *(Level weakening). If $\Gamma \vdash_N P$ and $M \ge N$, then $\Gamma \vdash_M P$.*

Proof. Easily done by induction on the typing derivation. □

Lemma 25 *(α-conversion). If $\Gamma \vdash_N P$ and $P \equiv_\alpha P'$, then $\Gamma' \vdash_N P'$ for some Γ'.*

Proof. Easily done by induction on the typing derivation. □

Definition 26 *(Origin).* The *origin* of expression e in process P, denoted $\mathrm{org}_P(e)$ (or $\mathrm{org}(e)$ when P is clear from context), is (i) $\perp$ if e does not contain any variable or bound name, (ii) input prefixed subprocess $b(\tilde{y}).R$ or $!a(\tilde{y}).R$ in P if expression $e \in R$ contains variable y_i or (iii) restricted subprocess $(\boldsymbol{\nu}c)\ R$ in P if $e = c$.

Thus, Definition 14 can be rephrased as:

Definition 27 *(Controlled (rephrased)).* Let P be a typable process, $!a(\tilde{x}).Q$ a subprocess of P, and e an *integer expression* appearing in Q. We say that e is *controlled in* $!a(\tilde{x}).Q$ *of* P, noted $\mathsf{ok}(e, !a(\tilde{x}).Q \in P)$ (or $\mathsf{ok}(e)$ when $!a(\tilde{x}).Q$ and P is clear from context), whenever one of the following holds:

1. $\mathsf{org}_P(e) = \bot$,
2. $\mathsf{org}_P(e) = !a(\tilde{x}).R$,
3. or $\mathsf{org}_P(e) = b(\tilde{y}).R \subseteq Q$, in this case:

   (a) $\mathsf{org}_P(b) = (\boldsymbol{\nu} b)\ R' \subseteq Q$,
   (b) $\mathsf{sub}(b, R) = \{b(\tilde{y})\}$,
   (c) $\mathsf{obj}(b, R) = \{\overline{d}\langle \dots, b, \dots \rangle\} \subseteq \mathsf{calls}(Q\,;\,)$,

We call *controlled channels* the linear channels inside service definition which abide to rule 3 of Definition 12. When considering a typable transition sequence starting in P_0, we say that name a instantiates name $u \in P_0$ whenever an input or communication of the sequence substitutes name u with a.

Output control. Lemma 28, crucial prerequisite of Theorem 29, bounds the content of messages in controlled output consequences 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 Bellantoni and Cook [7]. Below the side conditions (i) and (ii) ensure that we only consider calls. We prove it by induction on regions, building polynomials w.r.t. the auxiliary calls done from the inside of replication. Theorem 29 states that our analysis rules out non-polytime behaviours. Its proof is done by induction on regions and by examining the depending sets of a replicated input transition $\theta.!a(\tilde{v})$ with Lemmas 22 and 23. Lemma 28 is used to ensure that calls to auxiliary services are done on arguments polynomial in $|\tilde{v}|$.

Lemma 28 *(Output control). Assume P is a sound process typed with $\Gamma \vdash_{N_m} P$ and N the level of a service of P. Then there exists a monotone polynomial $\mathcal{P}_N$ s.t. if $S = (P_i, l_i)_{i \in I \subseteq \mathbb{N}}$ is a computation from P, two 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 a of level N and $\tilde{v} = \tilde{v}_r; \tilde{v}_s$ up to permutation,*
3. *l_j contains $\overline{b}\langle \tilde{v}' \rangle$ with b,*

then $v'_j \le \mathcal{P}_N(|\tilde{v}_r|) + |\tilde{v}_s|$.

Proof. If $(\tilde{e} \triangleleft \tilde{T}) = (\tilde{e}_r; \tilde{e}_s)$, we write $\tilde{e}_r = \tilde{e}\|^{\star}_{\tilde{T}}$ and $\tilde{e}_s = \tilde{e}\|_{\tilde{T}}$.

We notice that, thanks to Definition 14, inside the replicated input $!c(\tilde{x}).Q$ with c of type $(\tilde{T})_N$, received integers in the typing environment are either integers of $\tilde{x}$, or received through controlled linear channels. We build the polynomials $\mathcal{F}_N$ by induction on levels N.

Case (1): if level 1 is minimal then there is at most one recursive call with parameters $\tilde{x}' < \tilde{x}$ and no call to lower levels. We examine all integer expressions in all replications $!c(\tilde{x}).Q$ on level 1. We take N the sum of each integer appearing in these expressions and we set $\mathcal{F}_N(X) = N * X$.

Case (2): Otherwise we examine a replication $!c(\tilde{x}).Q$ with c of type $(\tilde{T})_N$. We pose M the sum of all integers appearing in expressions. We define a relation $<$ on outputs present inside Q as $\overline{u_1}\langle \tilde{w}_1 \rangle < \overline{u_2}\langle \tilde{w}_2 \rangle$ whenever there exists a request name d s.t. $\overline{u_2}\langle \tilde{w}_2 \rangle$ is guarded by an input on d, u_1 is a replicated name and $d \in \tilde{w}_1$. We consider one enumeration E of the K outputs u_p of the replication following the normalising components of $<$. We write $E|_k$ to denote the first k elements of E. Induction hypothesis gives monotone polynomials $\mathcal{F}^p$ for these replicated outputs (we set $\mathcal{F}^p(X) = M * X$ for the linear one). We set $\mathcal{F}_{E|_k}(X) = (M * \mathcal{F}_k(M * \dots \mathcal{F}_1(M * X) \dots))$.

We pose $\mathcal{F}^E(X)$ the solution of the equation $\mathcal{F}^E(X) = M + \mathcal{F}^E(X-1) + \mathcal{F}_E(X)$, which is polynomial. We set $\mathcal{F}_N$ the sum of all $M + \mathcal{F}_E$ for all enumerations in all replications guarded by names of level N.

Now we prove this obtained polynomial gives output bounds in transitions as stated in Lemma 28. Suppose a is an instance of name c in P (that is a is c or has been substituted to c in S). We proceed by induction on N.

Case (1): If level 1 is minimal we recall only one recursive call inside $!a(\tilde{x}).Q$ is possible, during a recursive call, thanks to Definition 14. We proceed by induction on $\tilde{x}\|^{\star}_{\tilde{T}}$. If $\tilde{x}\|^{\star}_{\tilde{T}}$ is minimal no recursion is possible. Otherwise, Rule (Serv) controls the recursive call is done on $\tilde{x}'$ with $\tilde{x}'\|^{\star}_{\tilde{T}}$ strictly smaller and $\tilde{x}'\|_{\tilde{T}}$ identical. As a result, in any consequence output $\overline{b}\langle \tilde{v}' \rangle$ of $!a(\tilde{v})$ either v_i does not depend on the recursive call and $v_i \le M + x_i$ and we conclude, or v_i depends on the recursive call result, induction hypothesis gives $v_i \le M + \mathcal{F}_1(|\tilde{x}'\|^{\star}_{\tilde{T}}|) + |\tilde{x}\|_{\tilde{T}}|$ which is $\le M + M * |\tilde{x}'\|^{\star}_{\tilde{T}}| + |\tilde{x}\|_{\tilde{T}}|$. We notice that $M + M * |\tilde{x}'\|^{\star}_{\tilde{T}}|$ is smaller than $\mathcal{F}_1(|\tilde{x}\|^{\star}_{\tilde{T}})|$ and conclude.

Case (2): Otherwise we examine the computation from a and consider the sequence of output consequences of a: there exists an enumeration E as defined above, of outputs (and their sent request names) inside $!c(\tilde{x}).P'$ which corresponds to this sequence. We also obtain polynomials defined above for each output and each sequence prefix. We proceed by induction on E and by recurrence on $\tilde{x}$ to prove that (i) if e_i is a safe integer in a recursive output or a linear output on x_i at step k in E then $e_i \leq N + \mathcal{F}_{E|_k}(|\tilde{x}\|^{\star}_{\tilde{T}}|) + |\tilde{x}\|_{\tilde{T}}| + \mathcal{F}_N(|\tilde{x}'\|^{\star}_{\tilde{T}}|)$; and (ii) if e_i is an unsafe integer in a recursive output or a x_i output at step k in E then $e_i \leq M + \mathcal{F}_{E|_k}(|\tilde{x}\|^{\star}_{\tilde{T}}|) + |\tilde{x}\|_{\tilde{T}}|$.

Case (2-1): Consider the first element $\overline{u}\langle\tilde{e}\rangle$ instance message types $\tilde{U}$. If it is guarded by a controlled channel input, we obtain a contradiction: this output is never played in any computation. Indeed, the type system forces the unsafe channel to be created locally and be passed in a replicated output; by minimality, there is no such output and the unsafe channel is never passed. If $\overline{u}\langle\tilde{w}\rangle$ is not guarded by any controlled channel, the elements of $\tilde{e}\|^{\star}_{\tilde{U}}$ are integers expressions of elements of $\tilde{x}\|^{\star}_{\tilde{T}}$ and $\tilde{e}\|_{\tilde{T}}$ are integers expressions of elements $\tilde{x}\|^{\star}_{\tilde{T}}$. Thus $e_i \leq M + M * |\tilde{x}\|^{\star}_{\tilde{T}}|$. We conclude.

Case (2-2): Consider an output $\overline{u}\langle\tilde{e}\rangle$ in enumeration E appearing after E'. Suppose it is guarded by a controlled input $d(\tilde{z})$ which is greater in the ordering defined above: contradiction, this output is never played in any computation: to do so it needs $d(\tilde{z})$ to be played first, but d is created locally and there is no output in the replication which sends d (otherwise that output would be under the output on u for the ordering). If $\overline{u}\langle\tilde{w}\rangle$ is only guarded by controlled channels $d_1(\tilde{z}_1), \dots, d_k(\tilde{z}_k)$ lower in the ordering, it means the elements of $\tilde{e}$ are integers expressions of elements of $\tilde{x}\|^{\star}_{\tilde{T}}$ and elements of $\tilde{z}_i$. Note that no d_i is extruded (appearing in a transition $\theta.\overline{b}\langle\dots, d_i, \dots\rangle$), because of the Definition 18, as a result, all d_i are sent on outputs played in replicated communications.

Case (2-2-1): Suppose e_i is safe. Then it is an integer expression of either $\tilde{x}\|^{\star}_{\tilde{T}}$ (because of subtyping), $\tilde{x}\|_{\tilde{T}}$ (and we conclude), or an integer expression of one $\tilde{z}_i$. We discuss the output sending d_i:

1. if d_i is sent by the recursive call $\overline{c}\langle\tilde{x}'\rangle$. We use the induction hypothesis on $\tilde{x}'$ and obtain $z_{i,j} \leq N + \mathcal{F}_N(|\tilde{x}'\|^{\star}_{\tilde{T}}|) + |\tilde{x}'\|_{\tilde{T}}|$. We conclude.
2. If d_i is sent by a safe auxiliary call $\overline{c'}\langle\tilde{w}\rangle$ given step k in the enumeration, we use the induction hypothesis on $E|_k$ and obtain w_i safe implies $w_i \leq N + \mathcal{F}_{E|_{k-1}}(|\tilde{x}\|^{\star}_{\tilde{T}}|) + |\tilde{x}\|_{\tilde{T}}| + \mathcal{F}_N(|\tilde{x}'\|^{\star}_{|})$, and w_i unsafe implies $w_i \leq \mathcal{F}_{E|_{k-1}}(\tilde{x}\|^{\star}_{\tilde{T}}) + \tilde{x}\|_{\tilde{T}}$. We examine our type system rules and deduce that the matching output of $d_i(\tilde{z}_i)$ is a causal dependency of the replicated input matching $\overline{c'}\langle\tilde{w}\rangle$. We use induction hypothesis on the level of c' (lower than N) to obtain that the $z_{i,j} \leq N + \mathcal{F}_{E|_k}(|\tilde{x}\|^{\star}_{\tilde{T}}|) + |\tilde{x}\|_{\tilde{T}}| + \mathcal{F}_N(|\tilde{x}'\|^{\star}_{\tilde{T}}|)$
3. If d_i is sent by an unsafe auxiliary call $\overline{c'}\langle\tilde{w}\rangle$, which is possible because of subtyping, we proceed as in the previous case (we do not obtain the recursive term).

Case (2-2-2): Suppose e_i is unsafe. Then if an integer expression of either $\tilde{x}\|^{\star}_{\tilde{T}}$, (and we conclude), or an integer expression of one *unsafe* $\tilde{z}_i$. The output sending d_i can only be an output at a smaller level. We proceed as above to obtain that $z_{i,j} \leq N + \mathcal{F}_{E|_k}(|\tilde{x}\|^{\star}_{\tilde{T}}|) + |\tilde{x}\|_{\tilde{T}}|$.

We obtain that all e_i in outputs abiding to the hypothesis are such that $e_i \leq N + \mathcal{F}_E(|\tilde{x}\|^{\star}_{\tilde{T}}|) + |\tilde{x}\|_{\tilde{T}}| + \mathcal{F}^E(|\tilde{x}'\|^{\star}_{\tilde{T}}|) \leq \mathcal{F}_N(|\tilde{x}\|^{\star}_{\tilde{T}}|) + |\tilde{x}\|_{\tilde{T}}|$. As b instantiates such a name u we conclude. □

Stating soundness. Thanks to Lemma 28, a bound on depending transitions is obtained, allowing us to state the soundness theorem.

Theorem 29 *(Soundness). If P is sound then P is bound by a polynomial.*

Proof. By induction on levels N, by induction on $\tilde{v}\|^{\star}_{\tilde{T}}$ used in P, we prove "for every typable computation $C = (l_k, \Gamma_{k+1}, P_{k+1})_{k \in N \subseteq \mathsf{nat}}$ from P, for any m, if $\Gamma_m \vdash_{N_m} P_m$, $!a(\tilde{v}) \in l_m$ and $\Gamma_m(a) = (\tilde{T})_{N'}$ with $N' \leq N$ then $\#(\mathbf{D}(l_m)_S) \leq \mathcal{F}_N(|\tilde{v}\|^{\star}_{\tilde{T}}|)$" (considering recursive arguments is enough).

Case (1): Suppose an initial level 1 and a typable computation from P $C = (l_k, \Gamma_{k+1}, P_{k+1})_{k \in N \subseteq \mathsf{nat}})$. Our type system and Theorem 19 ensure that any continuation Q of a replicated input prefix $!a(\tilde{x}).Q$ with $a : (\tilde{T})_1$ inside P_m contains at most one replicated output prefix $\overline{a}\langle\tilde{e}\rangle$, that this output prefix is on level 1 with strictly smaller recursive arguments $\tilde{e}$. Thus if $!a(\tilde{v}) \in l_m$ at path θ, the depending set given by Lemma 22 is $l_m \cup \bigcup_{!a(v) \subseteq_{\mathsf{d}} l_k} \{l_k\}$. Lemma 23 ensures that $\#(\{!a(v) \subseteq_{\mathsf{d}} l_k\})$ is bound by a constant C depending on P alone. We examine the content of $!a(v) \subseteq_{\mathsf{d}} l_k$ contains linear inputs, outputs and at most one replicated communication. Causality relation ensures that the size of union of the depending sets of linear inputs and outputs is less than C. We pose $\mathcal{F}_N$ the solution of $\mathcal{F}_N(X) = \mathcal{F}_N(X-1) + C$; using Rule (Serv), as $\tilde{e}\|^{\star}_{\tilde{T}} < \tilde{x}\|^{\star}_{\tilde{T}}$ we conclude.

1. If $\tilde{v}\|^{\star}_{\tilde{T}} = (0, \dots, 0)$, then no such replicated communication exists, as it requires an output on strictly smaller recursive arguments. We conclude.

2. Otherwise, if there is l_p containing output $\overline{a}\langle\tilde{v}'\rangle$ with $\tilde{v}'\|^{\star}_{\tilde{T}} < \tilde{v}\|^{\star}_{\tilde{T}}$ we use the induction hypothesis to obtain that $\#(\mathbf{D}(l_p)) \leq \mathcal{F}_N(\tilde{v}'\|^{\star}_{\tilde{T}})$ and we conclude.

Case (2): If N is not minimal, we apply the same reasoning with the difference that $\{!a(v) \subseteq_{\mathrm{d}} l_k\}$ also contains some linear inputs, some linear outputs, at most one replicated communication on level N and several replicated communications l_{r_j} which contain outputs $\overline{b_j}\langle\tilde{v}_j\rangle$ of type $\overline{b_j}\langle\tilde{v}_j\rangle$ $N_j < N$. We use Lemma 28 to obtain that $\tilde{v}_j\|^{\star}_{\tilde{T}_j} \leq \mathcal{F}_{N_j}(|\tilde{x}\|^{\star}_{\tilde{T}}|)$. We use the induction hypothesis to get that $\#(\mathbf{D}(l_{r_j})) \leq \mathcal{F}_{N_j}(|\tilde{v}_j\|^{\star}_{\tilde{T}_j}|)$ and we get $\#(\mathbf{D}(l_{r_j})) \leq \mathcal{F}_{N_j}(\mathcal{F}_{N_j}(|\tilde{x}\|^{\star}_{\tilde{T}}|))$. We conclude as in the previous case, by computing a solution of an equation $\mathcal{F}_N(X) = \mathcal{F}_N(X-1) + \sum_j Q_j(X) + C$ with Q_j polynomials. Hence the solution is polynomial. □

6.3. Completeness

Completeness (Theorem 31) is stated w.r.t. the set of recursive polytime functions (see Bellantoni and Cook [7] for instance): we can compute all recursive polytime functions with sound processes. Proof is done by building a process from the polytime characterisation of Bellantoni and Cook [7].

Definition 30 *(Computing function).* We say that process P *computes function* $F : \mathbb{N}^q \to \mathbb{N}$ at address a, whenever $P \xrightarrow{a(\tilde{n},c)} P'$, there is a computation $(l_k, P_k)_{k \leq m}$ from P', $l_m = \overline{c}\langle F(\tilde{n})\rangle$. We say P *computes* F if there exists a name a s.t. P computes function F at address a.

Theorem 31 *(Completeness). If F is a function computable in polynomial time, then there exists a sound process P which computes F.*

Proof. We have to build processes for the base functions, composition and recursion and prove they abide to our type discipline.

Base functions can be directly expressed through expressions appearing in message positions inside continuation. For instance, in the continuation P of $!a(x_1, \ldots, x_n, u).P$, one can find an output with message $\langle x_3, (x_1+1)+1, 3, x_1-1\rangle$.

Conditional can be expressed through the guarded choice structure, as in
$!a(a, b_1, b_2, c).[a \bmod 2 = 0] \ldots b_1 \cdots + [a \bmod 2 \neq 0] \ldots b_1 \ldots$

Predicative recursion is represented by

$$\begin{aligned}&!f(x_1,\ldots,x_n,c).[x_1=0]\overline{g}\langle x_1,\ldots,x_n,c\rangle + [x_1\neq 0](\boldsymbol{\nu} r_1,r_2)\ (\overline{f}\langle x_1-1,\ldots,x_n,r_1\rangle\\ &\quad \mid\ r_1(res_1).\overline{h}\langle x_1,\ldots,x_n,res_1,r_2\rangle\ \mid\ r_2(res_2).\overline{c}\langle res_2\rangle)\end{aligned}.$$

Here, the type system requires x_1 to be unsafe and allows $x_2, \ldots, x_n$ to be of any kind. res_1, result of the recursive call is received on a linear channel r_1 and, as a result of the type system, is passed to h as a safe argument.

Safe composition is represented by:
$!f(x,s,c).(\boldsymbol{\nu} r_1,r_2,r)(\overline{h_1}\langle x,s,r_1\rangle\ \mid\ \overline{h_2}\langle x,r_2\rangle\ \mid\ r_1(res_1).r_2(res_2).\overline{g}\langle res_1,res_2,r\rangle\ \mid\ r(res).\overline{c}\langle res\rangle)$

Here f receives a recursive argument x and a safe argument s. Two functions h_1 and h_2 at level lower than the one of f are called, one using only unsafe arguments and another one using both. The type system ensures that r_1 is safe and r_2 is unsafe and that the first argument of g is safe (that is, g will not use res_1 to do recursion). Moreover, these definitions respect Definition 14. □

Remark. In the original ICC article [7], the authors use binary integers. In our paper, we use unary integers. Although this does not affect the soundness result (as the safe-unsafe division is enough to prevent non-polynomial complexity), binary and unary integers yield different completeness results in the sequential case, and this difference should be taken into account when addressing completeness results for our system.

Moreover, the class of functions represented by typable processes might be related to non-deterministic complexity classes, as our system allows different definitions of the same function (i.e. two different replicated inputs on the same channel f). At each recursive call, one definition has to be selected, yielding a non-deterministic complexity more powerful than a comparable sequential system.

Proposition 32 *(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 14. The use of simple processes is an alternative to the information flow analysis.

Definition 33 *(Simple process).* We say typable P is *simple* whenever every replication in P has the form:

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

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 14. 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 34 states that the flow analysis is not needed on simple processes.

Theorem 34 *(Simple soundness and completeness).* (1) *If P is simple and typable, then* sound(P)*; and* (2) *If F is a function computable in polynomial time, then there exists a simple P which computes F such that* sound(P).

Proof. By the same argument presented in Demangeon [17], the decision algorithm first decides the usage relation (which names and expression must have same type) in linear time. Then a termination constraint graph is build in linear time, yielding a level affectation if possible (otherwise process is not typable). The next step is identifying recursion position and it is done by inspecting service definitions from the lower level upward. An integer received by a service is given type $\mathsf{nat}_\star$ if it is an unsafe argument, or if it is used in a $\mathsf{nat}_\star$ position in an auxiliary call. Other names are given type nat. Afterwards, linear channels are typed. According to the way the inputs on these channels is done, they might be given unsafe channel type. If so, we check that Rule (UOut) can be applied when they are extruded. Then the process can be typed inductively according to the results of the previous passes. Unreplicated part is typed at level ∞. Choices not directed by the syntax consist of the choices of rules to type calls (it is settled by previous analysis).

The origin of all integers appearing in output messages inside replications is, by definition of simple processes, either the initial replicated prefix or an input d_i. In the latter case, a structure of simple processes ensures d_i abides to the clauses of Definition 14.

A rapid analysis of the terms present in proof of Theorem 31 shows that all these terms abide to the general scheme of simple process. For instance for safe composition we write:

$!f(x, s, c).(\boldsymbol{\nu} r_1, r_2, r)(\overline{h_1}\langle x, s, r_1\rangle \mid r_1(res_1).(\overline{h_2}\langle x, r_2\rangle \mid r_2(res_2).(\overline{g}\langle res_1, res_2, r\rangle \mid r(res).\overline{c}\langle res\rangle)))$ □

On sequentiality of typed services. The distributed nature of the language used to describe services allows one to build networks where multiple parallel calls are treated simultaneously.

This differs from the sequential nature of the computations provided by the original language of recursive functions [7]. The sequentiality is broken at two levels: not only our framework allows several calls to the same service to be treated simultaneously (i.e. interleaved), but a single call to a service can generate different computations, as auxiliary and recursive calls are unordered.

Yet, most arithmetic examples presented in this paper impose a sequence on their subcalls. For instance, to compute the product of x and y, one has to compute recursively the product of $x-1$ and y first, then add to this result the value of y. These two operations are *ordered*: the call to *add* requires the result of the recursive call to *mult* to be known. The same constraint applies to the *cube* service: the first call to *mult* on (x, x) takes place before the second call to *mult* - because the second one uses the result of the first one as argument.

However, our system allows the typing of processes in which several subcalls are made in parallel, meaning it is able to type processes such as:

$$!s_1(x, y, r).(\nu c_1, c_2, c_3)(\overline{mult}\langle x, x, c_1\rangle \mid \overline{mult}\langle y, y, c_2\rangle \mid c_1(u).c_2(v).\overline{add}\langle u, v, c_3\rangle \mid c_3(z).\overline{r}\langle z\rangle)$$

Service s_1 computes $x^2 + y^2$, and the square subcomputations are done in parallel - using the *mult* channel of the product service. Once the results of these calls are retrieved, they are sent to the *add* channel of the sum service in order to compute the final result. As there is no recursive call in s, the (UOut) rule will allow the lifting of u to an unsafe types in order for it to be passed as a first argument to *add* (as in example $\boldsymbol{C}$ from Fig. 4):

$$!s_2(x, r).(\nu c_1, c_2)([x=0]\overline{r}\langle 0\rangle + [x \neq 0]\ \overline{s_2}\langle x-1, c_1\rangle \mid \overline{mult}\langle x, x, c_2\rangle \mid c_1(u).c_2(v).\overline{add}\langle v, y, c_3\rangle \mid c_3(z).\overline{r}\langle z\rangle)$$

Service s_2 computes $\Sigma_{i=0}^{x} i^2$ recursively. Here, the recursive call $\overline{s_2}$ is performed in parallel to the square computation called on *mult*. Typing in ensured by lifting to unsafe the result of the product (performed by a service of strictly smaller level than s_2).

Moreover, the concurrent nature of the pi-calculus allows one to define non-deterministic functions. The current restrictions of the control flow impose some strong constraints - for instance, one cannot wait for the result of a recursive call

$\overline{s}\langle x-1, y, c\rangle$ with two different subprocesses $c(x_1).P_1 \mid c(x_2).P_2$, as it violates the definition of controlled process. These constraints could be changed to accommodate some in-service concurrency, at the cost of additional technicalities.

Yet, the current system allows the definition of non-deterministic services by simply writing two definitions on the same channel. For instance:

$$!s_3(x,r).(\nu c_1, c_2, c_3)\ ([x=0]\overline{r}\langle 0\rangle + [x \neq 0]\ \overline{s_3}\langle x, c_1\rangle \mid \overline{f}\langle x, c_2\rangle \mid c_1(u).c_2(v).\overline{add}\langle v, u, c_3\rangle \mid c_3(z).\overline{r}\langle z\rangle)$$
$$!s_3(x,r).(\nu c_1, c_2, c_3)\ ([x=0]\overline{r}\langle 0\rangle + [x \neq 0]\ \overline{s_3}\langle x, c_1\rangle \mid \overline{g}\langle x, c_2\rangle \mid c_1(u).c_2(v).\overline{add}\langle v, u, c_3\rangle \mid c_3(z).\overline{r}\langle z\rangle)$$

defines two services with the same name s_3 and similar behaviours: they perform in parallel a recursive call on themselves and an auxiliary call on either f or g, and they return the sum of the two results.

As both services are listening concurrently on s_3, an initial call on this channel can trigger either of these two definitions. And when a recursive call is made, it triggers, in a non-deterministic way, one of the two definitions. Thus service s_3 computes $\Sigma_{i=0}^{x} h_i(i)$ with each h_i being either f or g.

7. Counter-examples

In this section, we present a series of examples (some of them reminiscent of examples of Fig. 4) rejected by our analysis, justifying one by one the different design choices.

7.1. Motivating the type system

In the following paragraphs, we justify the different constraints enforced by the process syntax in Section 2 and the type system in Section 3.

Well-formedness definition for integer expressions.

$$\boldsymbol{L_0} = !a(x).\overline{a}\langle x-1\rangle$$

Process L_0 does not abide to the well-formedness condition of integer expressions, as expression $x-1$ is not under a guard $[x \neq 0]$. Indeed, an input $a(0)$ would produce an expression $0-1$, which has no value in our semantics.

Levels and control of recursive call.

Using integer levels and controlling recursive outputs (through predicate "$\mathrm{out}_N(P;\Gamma) = \emptyset$ or $\mathrm{out}_N(P;\Gamma) = \{\overline{b}\langle\tilde{e}\rangle\}$" in rule (Serv)) ensures termination and prevent the following processes to be typed:

$$\boldsymbol{L_1} = !a.\overline{b} \mid !b.\overline{a}$$
$$\boldsymbol{L_2} = !a(x).\overline{a}\langle x+1\rangle$$

Process $\boldsymbol{L_1}$ describes a diverging behaviour between two names a and b: one output $\overline{a}$ can be traded for output $\overline{b}$ and the other way around. Without the use of levels, process $\boldsymbol{L_1}$ would be able to receive a call on a and then engage in an infinite sequence of internal transitions. Rule (Serv) ensures that each output in the continuation of a service definition has to be done on a smaller or equal level. As a appears in continuation of $!b$ and the other way around, the only way to type this process would be to give a and b the same level. Moreover, our type system allows recursive calls (calls on the same level as the service itself) but at most one in each service and only if there is an explicit decreasing in arguments. Here, there is no strict decreasing in the messages of $\overline{b}$ and $!a$ (as there is no message), thus the process is rejected.

Process $\boldsymbol{L_2}$ performs an infinite behaviour after receiving a call on name a: the message of the output on a is increasing each time the service is called. Our type system rejects this process as it compares parameter x and argument $x+1$ of the output (predicate $\mathrm{out}_N(P;\Gamma) = \{\overline{a}\langle x+1\rangle\}$ of Rule (Serv), with N being the level of a) and cannot ensure a strict decreasing (it is not the case that $x+1 < x$ according to the rules for expression comparison).

Unicity of recursive call.

The following examples explain how the different predicates enforced by rule (Serv) rule out exponential behaviour.

$$\boldsymbol{L_3} = !a(x).[x \neq 0](\overline{a}\langle x-1\rangle \mid \overline{a}\langle x-1\rangle)$$
$$\boldsymbol{L_4} = !a(x).[x \neq 0](\overline{a}\langle x-1\rangle \mid \overline{b}\langle x-1\rangle) \mid !b(y).\overline{a}\langle y-1\rangle$$

Process $\boldsymbol{L_3}$ performs two recursive calls with strictly smaller arguments. An initial input $a(N)$ generates $\Theta(2^N)$ transitions. Our type system rejects this process by ensuring that at most one recursive call is possible, thanks to the predicate $\mathrm{out}_N(P;\Gamma) = \emptyset$ or $\mathrm{out}_N(P;\Gamma) = \{\overline{b}\langle\tilde{e}\rangle\}$. In this case, $\mathrm{out}_N(P;\Gamma)$ is the pair $\{\overline{a}\langle x-1\rangle, \overline{a}\langle x-1\rangle\}$.

When trying to typecheck process $\boldsymbol{L_4}$, the presence of $\overline{a}$ in the b service definition, and the presence of $\overline{b}$ in the a service definition, force us to give the same level to a and b. Thus, the definition of service a contains two outputs on the level of a and rule (Serv) cannot be applied, hence the process is rejected (the multiset of outputs is a pair, and not a singleton). An input $a(N)$ lets $\boldsymbol{L_4}$ perform an exponential number of internal transitions (its growth is comparable to the Fibonnaci sequence).

Safety of integer arguments.

$$\boldsymbol{L_5} = \boldsymbol{A} \mid \boldsymbol{P} \mid !\mathit{fact}(x,r).[x=0]\overline{r}\langle 1\rangle + [x \neq 0](\nu c, d)(\overline{\mathit{fact}}\langle x-1, c\rangle \mid c(y).\overline{\mathit{mult}}\langle x, y, d\rangle \mid d(z).\overline{r}\langle z\rangle)$$

Process $\boldsymbol{L_5}$ is introduced above as process $\boldsymbol{F}$ and is used here to motivate how checking the predicativity of recursion (the fact that the result of a recursive call cannot appear in recursion position), which allows the type system of Section 3 to reject exponential behaviours. As explained in Section 3, process $\boldsymbol{L_5}$ computes the factorial function, and an input $\mathit{fact}(N)$ generates a $\Theta(N!)$ number of transitions. Our type system rejects this process, as rule (SOut) and the well-formedness condition on types force the channel c passed on the recursive call to have a safe channel type (nat), meaning that y is a safe integer of type nat. However, service *mult* requires both his arguments to be unsafe integers (as recursions are done on both of them). This prevents y to be used in the *mult* call.

Well-formedness definition for types. Well-formedness of types prevents the construction of type $T = (\mathsf{nat}_\star, (\mathsf{nat}_\star))_3$: indeed, it forces linear channel types appearing inside replicated name types to have *safe* types. As a consequence, we cannot give type T to *fact*, which would allow us to type process $\boldsymbol{L_5}$. Indeed, the result of the recursive call would be treated as unsafe and be passed without typing error to *mult*.

7.2. Breaking the flow analysis

In the following paragraphs, we justify the different constraints enforced by the flow analysis of Section 5.

Uncontrolled integer in results and answer channels. The following examples justify Rules 1. and 2.b in Definition 14, forcing integer expressions passed on calls and answer channels (name received through replicated inputs) to be controlled.

$$\boldsymbol{L_6} \;=\; \boldsymbol{A} \mid !\mathit{mult}(x,y,r).[x=0]\,\overline{r}\langle 0\rangle + [x\neq 0]\,(\boldsymbol{\nu} d_2)\,(\overline{\mathit{mult}}\langle x-1, y, d_1\rangle \mid d_1(\mathit{res}).\overline{\mathit{add}}\langle y, \mathit{res}, d_2\rangle \mid d_2(z).\overline{r}\langle z\rangle)$$

Process $\boldsymbol{L_6}$ (called $\boldsymbol{P'}$ in Fig. 4) is a copy of process $\boldsymbol{P}$ except name d_1 is not private: it means external inputs $d_1(N)$ can interfere with transitions. We explain in Section 5 how our type system rejects process $\boldsymbol{L_6}$. Rule 1. of Definition 14 requires *res*, received on d_1 and passed on a call to *add*, to be controlled, meaning that d_1 has to be restricted locally, which is not the case.

$$\boldsymbol{L'_6} \;=\; \boldsymbol{A} \mid !\mathit{mult}(x,y,r).\quad [x=0]\,\overline{r}\langle 0\rangle + [x\neq 0]\,(\nu d_1)\quad (\overline{\mathit{mult}}\langle x-1, y, d_1\rangle \mid d_1(\mathit{res}).\overline{\mathit{add}}\langle y, \mathit{res}, d_2\rangle \mid d_2(z).\overline{r}\langle z\rangle)$$

Process $\boldsymbol{L'_6}$ is a process similar to process $\boldsymbol{P}$ except name d_2 is not private. As a result, the process can receive $d_2(1000)$ and send 1000 on its answer channel r, producing a situation similar as the one described above. Rule 2.b of Definition 14 rejects the process, as the integer content z of $\overline{r}\langle z\rangle$ is not controlled.

$$\begin{aligned}\boldsymbol{L''_6} \;=\; \boldsymbol{A} \mid a(d_2).!\mathit{mult}(x,y,r).\quad [x=0]\,\overline{r}\langle 0\rangle + [x\neq 0]\,(\nu d_1)\quad &(\overline{\mathit{mult}}\langle x-1, y, d_1\rangle \mid d_1(\mathit{res}).\overline{\mathit{add}}\langle y, \mathit{res}, d_2\rangle\\ &\mid d_2(z).\overline{r}\langle z\rangle)\end{aligned}$$

Process $\boldsymbol{L''_6}$ is, again, a similar process except name d_2 is bound by prefix, violating Rule 14.2.b: name d_2 is not restricted inside the replication.

$$\begin{aligned}\boldsymbol{L'''_6} \;=\; \boldsymbol{A} \mid (\nu d_2)\,(!\mathit{mult}(x,y,r).\quad [x=0]\,\overline{r}\langle 0\rangle + [x\neq 0]\,(\nu d_1)\quad &(\overline{\mathit{mult}}\langle x-1, y, d_1\rangle \mid d_1(\mathit{res}).\overline{\mathit{add}}\langle y, \mathit{res}, d_2\rangle\\ &\mid d_2(z).\overline{r}\langle z\rangle) \mid b(x').\overline{d_2}\langle x'\rangle)\end{aligned}$$

Same reasoning applies to process $\boldsymbol{L'''_6}$: d_2 is restricted but *outside* of the replication, letting arbitrarily large integers received on b interfer with the computation flow.

Wrong usage of return channel. The following example justifies Rule 2.a of Definition 14, forcing answer channels not to be passed in messages.

$$\begin{aligned}\boldsymbol{L_7} \;=\; \boldsymbol{A} \mid !\mathit{mult}(x,y,r).\quad (\overline{b}\langle r\rangle \mid [x=0]\,\overline{r}\langle 0\rangle \quad &+[x\neq 0]\,(\nu d_1, d_2)(\overline{\mathit{mult}}\langle x-1, y, d_1\rangle \mid d_1(\mathit{res}).\overline{\mathit{add}}\langle y, \mathit{res}, d_2\rangle\\ &\mid d_2(z).\overline{r}\langle z\rangle) \mid b(x_1).c(x_2).\overline{x_1}\langle x_2\rangle\end{aligned}$$

Process $\boldsymbol{L_7}$ is similar to process $\boldsymbol{P}$, with the difference that the continuation of the service definition for *mult* contains an output $\overline{b}\langle r\rangle$ which sends name r to the outside. The name is received on an external process, which can receive any integer x_2 on c and pass it on r.

As a result, when receiving a request $!\mathit{mult}(3,3,r)$, name r can be sent on b, then external input $c(1000)$ can be performed, yielding an output $\overline{r}\langle 1000\rangle$.

As above, if we replace process $\boldsymbol{P}$ by process $\boldsymbol{L_7}$ in the definition of process $\boldsymbol{C}$, service offered on channel *cube* is no longer polytime: a request $!\mathit{cube}(3,r)$ causes a subrequest $\overline{\mathit{mult}}\langle 3,3,d\rangle$ which can return result 1000 on d, because of external input $c(1000)$. The second subrequest $\overline{\mathit{mult}}\langle 3,1000,d\rangle$ will cause more than 3000 transitions. As a result, there is no longer a bound of the number of transitions a request $!\mathit{cube}(3,r)$ causes, as an arbitrary large integer can be received on c. Our flow analysis prevents this behaviour from happening, as a channel received with a request cannot be sent, according to Rule 2.a of Definition 14.

8. List structures

Motivating examples. Our work demonstrates how complexity of message generation can be harnessed using services computing function over the integers. These are modelled and abstract more complex real-world services, performing operations on large data structure. In this section, we explain how the type system from Section 3 and the analysis from Section 5 can be extended to handle data structures explicitly, such as list expressions together with operation :: (construction). This allows us to analyse the following motivating examples.

$$\begin{aligned}
\boldsymbol{T} &= !treat(d, ans).([d = \epsilon]\,\overline{ans}\langle\epsilon\rangle + [d = hd :: tl](\boldsymbol{\nu} r_1, r_2)\ (\overline{treat}\langle tl, r_1\rangle \mid \overline{aux}\langle hd, r_2\rangle \mid r_1(x).r_2(y).\overline{ans}\langle y :: x\rangle)\\
\boldsymbol{Mu} &= !multi(d, z).[z \neq 0](\boldsymbol{\nu} r)\ (\overline{treat}\langle d, r\rangle \mid \overline{multi}\langle d, z-1\rangle)\\
\boldsymbol{Ch} &= !chain(d, z, c).[z = 0]\overline{c}\langle\epsilon\rangle + [z \neq 0](\boldsymbol{\nu} r_1, r_2)(\overline{chain}\langle d, z-1, r_1\rangle \mid r_1(x).\overline{treat}\langle x, r_2\rangle \mid r_2(y).\overline{c}\langle y\rangle)
\end{aligned}$$

Process $\boldsymbol{T}$ waits on channel *treat* for a list d and a channel *ans*. In a recursive way, a non-empty d is deconstructed in hd and tl and the service calls itself, through a fresh channel r_1, on tl, and calls an abstract auxiliary service *aux* (not represented) on hd. Eventually, it returns $y :: x$, a new list obtained from the answers of the two calls. Process $\boldsymbol{Mu}$ waits on channel *multi* for a request composed of a similar data structure and an integer z. Through recursive call, $\boldsymbol{Mu}$ will spawn z independent requests to $\boldsymbol{T}$. When considering messages exchanged, the complexity of $\boldsymbol{Mu}$ can be described as polytime on the condition that the service *aux* is itself polytime. Indeed, a request $!multi(D, N)$ causes a number $\Theta(|D| \times N)$ calls to *aux*. In similar service $\boldsymbol{Ch}$, the result of the recursive call to *chain*, once received, is used in recursion position in a call to *treat* yielding a potentially exponential behaviour: for instance, if the result of service *aux* is a structure of size 2, it means the result of *treat* is double the size of its parameter, and a request $chain(D, N, r)$ causes $\Theta(|D| \times 2^N)$ calls to *aux*. We want our system to be able to accept service *multi* and reject service *chain*.

We give type $(\mathtt{list}_\star[\alpha], (\mathtt{list}[\alpha]))_1$ to *treat* and type $(\mathtt{list}_\star[\alpha], \mathsf{nat})_2$ to *multi* with α the type of the data structure elements. The continuation of $\boldsymbol{T}$ is typable at level 1 as $(tl;) < (d;)$. The continuation of $\boldsymbol{Mu}$ is typable at level 2 as integer expressions in recursive call message $(d, n-1;)$ are strictly smaller than the ones in parameters $(d, n;)$. Note that both parameters of *multi* have to be unsafe, as one is used by the recursion in $\boldsymbol{Mu}$ and the other one is used to as a recursive argument when passed to $\boldsymbol{T}$. The example abides to the constraints of Definition 14: x is bound in $r_1(x).[\dots]$ and y in $r_2(y).[\dots]$ and both r_1 and r_2 are restricted locally, use only once in a subject position in input and used only once in an object position, inside calls to services *treat* and *aux*. And channel *ans* is used only once in a subject position, its content is controlled, and it is not passed to other channels.

Our system rejects $\boldsymbol{Ch}$: as in $\boldsymbol{Mu}$, typing rules force the first two parameters of *chain* to be unsafe as they are both used in recursion positions in $\boldsymbol{Ch}$ and $\boldsymbol{T}$. Channel r_1 is passed inside a recursive call, by rule (SOut), it has to be a safe linear channel, implying x is of type $\mathtt{list}[\alpha]$. Yet, x is used as first argument of *treat*, which, as explained above, is of type $(\mathtt{list}_\star[\alpha], (\mathtt{list}[\alpha]))_1$. Hence types are mismatched.

Adapting the type system. The syntax of list expressions is given by:

$$e, e' ::= \epsilon \mid a :: e$$

We add to the syntax of π-terms the decomposition through head and tail:

$$P, Q ::= \dots \mid [e = \epsilon]P + [e = u :: e']Q$$

The choice construct is a bit more complicated compared to $[x = 0]P + [x \neq 0]Q$ because it binds u and e' in the continuation Q. This binding has little effect on the semantics, it could be expressed through functions “head” and “tail”.

To adapt the type system we need to adapt the notions of controlled and uncontrolled to the list structures: informally, a controlled list would be a list which has been constructed from received list parameters, the use of external list services, and the construction operator. The latter is the counterpart of the successor function in integer computation; the difference being that we the element passed to the constructor to be controlled as well (otherwise, we'll add potentially arbitrary value inside the computation).

For the integer-level computation, one difference is that controlled integers can now come from the destruction of a list structure, and we will consider them controlled as long as the list they come from is controlled. Thus this notion is *deep* in the sense that when a superstructure is controlled, it means all its elements are.

For recursion instead of checking that a recursive call is made on $x - 1$, we need to check that it is made on the tail of the list, which can only be done by referring to it abstractly in the syntax, thanks to a “tail” variable appearing in the decomposition operator and in the recursive call. We update the $(\lhd)$ operator to reflect this change (a list e_1 is smaller than e_2 if it $e_1 = a_1 :: \cdots :: a_n :: e_2$ for some n). Input, output and choice rules are similar to the ones for the integer values (see Fig. 8).

Main results. We now state the main results of this section.

Lemma 35 *(Subject substitution). Soundness is preserved by substitution.*

Value Typing

$$\frac{e \in \mathbf{List}(\mathtt{A})}{\Gamma \vdash e : \mathbf{List}(\mathtt{A})} \qquad \frac{\Gamma \vdash e : \mathbf{List}(\mathtt{A})}{\Gamma \vdash \mathtt{hd}(e) : \mathtt{A}} \qquad \frac{\Gamma \vdash e : \mathbf{List}(\mathtt{A}) \quad \Gamma \vdash a : \mathtt{A}}{\Gamma \vdash a :: e : \mathbf{List}(\mathtt{A})}$$

Process Typing

$$\text{(Cond)}\,\frac{\Gamma \vdash_N P_i \quad (i=1,2) \quad \Gamma \vdash e : \mathbf{List}(\mathtt{A})}{\Gamma \vdash_N [e=\epsilon]P_1 + [e = a :: e']P_2} \qquad \text{(Serv)}\,\frac{\begin{array}{c}\Gamma \vdash_N P \quad \Gamma \vdash u : (\tilde{T})_N \quad \Gamma \vdash \tilde{y} : \tilde{T} \quad (1)\ \mathsf{out}_N(P;\Gamma) = \emptyset \text{ or;} \\ (2)\ \mathsf{out}_N(P;\Gamma) = \{\overline{b}\langle \tilde{e}\rangle\} \text{ with } \Gamma(b) = \Gamma(u) \text{ and } (\tilde{e} \triangleleft \tilde{T}) < (\tilde{y} \triangleleft \tilde{T})\end{array}}{\Gamma \vdash_\infty !u(\tilde{y}).P}$$

Fig. 8. Typing rules for lists.

Proof. The proof is the same as the one for Lemma 17, as the nature of the types of the value themselves are not used in proof arguments. □

Theorem 36 *(Subject transition). Soundness is preserved by transition.*

Proof. The proof is the same as the one for Theorem 19, using Lemma 35 instead of Lemma 17.

The notion of origin (Definition 26) is the main change between the two proofs, it has to be extended to accommodate

1. an expression can originate from the opening of a list through the $[e = a :: e']$ expression (which is now binding, as stated above): when a variable x is bound by such a construct (i.e. it appears in a), the origin of x is the origin of e.
2. a single expression can have multiple origins, when writing $\overline{a}\langle x :: y :: l\rangle$ the variable x, y and l can be bound at different places. To accommodate this change, the notion of "controlled origin" needs to be redefined by stating that an expression is controlled if every variable in the expression is controlled. □

Theorem 37 *(Soundness). If P is sound then P is bound by a polynomial.*

Proof. The main framework of the proof remains unchanged, as the ($\triangleleft$) comparison is enough to compare values and processes. As "loops" (recursion inside functions) on lists are triggered by consuming a list element, the construction of the resulting polynomial is similar. The additional computation resulting from the use of the value itself of the head of the list in another recursive function is taken into account by our layering approach to the construction of the polynomial. □

Theorem 38 *(Completeness). If F is a function computable in polynomial time, then there exists P sound which computes F.*

Proof. The proof is quite similar as the one for Theorem 31, as the difference between the translation of integer functions and list functions is little. The same modules (for composition, recursion, ...) are used, with list arguments instead of integer arguments, only the types differ from the modules in the proof of Theorem 31. □

9. Related works

9.1. Implicit computational complexity

ICC has been developed in many different contexts, e.g. using structural constraints [7,25] and type systems based on linear logics [6,24,5]. Our system develops ICC into a process algebra framework inspired by one of the classical systems [7]. 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 Degano and Priami [16]: instead of counting the computation steps as in Bellantoni and Cook [7], 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.

Our approach is parametric of the type system, and by extension, parametric of the implicit complexity system we use as its core: other systems such as Leivant-Marion characterisation [25] or safe recursion over arbitrary structure [9] could be considered and adapted.

9.2. Complexity in process algebra

We use a *level* system inspired by Deng and Sangiorgi [20], Demangeon et al. [18] and by *regions* from Amadio [2]. These systems decorate types with integer levels (or abstract regions), and check that no loop arises between them (inside

π-replications for Deng and Sangiorgi [20], Demangeon et al. [18] or λ-references for Amadio [2] and Madet [26]), 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 [2,26]. The second type system presented in Deng and Sangiorgi [20] 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. Work by Amadio and Dabrowski [3] studies complexity in a synchronous π-calculus, reminiscent of the Esterel model [8]: 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 as in Amadio and Dabrowski [3]. The work in Dal Lago et al. [13] defines a type system based on soft linear logics [24] 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 Lafont [24]. This differs from our treatment of the π-calculus with full constructs, i.e. mobility, channel-passing and replications. The work in Dal Lago and Di Giamberardino [12] 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 [10,29], they are able to lightly modify the session system in Caires and Pfenning [10] in order to capture polytime 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 [23,1] which use size annotation for control of recursion. In our case, the use of levels and expression comparison is more tailored for process verification.

Baillot and Ghyselen [5] prove complexity results for the π-calculus using sized types. Their goal is different as they target *parallel complexity* (studying the *span* of a process) and consider closed processes (no external communication).

Our work addressing complexity for data-structures services differs from the one in Marion [27]. The latter uses method based on prefixes while we consider a notion of list complexity (deconstructing the structure into a lower-order element "head" and a possibly empty continuation "tail").

9.3. *Causality*

The existing works about complexity of processes rely on counting the number of reductions a process performs w.r.t. its size [3,13,12]. 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 in Theorem 31. We restrict Degano and Priami's definition of causality [16], 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 Cristescu et al. [11], as long as one weakens them to break the causality links from linear communications, as explained in Section 4.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

No data was used for the research described in the article.

Acknowledgments

We deeply thank the reviewers' detailed and helpful comments. The second author is supported by EPSRC EP/T006544/1, EP/K011715/1, EP/K034413/1, EP/L00058X/1, EP/N027833/1, EP/N028201/1, EP/T014709/1, EP/V000462/1, EP/X015955/1 and VeTSS.

References

[1] A. Abel, B. Pientka, Well-founded recursion with copatterns and sized types, J. Funct. Program. 26 (2016) e2.
[2] R.M. Amadio, On stratified regions, in: APLAS, Springer, 2009, pp. 210–225.
[3] R.M. Amadio, F. Dabrowski, Feasible reactivity in a synchronous pi-calculus, in: PPDP, ACM, 2007, pp. 221–230.
[4] P. Baillot, A. Ghyselen, Combining linear logic and size types for implicit complexity, in: D.R. Ghica, A. Jung (Eds.), 27th EACSL Annual Conference on Computer Science Logic, CSL 2018, September 4-7, 2018, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Birmingham, UK, 2018, pp. 9:1–9:21.

[5] P. Baillot, A. Ghyselen, Types for complexity of parallel computation in pi-calculus, ACM Trans. Program. Lang. Syst. 44 (2022) 15, https://doi.org/10.1145/3495529.
[6] P. Baillot, K. Terui, Light types for polynomial time computation in lambda-calculus, in: LICS, IEEE, 2004, pp. 266–275.
[7] S. Bellantoni, S.A. Cook, A new recursion-theoretic characterization of the polytime functions, in: STOC, ACM, 1992, pp. 283–293.
[8] G. Berry, L. Cosserat, The ESTEREL synchronous programming language and its mathematical semantics, in: CONCUR, Springer, 1984, pp. 389–448.
[9] O. Bournez, F. Cucker, P.J. de Naurois, J. Marion, Computability over an arbitrary structure. Sequential and parallel polynomial time, in: FOSSACS, Springer, 2003, pp. 185–199.
[10] L. Caires, F. Pfenning, Session types as intuitionistic linear propositions, in: CONCUR, Springer, 2010, pp. 222–236.
[11] I. Cristescu, J. Krivine, D. Varacca, A compositional semantics for the reversible pi-calculus, in: LICS, IEEE, 2013, pp. 388–397.
[12] U. Dal Lago, P. Di Giamberardino, On session types and polynomial time, Math. Struct. Comput. Sci. 26 (2016) 1433–1458, https://doi.org/10.1017/S0960129514000632.
[13] U. Dal Lago, S. Martini, D. Sangiorgi, Light logics and higher-order processes, Math. Struct. Comput. Sci. 26 (2016) 969–992.
[14] P. Degano, F. Gadducci, C. Priami, A causal semantics for CCS via rewriting logic, Theor. Comput. Sci. 275 (2002) 259–282, https://doi.org/10.1016/S0304-3975(01)00165-7.
[15] P. Degano, F. Gadducci, C. Priami, Causality and replication in concurrent processes, in: PSI, 2003, pp. 307–318.
[16] P. Degano, C. Priami, Causality for mobile processes, in: ICALP, 1995, pp. 660–671.
[17] R. Demangeon, Termination of Distributed Systems, Ph.D. thesis, ENS Lyon / Università di Bologna, 2010.
[18] R. Demangeon, D. Hirschkoff, D. Sangiorgi, Termination in impure concurrent languages, in: CONCUR, Springer, 2010, pp. 328–342.
[19] R. Demangeon, N. Yoshida, Causal computational complexity of distributed processes, in: A. Dawar, E. Grädel (Eds.), Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018, ACM, 2018, pp. 344–353.
[20] Y. Deng, D. Sangiorgi, Ensuring termination by typability, Inf. Comput. 204 (2006) 1045–1082, https://doi.org/10.1016/j.ic.2006.03.002.
[21] E. Hainry, J. Marion, R. Péchoux, Type-based complexity analysis for fork processes, in: F. Pfenning (Ed.), Foundations of Software Science and Computation Structures - 16th International Conference, FOSSACS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings, Springer, 2013, pp. 305–320.
[22] K. Honda, M. Tokoro, An object calculus for asynchronous communication, in: ECOOP'91, 1991, pp. 133–147.
[23] J. Hughes, L. Pareto, A. Sabry, Proving the correctness of reactive systems using sized types, in: POPL, 1996, pp. 410–423.
[24] Y. Lafont, Soft linear logic and polynomial time, Theor. Comput. Sci. 318 (2004) 163–180.
[25] D. Leivant, J. Marion, Lambda calculus characterizations of poly-time, Fundam. Inform. 19 (1993) 167–184.
[26] A. Madet, A polynomial time λ-calculus with multithreading and side effects, in: D.D. Schreye, G. Janssens, A. King (Eds.), Principles and Practice of Declarative Programming, PPDP'12, Leuven, Belgium - September 19 - 21, 2012, ACM, 2012, pp. 55–66.
[27] J. Marion, A type system for complexity flow analysis, in: LICS, IEEE, 2011, pp. 123–132.
[28] R. Milner, J. Parrow, D. Walker, A calculus of mobile processes, I., Inf. Comput. 100 (1992) 1–40, https://doi.org/10.1016/0890-5401(92)90008-4.
[29] P. Wadler, Propositions as sessions, J. Fam. Psychol. 24 (2014) 384–418, https://doi.org/10.1017/S095679681400001X.