Types for Complexity of Parallel Computation in Pi-calculus

PATRICK BAILLOT and ALEXIS GHYSELEN, Univ Lyon, CNRS, ENS de Lyon, Universite Claude-Bernard Lyon 1, LIP, F-69342, Lyon Cedex 07, France

Type systems as a technique to analyse or control programs have been extensively studied for functional programming languages. In particular, some systems allow one to extract from a typing derivation a complexity bound on the program. We explore how to extend such results to parallel complexity in the setting of pi-calculus, considered as a communication-based model for parallel computation. Two notions of time complexity are given: the total computation time without parallelism (the work) and the computation time under maximal parallelism (the span). We define operational semantics to capture those two notions and present two type systems from which one can extract a complexity bound on a process. The type systems are inspired both by sized types and by input/output types, with additional temporal information about communications.

CCS Concepts: • **Theory of computation** → **Type structures**; *Process calculi*; • **Software and its engineering** → Software verification;

Additional Key Words and Phrases: Type systems, pi-calculus, process calculi, complexity analysis, implicit computational complexity, sized types

ACM Reference format:
Patrick Baillot and Alexis Ghyselen. 2022. Types for Complexity of Parallel Computation in Pi-calculus. *ACM Trans. Program. Lang. Syst.* 44, 3, Article 15 (July 2022), 50 pages.
https://doi.org/10.1145/3495529

1 INTRODUCTION

The problem of certifying time complexity bounds for programs is a challenging question, related to the problem of statically inferring time complexity, and it has been extensively studied in the setting of sequential programming languages. One particular approach to these questions is that of type systems, which offers the advantage of providing an analysis that is formally grounded, compositional, and modular. In the functional framework several rich type systems have been proposed, such that if a program can be assigned a type, then one can extract from the type derivation a complexity bound for its execution on any input (see, e.g., References [4, 9, 23–25, 28]). The type system itself thus provides a complexity certification procedure, and if a type inference algorithm is also provided one obtains a complexity inference procedure. This research area is also related to implicit computational complexity, which aims at providing type systems or static

This work was supported by the LABEX MILYON (Grant No. ANR-10-LABX-0070) of Universite de Lyon.
Authors' address: P. Baillot, CRIStAL, Universit de Lille, Bât. ESPRIT, Avenue Henri Poincaré, 59655 Villeneuve d'Ascq; email: patrick.baillot@univ-lille.fr; A. Ghyselen, DIAPASoN, ALMA MATER STUDIORUM - Università di Bologna - Via Zamboni, 33 - 40126 Bologna; email: alexis.ghyselen@unibo.it.

0164-0925/2022/07-ART15 $15.00
https://doi.org/10.1145/3495529

criteria to characterize some complexity classes within a programming language (see, e.g., References [6, 7, 16, 18, 21, 27, 36]) and which have sometimes in a second step inspired a complexity certification or inference procedure.

However, while the topic of complexity certification has been thoroughly investigated for sequential programs, both for space and time bounds, there only have been a few contributions in the settings of parallel programs and distributed systems. In these contexts, several notions of cost can be of interest to abstract the computation time. First, one can wish to know what the total cumulated computation time on all processors is during a program execution. This is called the *work* of the program. Second, one can wonder if an infinite number of processors were available, what would be the execution time of the program when it is maximally parallelized. This is called the *span* or *depth* of the program.

In Reference [26], the authors addressed the problem of analysing the time complexity of programs written in a parallel first-order functional language. In this language one can spawn computations in parallel and use the resulting values in the body of the program. This allows one to express a large bunch of classical parallel algorithms. Their approach is based on amortized complexity and builds on a line of work in the setting of sequential languages to define type systems, which allow one to derive bounds on the work and the span of the program. However, the language they are investigating does not allow communication between those computations in parallel. Our goal is to provide an approach to analyse the time complexity of programs written in a rich language for communication-based parallel computation, allowing the representation of several synchronization features. We use for that π-calculus, a process calculus that provides process creation, channel name creation and name-passing in communication [37, 38]. An alternative approach could be to use a language described with session types, as in References [12, 13]. We will discuss the expressivity of both languages in Section 5.3.

We want to propose methods that, given a parallel program written in π-calculus, allow one to derive upper bounds on its work and span. Let us mention that these notions are not only of theoretical interest. Some classical results provide upper bounds, expressed by means of the work (w) and span (s), on the evaluation time of a parallel program on a given number p of processors. For instance, such a program can be evaluated on a **shared-multiprocessor system (SMP)** with p processors in time $O(\max(w/p, s))$ (see, e.g., Reference [22]).

Our goal in this article is essentially fundamental and methodological, in the sense that we aim at proposing type systems that are general enough, well-behaved, and provide good complexity properties. We also show intuitively how to relate our type systems to already existing type inference procedures for sized types, as in Reference [4], but we do not focus on automatization of type inference in this article.

We want to be able to derive complexity bounds that are parametric in the size of inputs, for instance, which depend on the length of a list. For that it will be useful to have a language of types that can carry information about sizes, and for this reason we take inspiration from sized types [9, 29]. So data-types will be annotated with an index that will provide some information on the size of values. Our approach then follows the standard approach to typing in the π-calculus, namely, typing a channel by providing the types of the messages that can be sent or received through it. Actually a second ingredient will be necessary for us, input/output types. In this setting a channel is given a set of capabilities: it can be an input, an output, or have both input/output capabilities. This distinction between outputs and inputs is especially useful for subtyping, a very important notion in sized types as it gives more flexibility. Indeed, an input channel and an output channel do not have the same behaviour with regard to subtyping.

Outline of the article. First, we describe in Section 3 a π-calculus with an explicit `tick` construction; this allows one to specify several cost models, instead of only counting the number of

reduction steps. We then describe in Section 4 a definition of the work of a process, design a type system, and establish a soundness theorem: If a process is well-typed in this type system, then its type provides an expression that, for its execution on any input, bounds the work. We also provide some hints on a type inference algorithm for this type system. Then, in Section 5, we give a formal definition of parallel complexity (span) in π-calculus. Next, we design another type system and again establish a soundness theorem but for span. Afterwards, we describe in Section 6 an example of parallel algorithm that can be analysed by this type system for span: bitonic sort. Finally, in Section 7, we compare our notion of span with the causal complexity from the literature.

This article is an extended version of the conference paper [5]. With respect to this previous paper, we give two additional results—one toward type inference for the work type system (Section 4.6) and another relating our definition of span to the notion of causal complexity (Section 7). We also give more details on the proofs of lemmas and theorems stated in the conference paper, and more detailed examples to illustrate the notions.

Discussion. Note that even though one of the main usages of π-calculus is to specify and analyse concurrent systems, the present article does not aim at analysing the complexity of arbitrary π-calculus concurrent programs. Indeed, some typical examples of concurrent systems, like semaphores, will simply not be typable in the system for span (see Section 5.3) because of linearity conditions. As explained above, our interest here is instead focused on parallel computation expressed in the π-calculus, which can include some form of cooperative concurrency. We believe the analysis of complexity bounds for concurrent π-calculus is another challenging question, which we want to address in future work.

A comparison with related works will be done in Section 8.

2 THE PI-CALCULUS AND INFORMAL INTRODUCTION TO SIZED TYPES

In this work, we consider the π-calculus as a model of parallelism. The main points of π-calculus are that processes can be composed in parallel, communication between processes happens with the use of channels, and channel names can be created dynamically.

2.1 Syntax of Pi-calculus

We present here a classical syntax for the asynchronous π-calculus. More details about π-calculus and variants of the syntax can be found in Reference [38]. We define the sets of *variables*, *expressions*, and *processes* by the following grammar:

$$v := x, y, z \mid a, b, c \qquad e := v \mid \mathtt{0} \mid \mathtt{s}(e) \mid [] \mid e :: e',$$
$$P, Q := \mathbb{0} \mid (P \mid Q) \mid !a(\widetilde{v}).P \mid a(\widetilde{v}).P \mid \overline{a}\langle\widetilde{e}\rangle \mid (\nu a)P \mid \mathtt{tick}.P$$
$$\mid \mathtt{match}\ e\ \{\mathtt{0} \mapsto P;;\ \mathtt{s}(x) \mapsto Q\} \mid \mathtt{match}\ e\ \{[] \mapsto P;;\ x :: y \mapsto Q\}.$$

Variables x, y, z denote *base type variables*, they represent integers or lists. Variables a, b, c denote *channel names*. The notation $\widetilde{v}$ stands for a sequence of variables $v_1, v_2, \ldots, v_k$. In the same way, $\widetilde{e}$ is a sequence of expressions. This syntax relies on binders; for example, in $a(\widetilde{v}).P$, the variables in $\widetilde{v}$ are bound in P, and similarly in $\mathtt{match}\ e\ \{[] \mapsto P;;\ x :: y \mapsto Q\}$, the variables x and y are bound in Q. We work up to α-renaming (renaming of bound variables), and we write $P[\widetilde{v} := \widetilde{e}]$ to denote the substitution of the free variables $\widetilde{v}$ in P by $\widetilde{e}$.

Intuitively, $\mathbb{0}$ represents the empty process, $P \mid Q$ stands for the parallel composition of P and Q, $a(\widetilde{v}).P$ represents an input: it stands for the reception on the channel a of a tuple of values identified by the variables $\widetilde{v}$ in the continuation P. The process $!a(\widetilde{v}).P$ is a replicated version of $a(\widetilde{v})$; it behaves like an infinite number of $a(\widetilde{v})$ in parallel. Such a channel a in a replicated input

will often be called a *server*. The process $\overline{a}\langle\widetilde{e}\rangle$ represents an output: It sends a sequence of expressions on the channel a. A process $(\nu a)P$ dynamically creates a new channel name a and then proceeds as P. We also have classical pattern matching on data types, and finally, in $\mathtt{tick}.P$, the tick incurs an additional cost of one, and this process has the same semantic behaviour as P. This constructor is the source of time complexity in a program. It can represent different cost models and it is more general than only counting the number of reduction steps. For example, by adding a `tick` after each input, we can count the number of communications in a process. By adding it after each replicated input on a channel a, we can count the number of calls to a. If we want to count the number of reduction steps, then we can add a `tick` after each input and pattern matching.

2.2 Informal Introduction to the Semantics and Type Systems

As for the semantics, we design two semantics depending on the kind of complexity we want to consider. First, we consider the *work* of a process, corresponding to the total number of `tick` in a computation. The important rules are:

$$\frac{}{!a(v).P \mid \overline{a}\langle\widetilde{e}\rangle \to_0 !a(v).P \mid P[\widetilde{v} := \widetilde{e}]} \qquad \frac{}{a(v).P \mid \overline{a}\langle\widetilde{e}\rangle \to_0 P[\widetilde{v} := \widetilde{e}]} \qquad \frac{}{\mathtt{tick}.P \to_1 P}$$

where the subscript indicates the complexity cost of this reduction step. So removing a `tick` has complexity one, and the other reduction steps have complexity zero. With this, we can define the work as the sum of the subscripts in a reduction path. As the π-calculus is non-deterministic, we consider a worst-case analysis.

Second, we consider another complexity, the *span*, corresponding to maximum parallel complexity. Informally, it means that all computations that are put in parallel can be done simultaneously, as if we had an infinite number of processors. In practice, it means that any number of `tick` in parallel should be removed simultaneously. So, for example, $\mathtt{tick}.\mathtt{0} \mid \mathtt{tick}.\mathtt{0} \mid \mathtt{tick}.\mathtt{0}$ should have span complexity 1 (whereas its work is 3).

To define span formally, we introduce annotations in the π-calculus: a new constructor $n : P$, where n is an integer. Intuitively, $n : P$ represents the process P with n ticks before, or alternatively, a process P that would be ready after n units of time. The important rules are

$$\frac{}{n\colon !a(v).P \mid m : \overline{a}\langle\widetilde{e}\rangle \Rightarrow n\colon !a(v).P \mid \max(m,n) : P[\widetilde{v} := \widetilde{e}],}$$

$$\frac{}{n : a(v).P \mid m : \overline{a}\langle\widetilde{e}\rangle \Rightarrow \max(m,n) : P[\widetilde{v} := \widetilde{e}],} \qquad \frac{}{n : \mathtt{tick}.P \Rightarrow (n{+}1) : P.}$$

So intuitively, we have the same rules as before but with additional information in the annotations. When doing a communication between an input and an output, in order for the continuation $P[\widetilde{v} := \widetilde{e}]$ to be available, both the input and the output should be ready. As they take, respectively, n and m units of time to be ready, the continuation is available after $\max(m, n)$ units of time. As to the last rule, it expresses the fact that the annotations indeed count the number of `tick`. From this, the span of a process is defined as the maximal annotation seen in all reduction paths from $0 : P$.

Example 2.1. We illustrate those semantics on a toy example simulating the recursive calls of the Fibonacci function. This is described in Figure 1.

Intuitively, on an input n, the channel $toyfib$ behaves in such a way that if $n = \mathtt{0}$ or $n = 1$, then the computation stops (because of the empty process $\mathtt{0}$). Otherwise, the channel does two recursive calls to itself with value n–1 and n–2. In this example, the `tick` constructor is used to count the number of calls to the server. Let us describe one possible reduction for work complexity

```
P ≡ !toyfib(n). tick match(n) { 0 ↦ 0 ;; s(m) ↦ match(m) {
      0 ↦ 0 ;;
      s(p) ↦ toyfib⟨m⟩ | toyfib⟨p⟩
    }}
```

Fig. 1. Toy example simulating the Fibonnaci recursive calls.

when we do a call to this server with $n = 3$:

$$P \mid \overline{toyfib}\langle 3\rangle \to_0 P \mid \mathtt{tick.match}\ 3\ \{\mathtt{0} \mapsto \mathtt{0};;\ \mathtt{s}(m) \mapsto \cdots\} \to_1 P \mid \mathtt{match}\ 3\ \{\mathtt{0} \mapsto \mathtt{0};;\ \mathtt{s}(m) \mapsto \cdots\}$$
$$\to_0^* P \mid \overline{toyfib}\langle 2\rangle \mid \overline{toyfib}\langle 1\rangle \to_0^* P \mid \mathtt{tick.match}\ 2\ \cdots \mid \mathtt{tick.match}\ 1\ \cdots$$
$$\to_1^2 P \mid \mathtt{match}\ 2\ \cdots \mid \mathtt{match}\ 1\ \cdots \to_0^* P \mid \overline{toyfib}\langle 1\rangle \mid \overline{toyfib}\langle 0\rangle \mid \mathtt{0}$$
$$\to_0^* P \mid \mathtt{tick.match}\ 1\ \cdots \mid \mathtt{tick.match}\ \mathtt{0}\ \cdots \mid \mathtt{0} \to_1^2 P \mid \mathtt{match}\ 1\ \cdots \mid \mathtt{match}\ \mathtt{0}\ \cdots \mid \mathtt{0} \to_0^* P \mid \mathtt{0} \mid \mathtt{0} \mid \mathtt{0}.$$

All the other reduction sequences are similar, only the order can change. Overall, if we count the number of $\to_1$, then we obtain a work of 5, which indeed corresponds to the number of calls to the function in Fibonacci of 3.

As for the span, we again look at a reduction but with annotated processes:

$$0 : P \mid 0 : \overline{toyfib}\langle 3\rangle \Rightarrow 0 : P \mid 0 : \mathtt{tick.match}\ 3\ \{\mathtt{0} \mapsto \mathtt{0};;\ \mathtt{s}(m) \mapsto \cdots\} \Rightarrow 0 : P \mid 1 : \mathtt{match}\ 3\ \{\mathtt{0} \mapsto \mathtt{0};;\ \mathtt{s}(m) \mapsto \cdots\}$$
$$\Rightarrow 0 : P \mid 1 : \overline{toyfib}\langle 2\rangle \mid 1 : \overline{toyfib}\langle 1\rangle \Rightarrow^* 0 : P \mid 1 : \mathtt{tick.match}\ 2\ \cdots \mid 1 : \mathtt{tick.match}\ 1\ \cdots$$
$$\Rightarrow^* 0 : P \mid 2 : \mathtt{match}\ 2\ \cdots \mid 2 : \mathtt{match}\ 1\ \cdots \Rightarrow^* 0 : P \mid 2 : \overline{toyfib}\langle 1\rangle \mid 2 : \overline{toyfib}\langle 0\rangle \mid 2 : \mathtt{0}$$
$$\Rightarrow^* 0 : P \mid 2 : \mathtt{tick.match}\ 1\ \cdots \mid 2 : \mathtt{tick.match}\ \mathtt{0}\ \cdots \mid 2 : \mathtt{0} \Rightarrow^* 0 : P \mid 3 : \mathtt{match}\ 1\ \cdots \mid 3 : \mathtt{match}\ \mathtt{0}\ \cdots \mid 2 : \mathtt{0}$$
$$\Rightarrow^* 0 : P \mid 3 : \mathtt{0} \mid 3 : \mathtt{0} \mid 2 : \mathtt{0}.$$

Again, all the other reduction sequences are similar. The span is then 3, because the maximal annotation seen in this reduction is 3. And it indeed corresponds to the number of calls to the Fibonacci function under maximal parallelism, that we can also see as the depth of the tree of calls to the function.

To analyse the complexity of processes, we design two type systems, one for work and one for span. Because the complexity can depend on the size of some values, such as the $toyfib$ server whose complexity depends on the value of n, we use *sized types* to keep track of the sizes of values in a process. We use integer expressions, called *indices*, to represent intervals. So, for instance, $\mathrm{Nat}[I, J]$ represents integers between I and J, where I and J are indices. As we do not always know the actual size of an integer, we may use *index variables*. For example, in the process P, we do not know the actual value of n (as it depends on the context), so we will give it the type $\mathrm{Nat}[i, i]$, where i is an index variable representing the (unknown) size of n.

First, we explain the type system for work. A judgement has the shape $\varphi; \Phi; \Gamma \vdash_{\mathrm{w}} P \triangleleft K$ where φ is a set of index variables, Φ a sef of constraints that we do not detail here, Γ a typing context, P a process and K an index representing the work complexity of K. The goal is to obtain the following theorem: if $\varphi; \Phi; \Gamma \vdash_{\mathrm{w}} P \triangleleft K$, then K is a bound on the work of P. Some simple rules of this type system are

$$\text{(Par)}\ \frac{\varphi; \Phi; \Gamma \vdash_{\mathrm{w}} P \triangleleft K_1 \qquad \varphi; \Phi; \Gamma \vdash_{\mathrm{w}} Q \triangleleft K_2}{\varphi; \Phi; \Gamma \vdash_{\mathrm{w}} P \mid Q \triangleleft K_1 + K_2,} \qquad \text{(Tick)}\ \frac{\varphi; \Phi; \Gamma \vdash_{\mathrm{w}} P \triangleleft K}{\varphi; \Phi; \Gamma \vdash_{\mathrm{w}} \mathtt{tick}.P \triangleleft K{+}1.}$$

The (Par) rule expresses the fact that the work is summed over parallel composition, and the (Tick) rule expresses the fact that a `tick` increases complexity by 1.

Let us detail briefly this type system on Example 2.1. The $toyfib$ server would be given a type $\forall i.\mathsf{serv}^{F(i)}(\mathsf{Nat}[i,i])$, expressing that for any index variable i representing the size of the input, the complexity of a call to this server is $F(i)$, where $F(m)$ for any integer m is the function defined by

$$F(0) = F(1) = 1, \qquad F(m+2) = 1 + F(m+1) + F(m).$$

This equation exactly corresponds to the description of the number of calls to a Fibonacci function, and it can be obtained in the type system. Formally, we can derive the following judgement:

$$i; \cdot; toyfib : \forall i.\mathsf{serv}^{F(i)}(\mathsf{Nat}[i,i]), n : \mathsf{Nat}[i,i] \vdash_{\mathrm{w}} \mathtt{tick.match}\ n\ \{\mathtt{0} \mapsto \mathbb{0};\ \mathtt{s}(m) \mapsto \cdots\} \triangleleft F(i). \quad (1)$$

And from this, we are able to deduce, in particular,

$$\cdot; \cdot; toyfib : \forall i.\mathsf{serv}^{F(i)}(\mathsf{Nat}[i,i]) \vdash_{\mathrm{w}} P \mid \overline{toyfib}\langle 3\rangle \triangleleft F(3).$$

As $F(3) = 5$, it is indeed a precise bound on the work. This typing derivation makes a good use of this unknown index i. Indeed, in the case where $i \geq 2$, the typing derivation for Equation (1) reaches a point where it has to type the subprocess $\overline{toyfib}\langle m\rangle \mid \overline{toyfib}\langle p\rangle$. In this case, the type system relies on the type $\forall i.\mathsf{serv}^{F(i)}(\mathsf{Nat}[i,i])$ to say that, as m has size i–1, then the complexity of $\overline{toyfib}\langle m\rangle$ is $F(i–1)$ and similarly, the complexity of $\overline{toyfib}\langle p\rangle$ is $F(i–2)$, and we recover the equation on the function F described above.

Now, for span, we also design a similar type system with judgements of the shape $\varphi; \Phi; \Gamma \vdash_{\mathrm{s}} P \triangleleft K$, with again the property that if $\varphi; \Phi; \Gamma \vdash_{\mathrm{s}} P \triangleleft K$, then K is a bound on the span of P. Intuitively, this type system corresponds to the previous one with additional information about time in types that we do not detail in this section. Some simple rules are

$$\text{(Par)}\ \frac{\varphi; \Phi; \Gamma \vdash_{\mathrm{s}} P \triangleleft K_1 \qquad \varphi; \Phi; \Gamma \vdash_{\mathrm{s}} Q \triangleleft K_2}{\varphi; \Phi; \Gamma \vdash_{\mathrm{s}} P \mid Q \triangleleft \max(K_1, K_2),} \qquad \text{(Tick)}\ \frac{\varphi; \Phi; \langle\Gamma\rangle_{-1} \vdash_{\mathrm{s}} P \triangleleft K}{\varphi; \Phi; \Gamma \vdash_{\mathrm{s}} \mathtt{tick}.P \triangleleft K{+}1.}$$

This time, the complexity of parallel composition is taken as the maximum, and for the (Tick) rule, we need a new operator $\langle\Gamma\rangle_{-1}$ on context that intuitively corresponds to reducing all time information by one, since the `tick` makes time advance by one unit.

Similar to the work, we can derive the following judgement:

$$i; \cdot; toyfib : \forall i.\mathsf{serv}_0^{G(i)}(\mathsf{Nat}[i,i]), n : \mathsf{Nat}[i,i] \vdash_{\mathrm{s}} \mathtt{tick.match}\ n\ \{\mathtt{0} \mapsto \mathbb{0};\ \mathtt{s}(m) \mapsto \cdots\} \triangleleft G(i), \quad (2)$$

where 0 is a time information indicating that the server is immediately ready to receive, and $G(i)$ is the function defined by

$$G(0) = G(1) = 1, \qquad G(m+2) = 1 + \max(G(m+1), G(m)) = 1 + G(m+1) = m + 2.$$

As $G(3) = 3$, we again obtain a precise bound for span. Informally, we have the same main idea than before for work, but because of the rule for parallel composition, when considering the subprocess $\overline{toyfib}\langle m\rangle \mid \overline{toyfib}\langle p\rangle$, the complexity is the maximum and not the sum.

In practice, this time information needed in the type system for span induces some important differences between the type system for work and for span, but this will be described in the body of the article.

3 SEMANTICS OF PI-CALCULUS

Let us define formally the semantics for π-calculus in this section. We first define on processes a congruence relation $\equiv$: This is the least congruence relation closed under

$$P \mid \mathbb{0} \equiv P \qquad P \mid Q \equiv Q \mid P \qquad P \mid (Q \mid R) \equiv (P \mid Q) \mid R$$
$$(\nu a)(\nu b)P \equiv (\nu b)(\nu a)P \qquad (\nu a)(P \mid Q) \equiv (\nu a)P \mid Q \text{ (when } a \text{ is not free in } Q).$$

Note that the last rule can always be applied from right to left by α-renaming. Also, one can see that contrary to usual congruence relation for the π-calculus, we do not consider the rule for replicated input ($!P \equiv !P \mid P$) as it will be captured by the semantics, and α-conversion is not taken as an explicit rule in the congruence. By associativity, we will often write parallel composition for any number of processes and not only two. Another way to see this congruence relation is that, up to congruence, a process is entirely described by a set of channel names and a multiset of processes. Formally, we can give the following definition.

Definition 3.1 (Guarded Processes and Canonical Form). A process G is *guarded* if it has one of the following shapes:

$$G := !a(v).P \mid a(v).P \mid \overline{a}\langle\widetilde{e}\rangle \mid \mathtt{tick}.P$$
$$\mid \mathtt{match}\ e\ \{\mathtt{0} \mapsto P;;\ \mathtt{s}(x) \mapsto Q\} \mid \mathtt{match}\ e\ \{[] \mapsto P;;\ x :: y \mapsto Q\}.$$

We say that a process is in *canonical form* if it has the form $(\nu\widetilde{a})(G_1 \mid \cdots \mid G_n)$ with $G_1, \ldots, G_n$ guarded processes.

Formally, we now show that all processes have a somewhat unique canonical form, as in Reference [34], Definition 4.1.2.

Lemma 3.2 (Existence of Canonical Form). *For any process P, there is a Q in canonical form such that $P \equiv Q$.*

The proof is direct by induction on P.

To show the uniqueness of the canonical form, let us first introduce some notations. Recall that α-renaming is not a rule of $\equiv$. We define the set $\mathtt{name}$ of channel variables and the multiset $\mathtt{gp}$ of guarded processes by:

- $\mathtt{name}(\mathtt{0}) = \emptyset$ and $\mathtt{gp}(\mathtt{0}) = \emptyset$,
- $\mathtt{name}(P \mid Q) = \mathtt{name}(P) \coprod \mathtt{name}(Q)$ and $\mathtt{gp}(P \mid Q) = \mathtt{gp}(P) + \mathtt{gp}(Q)$,
- $\mathtt{name}(P) = \emptyset$ and $\mathtt{gp}(P) = [P]$, when P is guarded,
- $\mathtt{name}((\nu a)P) = \mathtt{name}(\mathtt{P}) \coprod \{a\}$ and $\mathtt{gp}((\nu a)P) = \mathtt{gp}(P)$,

where $\coprod$ denotes the usual disjoint union, $+$ denotes the usual union of multisets, and $[P]$ denotes the multiset corresponding to the singleton P with multiplicity 1. Then, we can easily show the following lemma by definition of the congruence relation.

Lemma 3.3. *If $P \equiv Q$, then $\mathtt{name}(P) = \mathtt{name}(Q)$ and if $\mathtt{gp}(P) = [P_1, \ldots, P_n]$ and $\mathtt{gp}(Q) = [Q_1, \ldots, Q_m]$, then $m = n$ and for some permutation $Q'_1, \ldots, Q'_n$ of $Q_1, \ldots, Q_n$, we have $P_i \equiv Q'_i$ for all i.*

From this lemma, we can easily deduce the uniqueness of canonical form:

Lemma 3.4 (Uniqueness of Canonical Form). *If*

$$(\nu\widetilde{a})(P_1 \mid \cdots \mid P_n) \equiv (\nu\widetilde{b})(Q_1 \mid \cdots \mid Q_m),$$

with $P_1, \ldots, P_n, Q_1, \ldots, Q_m$ guarded processes, then $m = n$ and $\widetilde{a}$ is a permutation of $\widetilde{b}$. Moreover, for some permutation $Q'_1, \ldots, Q'_n$ of $Q_1, \ldots, Q_n$, we have $P_i \equiv Q'_i$ for all i.

We now define the usual reduction relation for the π-calculus, which we denote $P \rightarrow_0 Q$. It is defined by the rules given in Figure 2. Remark that substitution should be well-defined to do some reduction steps: Channel names must be substituted by other channel names and base type variables can be substituted by any expression except channel names. However, when we will consider typed processes, this will always yield well-defined substitutions.

$$\frac{}{!a(v).P \mid \overline{a}\langle \widetilde{e}\rangle \to_0 !a(v).P \mid P[\widetilde{v} := \widetilde{e}]} \qquad \frac{}{a(v).P \mid \overline{a}\langle \widetilde{e}\rangle \to_0 P[\widetilde{v} := \widetilde{e}]}$$

$$\frac{}{\mathtt{match}\ \mathtt{0}\ \{\mathtt{0} \mapsto P;;\ \mathtt{s}(x) \mapsto Q\} \to_0 P} \qquad \frac{}{\mathtt{match}\ \mathtt{s}(e)\ \{\mathtt{0} \mapsto P;;\ \mathtt{s}(x) \mapsto Q\} \to_0 Q[x := e]}$$

$$\frac{}{\mathtt{match}\ []\ \{[] \mapsto P;;\ x :: y \mapsto Q\} \to_0 P} \qquad \frac{}{\mathtt{match}\ e :: e'\ \{[] \mapsto P;;\ x :: y \mapsto Q\} \to_0 Q[x, y := e, e']}$$

$$\frac{P \to_0 Q}{P \mid R \to_0 Q \mid R} \qquad \frac{P \to_0 Q}{(va)P \to_0 (va)Q} \qquad \frac{P \equiv P' \qquad P' \to_0 Q' \qquad Q' \equiv Q}{P \to_0 Q}$$

Fig. 2. Standard reduction rules.

```
!add(n,m,a). match(n) {
  0 ↦ ā⟨m⟩ ;;
  s(p) ↦ (νb) (add̄⟨p,m,b⟩ | b(q). ā⟨s(q)⟩)
}

!fib(n,a). tick. match(n) {
  0 ↦ ā⟨0⟩ ;;
  s(m) ↦ match(m) {
    0 ↦ ā⟨1⟩ ;;
    s(p) ↦ (νb)(νc) (fib̄⟨m,b⟩ | fib̄⟨p,c⟩ | c(x).b(y). add̄⟨x,y,a⟩)
}}
```

Fig. 3. The Fibonacci function.

For now, this relation cannot reduce a process of the form $\mathtt{tick}.P$. Therefore, we need to introduce a reduction rule for `tick`. From this semantics, we will define a reduction corresponding to total complexity (*work*) and design a type system for this notion of complexity. Then, we will define parallel complexity (*span*) by taking an expansion of the standard reduction.

Let us give an example of process that will be of interest in the following. We express how to encode a usual recursive function in π-calculus by describing the Fibonacci function. Contrary to the function of Example 2.1, we use replicated input with a return channel to send the final value.

Example 3.5 (Fibonacci). This representation of the Fibonacci function is described in Figure 3, where the actual process P corresponds to the parallel composition of those two servers add and fib.

This example is similar to Example 2.1, but the differences are that we compute and return the value of the Fibonacci function using the return channel a. And so, when we do the recursive calls, we first create new channel names (b and c) and use them to recover the values of the two recursive calls.

Discussion. For the sake of simplicity, we consider only integers and lists as base types in our calculus, but the results can be generalized to other algebraic data-types. Moreover, we consider very simple expressions with only base constructors, but we could enrich the set of expressions with functions (such as addition on integers) and obtain the same results. Indeed, the core of our results lies in the parallel constructs, and the set of expressions has little impact on the theoretical results.

$$\frac{}{\texttt{tick}.P \to_1 P} \qquad \frac{P \to_1 P'}{P \mid Q \to_1 P' \mid Q} \qquad \frac{Q \to_1 Q'}{P \mid Q \to_1 P \mid Q'} \qquad \frac{P \to_1 P'}{(va)P \to_1 (va)P'}$$

Fig. 4. Simple tick reduction rules.

4 WORK OF A PROCESS

4.1 Semantics for Work

We first describe a semantics for the work. The one-cost reduction relation $\to_1$ is defined in Figure 4. Intuitively, this reduction removes exactly one tick at the top-level.

Then from any process P, a sequence of reduction steps to Q is just a sequence of one-step reductions with $\to_0$ or $\to_1$, and the work complexity of this sequence is the number of $\to_1$ steps. In this article, we always consider the worst-case complexity so the work of a process is defined as the maximal complexity over all such sequences of reduction steps from this process.

Notice that with this semantics for work, adding `tick` in a process does not change its behaviour: We do not create or erase reduction paths, we only modify the complexity.

Example 4.1 (Fibonacci). If we consider the process P described in Example 3.5, then we can see that the work of $(va)(P \mid \overline{\text{fib}}\langle 10, a\rangle)$ is $F(10)$ where F is defined, as in Section 2, by

$$F(0) = 1, \qquad F(1) = 1, \qquad F(n+2) = 1{+}F(n+1){+}F(n).$$

4.2 Sized Input/Output Types

We now define a type system to bound the work of a process. The goal is to obtain a soundness result: if a process P is typable then we can derive an integer expression K such that the work of P is bounded by K.

Our type system relies on the definition of indices to keep track of the size of values in a process. Those indices were, for example, used in Reference [9] and are greatly inspired by Reference [29]. The main idea of those types in a sequential setting is to control recursive calls by ensuring a decreasing in the sizes.

Definition 4.2. The set of *indices*, representing natural numbers, is given by the following grammar:

$$I, J, K := i, j, k \mid f(I_1, \ldots, I_n).$$

The variables i, j, k are called index variables. The set of index variables is denoted $\mathcal{V}$. The symbol f is an element of a given set of function symbols containing addition and multiplication. We also assume that we have subtraction as a function symbol, with $n{-}m = 0$ when $m \geq n$. Each function symbol f of arity $\mathsf{ar}(f)$ comes with an interpretation $[\![f]\!] : \mathbb{N}^{\mathsf{ar}(f)} \to \mathbb{N}$.

Given an index valuation $\rho : \mathcal{V} \to \mathbb{N}$, we extend the interpretation of function symbols to indices, noted $[\![I]\!]_\rho$ with

$$[\![i]\!]_\rho = \rho(i), \qquad [\![f(I_1, \ldots, I_{\mathsf{ar}(f)})]\!]_\rho = [\![f]\!]([\![I_1]\!]_\rho, \ldots, [\![I_{\mathsf{ar}(f)}]\!]_\rho).$$

In an index I, the substitution of the occurrences of i in I by J is denoted $I\{J/i\}$.

Definition 4.3 (Constraints on Indices). Let $\varphi \subset \mathcal{V}$ be a set of index variables. A *constraint* C on φ is an expression with the shape $I \bowtie J$ where I and J are indices with free variables in φ and $\bowtie$ denotes a binary relation on integers. Usually, we take $\bowtie \in \{\leq, <, =, \neq\}$. Finite sets of constraints are denoted by Φ.

For a set $\varphi \subset \mathcal{V}$, we say that a valuation $\rho : \varphi \to \mathbb{N}$ *satisfies* a constraint $I \bowtie J$ on φ, noted $\rho \vDash I \bowtie J$ when $[\![I]\!]_\rho \bowtie [\![J]\!]_\rho$ holds. Similarly, $\rho \vDash \Phi$ holds when $\rho \vDash C$ for all $C \in \Phi$. Likewise, we note $\varphi; \Phi \vDash C$ when for all valuations ρ on φ such that $\rho \vDash \Phi$ we have $\rho \vDash C$. Remark that the order $\leq$ in a context $\varphi; \Phi$ is not total in general. For example, if $\varphi = \{i, j\}$, then $\varphi; \cdot \nvDash i \leq ij$ and $\varphi; \cdot \nvDash ij \leq i$. As usual in type systems, we will use a sequence notation to represent sets, thus the set $\{i, j, k\}$ could be represented, for example, by the sequence (i, j, k), where we can implicitly change the order in the sequence.

Definition 4.4. The set of *types* and *base types* are given by the following grammars:

$$T := \mathcal{B} \mid \mathsf{ch}(\widetilde{T}) \mid \mathtt{in}(\widetilde{T}) \mid \mathtt{out}(\widetilde{T}) \mid \forall \widetilde{i}.\mathtt{serv}^K(\widetilde{T}) \mid \forall \widetilde{i}.\mathtt{iserv}^K(\widetilde{T}) \mid \forall \widetilde{i}.\mathtt{oserv}^K(\widetilde{T}),$$

$$\mathcal{B} := \mathsf{Nat}[I, J] \mid \mathsf{List}[I, J](\mathcal{B}).$$

Intuitively, in the context $\varphi; \Phi$, an integer n of type $\mathsf{Nat}[I, J]$ must be such that $\varphi; \Phi \vDash I \leq n \leq J$. Likewise, a list of type $\mathsf{List}[I, J](\mathcal{B})$ must have a length between I and J. We may use $\mathsf{Nat}[I]$ to denote the type $\mathsf{Nat}[I, I]$ to gain some space, especially in examples of type derivations.

For channel types, we would like to allow for some flexibility by using a subtyping relation. For this reason, we choose to use input/output types [38]. Intuitively, in such a system, in addition to the type of expressions that can be sent and received through it, a channel is also given a set of *capabilities*: either it is both an input and output channel, or it has only one of those capabilities. This is especially useful for subtyping, as an input channel and an output channel do not behave in the same way with regard to subtyping, as we will explain in Section 4.3. Unlike in usual input/output types, in this work, we also distinguish two kinds of channels: the *simple channels* (that we will often call channels), and replicated channels (called *servers*).

The three different types for channels and servers correspond to the three different sets of capabilities. We note serv when the server has both capabilities, iserv when it only has input and oserv when it only has output. Then, for servers, we have additional information: there is a quantification over index variables, and the index K stands for the *complexity* of the process spawned by this server. A typical example could be a server taking as input a list and a channel, and sending to this channel the sorted list, in time $k \cdot n$ where n is the size of the list: $P = !a(x, b). \cdots \overline{b}\langle e \rangle$ where e represents at the end of the computation the list x sorted. Such a server name a could be given the type $\forall i.\mathtt{serv}^{k \cdot i}(\mathsf{List}[0, i](\mathcal{B}), \mathtt{out}(\mathsf{List}[0, i](\mathcal{B})))$. This type means that for all integers i, if given a list of size at most i and an output channel waiting for a list of size at most i, the process spawned by this server will stop at time at most $k \cdot i$. Those bounded index variables $\widetilde{i}$ are especially useful for replicated input: as a replicated input is meant to be used several times with different values, it is necessary to allow for this kind of polymorphism on indices. Moreover, if a replicated input is used to encode a recursion, with this polymorphism, we can take into account the different recursive calls with different values and different complexities. This was briefly explained in Section 2 and it will be formally explained in Example 4.5.

We can now present the type system for work. A judgement for an expression has the shape $\varphi; \Phi; \Gamma \vdash e : T$, where φ is a set of index variables, Φ a set of constraints, Γ a typing context of the shape $v_1 : T_1, \cdots v_m : T_m$, e is an expression and T is a type. The set of index variables φ is such that the free index variables in Φ, Γ and T are included in φ. Intuitively, this typing means that if the constraints in Φ are satisfied, then e has type T in the context Γ. In this work, almost all relations (such as, e.g., subtyping, typing relation) depend on a context $\varphi; \Phi$, and φ always describes the set of usable free index variables, and Φ a set of constraints on φ. Rules for expressions are given in Figure 5. We use the notation $\varphi; \Phi; \Gamma \vdash \widetilde{e} : \widetilde{T}$ for a sequence of typing judgements for expressions in the tuple $\widetilde{e}$. The rules are standard for a sized type system. We have the usual (Ax) rule, then

$$\text{(Ax)}\ \frac{v : T \in \Gamma}{\varphi; \Phi; \Gamma \vdash v : T} \qquad \text{(Zero)}\ \frac{}{\varphi; \Phi; \Gamma \vdash \mathtt{0} : \mathsf{Nat}[0,0]} \qquad \text{(Nil)}\ \frac{}{\varphi; \Phi; \Gamma \vdash [\,] : \mathsf{List}[0,0](\mathcal{B})}$$

$$\text{(Succ)}\ \frac{\varphi; \Phi; \Gamma \vdash e : \mathsf{Nat}[I,J]}{\varphi; \Phi; \Gamma \vdash \mathtt{s}(e) : \mathsf{Nat}[I{+}1, J{+}1]}$$

$$\text{(Cons)}\ \frac{\varphi; \Phi; \Gamma \vdash e : \mathcal{B} \qquad \varphi; \Phi; \Gamma \vdash e' : \mathsf{List}[I,J](\mathcal{B})}{\varphi; \Phi; \Gamma \vdash e :: e' : \mathsf{List}[I{+}1, J{+}1](\mathcal{B})}$$

$$\text{(Sub)}\ \frac{\varphi; \Phi; \Delta \vdash e : U \qquad \varphi; \Phi \vdash \Gamma \sqsubseteq \Delta \qquad \varphi; \Phi \vdash U \sqsubseteq T}{\varphi; \Phi; \Gamma \vdash e : T}$$

Fig. 5. Typing rules for expressions.

$$\text{(Pzero)}\ \frac{}{\varphi; \Phi; \Gamma \vdash \mathbb{0} \triangleleft 0} \qquad \text{(Nu)}\ \frac{\varphi; \Phi; \Gamma, a : T \vdash P \triangleleft K}{\varphi; \Phi; \Gamma \vdash (\nu a)P \triangleleft K}$$

$$\text{(Intpm)}\ \frac{\varphi; \Phi; \Gamma \vdash e : \mathsf{Nat}[I,J] \qquad \varphi; (\Phi, I \le 0); \Gamma \vdash P \triangleleft K \qquad \varphi; (\Phi, J \ge 1); \Gamma, x : \mathsf{Nat}[I{-}1, J{-}1] \vdash Q \triangleleft K}{\varphi; \Phi; \Gamma \vdash \mathtt{match}\ e\ \{\mathtt{0} \mapsto P;;\ \mathtt{s}(x) \mapsto Q\} \triangleleft K}$$

$$\text{(Listpm)}\ \frac{\begin{array}{c}\varphi; (\Phi, I \le 0); \Gamma \vdash P \triangleleft K \\ \varphi; \Phi; \Gamma \vdash e : \mathsf{List}[I,J](\mathcal{B}) \qquad \varphi; (\Phi, J \ge 1); \Gamma, x : \mathcal{B}, y : \mathsf{List}[I{-}1, J{-}1](\mathcal{B}) \vdash Q \triangleleft K\end{array}}{\varphi; \Phi; \Gamma \vdash \mathtt{match}\ e\ \{[\,] \mapsto P;;\ x :: y \mapsto Q\} \triangleleft K}$$

$$\text{(Sub)}\ \frac{\varphi; \Phi; \Delta \vdash P \triangleleft K' \qquad \varphi; \Phi \vdash \Gamma \sqsubseteq \Delta \qquad \varphi; \Phi \vDash K' \le K}{\varphi; \Phi; \Gamma \vdash P \triangleleft K}$$

(a) Common Typing Rules for Processes

$$\text{(Par)}\ \frac{\varphi; \Phi; \Gamma \vdash_{\mathrm{w}} P \triangleleft K_1 \qquad \varphi; \Phi; \Gamma \vdash_{\mathrm{w}} Q \triangleleft K_2}{\varphi; \Phi; \Gamma \vdash_{\mathrm{w}} P \mid Q \triangleleft K_1{+}K_2} \qquad \text{(Tick)}\ \frac{\varphi; \Phi; \Gamma \vdash_{\mathrm{w}} P \triangleleft K}{\varphi; \Phi; \Gamma \vdash_{\mathrm{w}} \mathtt{tick}.P \triangleleft K{+}1}$$

$$\text{(In)}\ \frac{\varphi; \Phi; \Gamma \vdash a : \mathtt{in}(\widetilde{T}) \qquad \varphi; \Phi; \Gamma, \widetilde{v} : \widetilde{T} \vdash_{\mathrm{w}} P \triangleleft K}{\varphi; \Phi; \Gamma \vdash_{\mathrm{w}} a(\widetilde{v}).P \triangleleft K} \qquad \text{(Out)}\ \frac{\varphi; \Phi; \Gamma \vdash a : \mathtt{out}(\widetilde{T}) \qquad \varphi; \Phi; \Gamma \vdash \widetilde{e} : \widetilde{T}}{\varphi; \Phi; \Gamma \vdash_{\mathrm{w}} \overline{a}\langle \widetilde{e} \rangle \triangleleft 0}$$

$$\text{(Iserv)}\ \frac{\varphi; \Phi; \Gamma \vdash a : \forall \widetilde{i}.\mathtt{iserv}^K(\widetilde{T}) \qquad (\varphi, \widetilde{i}); \Phi; \Gamma, \widetilde{v} : \widetilde{T} \vdash_{\mathrm{w}} P \triangleleft K}{\varphi; \Phi; \Gamma \vdash_{\mathrm{w}} !a(\widetilde{v}).P \triangleleft 0}$$

$$\text{(Oserv)}\ \frac{\varphi; \Phi; \Gamma \vdash a : \forall \widetilde{i}.\mathtt{oserv}^K(\widetilde{T}) \qquad \varphi; \Phi; \Gamma \vdash \widetilde{e} : \widetilde{T}\{\widetilde{J}/\widetilde{i}\}}{\varphi; \Phi; \Gamma \vdash_{\mathrm{w}} \overline{a}\langle \widetilde{e} \rangle \triangleleft K\{\widetilde{J}/\widetilde{i}\}}$$

(b) Work Typing Rules for Processes

Fig. 6. Typing rules for work.

both (Zero) and (Nil) rules indicate the values with size zero, and both (Succ) and (Cons) rules indicate that those constructors increase size by one. The (Sub) rule allows one to change the typing derivation according to the subtyping relation that will be defined in Section 4.3.

We now present the typing rules for processes. A judgement has the shape $\varphi; \Phi; \Gamma \vdash_{\mathrm{w}} P \triangleleft K$, where K is an index, which means that under the constraints Φ, in the context Γ, the process P is typable and its work complexity is bounded by K. The rules are described in Figures 6(a) and 6(b).

Figure 6(b) describes rules specific to the work, whereas rules in Figure 6(a) will be reused for the span. Thus, we use the notation $\vdash$ in Figure 6(a) instead of $\vdash_{\mathrm{w}}$ to express that they are not specific to work.

The rules can be seen as a combination of input/output typing rules with rules found in a size type system for functional programs. The common rules that are used for both the span and work type systems are the rules for simple constructors of π-calculus and for sizes in pattern matching. The (Pzero) rule states that the empty process is typable, and it has complexity zero. The (Nu) rule allows one to add a new name in the context, without changing the complexity. The pattern matching rule for integers (Intpm) is a standard rule for sized types [9]. It says that if an integer e is between I and J, then in the first branch of the pattern matching, we can assume that $I \geq 0$, since e is equal to 0 in this branch. Similarly, in the second branch, we can assume that $J \geq 1$, since e should be greater than 1. In this case, the predecessor x has a size between I–1 and J–1. The rule (Listpm) is similar. Then, we have again the usual (Sub) rule, which can also arbitrary increase the upper-bound on the complexity. This is especially important, because, if we look back at the (Intpm) rule, it considers that both branches must have the same complexity K, but in practice they may have different complexities K_1 and K_2. However, thanks to the (Sub) rule, we can always consider a greater upper bound, in particular $\max(K_1, K_2)$. Thus, even if the branches have different complexities, we can always make them equal by considering the maximum with the (Sub) rule.

Rules that are specific to the work type system concern important constructors in the π-calculus. The (Par) rule expresses that the work of a parallel composition is the sum of the work of the two suprocesses. The (Tick) rules expresses that the `tick` constructor increases complexity by 1. The (In) rule states that to type an input, the continuation P should be typable in a context where the variables $\widetilde{v}$ are given the types $\widetilde{T}$, and the complexity of this input is the complexity of its continuation. Dually, in the (Out) rule, the expressions that are sent on this channel must have the expected types. As the complexity of a synchronization is already taken into account in the rule (In), the complexity in the (Out) rule is zero.

Finally, we have the rules for servers. In the rule (Iserv), the continuation P must satisfy the assumption given in the typing $\forall\widetilde{i}.\mathtt{iserv}^K(\widetilde{T})$: with the new index variables $\widetilde{i}$, in a context where $\widetilde{v}$ are given the expected types $\widetilde{T}$, the complexity of P must be K. Contrary to the (In) rule, this time the complexity K is captured in the type, but the bottom judgment has complexity zero. This is because in this case, we do now want a replicated input to have a non-zero complexity. Indeed, by definition of the reduction relation $\rightarrow_0$, a replicated input is never modified through a reduction. Thus, if a process has a replicated input, then this replicated input is retained in the normal form (if it exists). As a consequence, a non-zero complexity on an replicated input would always be an imprecise estimation of the complexity. Moreover, the complexity of a call to a server depends on the instantiation of $\widetilde{i}$ that is decided by an output to the server. That is why, in the last rule (Oserv), we have to find an instantiation $\widetilde{J}$ of the index variables $\widetilde{i}$ such that the types of expressions correspond to this instantiation. The complexity then depends on this instantiation. A typical example is if a server represents a function with quadratic complexity in the size of its argument, with type $\forall i.\mathtt{serv}^{i^2}(\mathrm{Nat}[i, i])$, and we call this server with the value 10, then we instantiate i by 10 and the overall complexity of this call to the server is $10^2 = 100$.

4.3 Subtyping

As explained before, with those types comes a notion of subtyping, to have some flexibility on bounds. Subtyping for base types is described by the rules of Figure 7(a). Both rules (Nat) and (List) express that we can always consider less precise bounds on the sizes.

$$\text{(Nat)}\ \frac{\varphi;\Phi \vDash I' \le I \qquad \varphi;\Phi \vDash J \le J'}{\varphi;\Phi \vdash \mathsf{Nat}[I,J] \sqsubseteq \mathsf{Nat}[I',J']} \qquad \text{(List)}\ \frac{\varphi;\Phi \vDash I' \le I \qquad \varphi;\Phi \vDash J \le J' \qquad \varphi;\Phi \vdash \mathcal{B} \sqsubseteq \mathcal{B}'}{\varphi;\Phi \vdash \mathsf{List}[I,J](\mathcal{B}) \sqsubseteq \mathsf{List}[I',J'](\mathcal{B}')}$$

(a) Subtyping Rules for Base Sized Types

$$\text{(I/O to In)}\ \frac{}{\varphi;\Phi \vdash \forall \widetilde{i}.\mathtt{serv}^K(\widetilde{T}) \sqsubseteq \forall \widetilde{i}.\mathtt{iserv}^K(\widetilde{T})} \qquad \text{(I/O to Out)}\ \frac{}{\varphi;\Phi \vdash \forall \widetilde{i}.\mathtt{serv}^K(\widetilde{T}) \sqsubseteq \forall \widetilde{i}.\mathtt{oserv}^K(\widetilde{T})}$$

$$\text{(I/O)}\ \frac{(\varphi,\widetilde{i});\Phi \vdash \widetilde{T} \sqsubseteq \widetilde{U} \qquad (\varphi,\widetilde{i});\Phi \vdash \widetilde{U} \sqsubseteq \widetilde{T} \qquad (\varphi,\widetilde{i});\Phi \vDash K = K'}{\varphi;\Phi \vdash \forall \widetilde{i}.\mathtt{serv}^K(\widetilde{T}) \sqsubseteq \forall \widetilde{i}.\mathtt{serv}^{K'}(\widetilde{U})}$$

$$\text{(In)}\ \frac{(\varphi,\widetilde{i});\Phi \vdash \widetilde{T} \sqsubseteq \widetilde{U} \qquad (\varphi,\widetilde{i});\Phi \vDash K' \le K}{\varphi;\Phi \vdash \forall \widetilde{i}.\mathtt{iserv}^K(\widetilde{T}) \sqsubseteq \forall \widetilde{i}.\mathtt{iserv}^{K'}(\widetilde{U})} \qquad \text{(Out)}\ \frac{(\varphi,\widetilde{i});\Phi \vdash \widetilde{U} \sqsubseteq \widetilde{T} \quad (\varphi,\widetilde{i});\Phi \vDash K \le K'}{\varphi;\Phi \vdash \forall \widetilde{i}.\mathtt{oserv}^K(\widetilde{T}) \sqsubseteq \forall \widetilde{i}.\mathtt{oserv}^{K'}(\widetilde{U})}$$

$$\text{(Trans)}\ \frac{\varphi;\Phi \vdash T \sqsubseteq T' \qquad \varphi;\Phi \vdash T' \sqsubseteq T''}{\varphi;\Phi \vdash T \sqsubseteq T''}$$

(b) Subtyping Rules for Server Types

Fig. 7. Subtyping.

Before explaining formally the rule for channel types, let us first introduce why input and output have different behaviours with regard to subtyping.

Let us consider, for the sake of the example, a type Nat for integers, and a type Real for real numbers, with Nat $\sqsubseteq$ Real. Subtyping can be understood in the following way: in any process where a value v is used as a real, then it is safe to feed this process with an integer. So, if $T \sqsubseteq U$, then in a process that uses a value of type U, it is safe to consider that this value has type T instead.

Let us apply the same reasoning to channels. Consider a channel a that receives a message, in a process $a(v).P$. We suppose that the process uses the channel a as a channel working with real numbers. Then, v is used as a real number in P, so it is safe to use a value v of type Nat instead. Overall, it is thus safe to assume this input is done on a value of type Nat. So, for an input channel, we should have : $\mathtt{in}(\mathsf{Nat}) \sqsubseteq \mathtt{in}(\mathsf{Real})$.

Now, consider a channel a that sends a message, in a process $\overline{a}\langle e\rangle$. We suppose that the process uses the channel a as a channel working with integers. Then, the message e sent on this channel is an integer. In particular, it is also a real number. So, it is safe to assume that all messages sent on this channel are real numbers, and we should have $\mathtt{out}(\mathsf{Real}) \sqsubseteq \mathtt{out}(\mathsf{Nat})$.

If we look at the variance of the subtyping relation, then for an input channel, we have covariance, and for an output channel, we have contravariance. Then, a channel with both capabilities should have invariance. That is why it is important to make a distinction between input channels and output channels.

Formally, subtyping for server types is described in Figure 7(b), the rules for channel types can be deduced from the rules for servers. The first two rules (I/O to In) and (I/O to Out) state that we can always remove a capability from a type. Then, the (I/O) rule states that a type with both capabilities should be invariant in its type. The (In) rule expresses covariance for types. As for complexity, if the continuation has a complexity bounded by K' then it is safe to bound it by K greater than K'. Dually, the (Out) rule expresses contravariance for types. As for complexity, it does not harm the typing to consider that all calls to a server take less time than expected. Finally, the (Trans) rule allows for transitivity of the subtyping relation. This rule is only useful to combine the rules (I/O to In) and (In) or (I/O to Out) and (Out).

We also extend subtyping to contexts, and we write $\Gamma \sqsubseteq \Delta$ when Γ and Δ have the same domain and for each variable $v : T \in \Gamma$ and $v : T' \in \Delta$, we have $T \sqsubseteq T'$.

4.4 Example

Example 4.5. Let us take again the process for Fibonacci described in Example 3.5. We first give a typing for add. As there is no tick in add, this typing only shows how input/output types and sized types behave. A simplified version of this typing is given in Figure 8(a), where we ignore the easy premises, in particular for axioms (with rule (Ax)). For example, in the first rule (Iserv), there should be a premise

$$\cdot;\cdot; add : \forall i,j.\mathtt{serv}^0(\mathsf{Nat}[i],\mathsf{Nat}[j],\mathtt{out}(\mathsf{Nat}[i{+}j])) \vdash add : \forall i,j.\mathtt{iserv}^0(\mathsf{Nat}[i],\mathsf{Nat}[j],\mathtt{out}(\mathsf{Nat}[i{+}j])).$$

But we ignore this as it is a simple derivation with the rules (Sub), (I/O to In) and (Ax). Also, recall that $\mathsf{Nat}[I]$ denotes the type $\mathsf{Nat}[I,I]$. There are several things to notice in this typing. First, we can see a use of polymorphism. With the first (Iserv) rule, the variables (n,m,a) are given the types $\widetilde{U}(i,j)$, but in the recursive call $\overline{add}\langle p,m,b\rangle$, those variables (p,m,b) have the the types $\widetilde{U}(i{-}1,j)$, expressing that the size of the first argument decreases by one. Moreover, constraints on indices are important to this typing. Indeed, to prove the following subtyping relations:

$$\varphi;(i \le 0) \vdash \mathsf{Nat}[j] \sqsubseteq \mathsf{Nat}[i{+}j], \qquad \varphi;(i \ge 1) \vdash \mathsf{Nat}[(i{-}1){+}j{+}1] \sqsubseteq \mathsf{Nat}[i{+}j],$$

we have to prove

$$\varphi;(i \le 0) \vDash j = i{+}j, \qquad \varphi;(i \ge 1) \vDash (i{-}1){+}j{+}1 = i{+}j.$$

The first equality relies on the fact that $i \le 0$ and so $i = 0$, and the second equality relies on $i \ge 1$, because, otherwise, we cannot deduce that $(i{-}1){+}1 = i$, since $0{-}1 = 0$ by definition. This behaviour of sizes in a type derivation is similar to the one in Reference [9] for functional programs.

We now describe the typing for the Fibonacci function. A simplified type derivation is given in Figure 8(b), and we detail here the important steps of this derivation. We use the Fibonacci function in indices, denoted $Fib(I)$, and we also use the function F defined in Example 4.1.

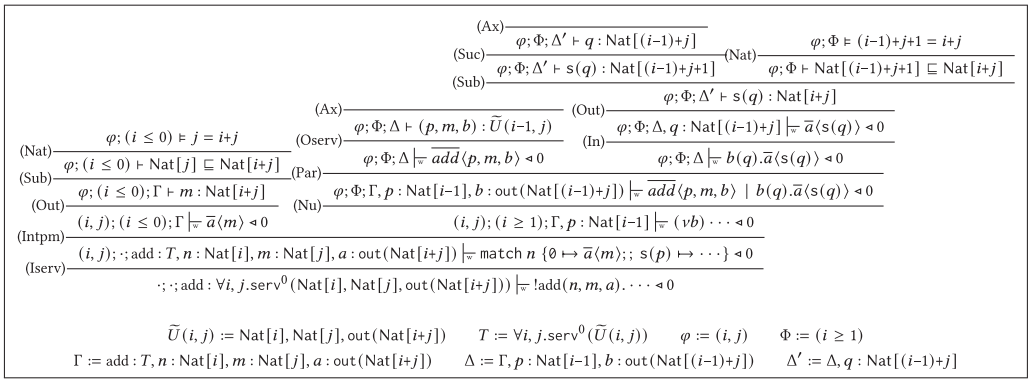
To begin with, we start with an (Iserv) rule, and we must prove that the continuation of the fib server has the expected complexity $F(i)$. As $F(i)$ is always greater than 1, we have $i;\cdot \vDash (F(i){-}1){+}1 = F(i)$, and thus after the (Tick) rule, we have to prove that the remaining subprocess has complexity $F(i){-}1$. We have then two pattern matchings. We ignore the case for zero and one but they are very similar to derivation for add: We can use the information $(i \le 0)$ or $(i \ge 1, i{-}1 \le 0)$ to show that the output to server has the expected type. In the main branch of pattern matching, we have the constraints $\Phi := (i \ge 1, i{-}1 \ge 1)$, which are equivalent to $(i \ge 2)$, stating that the size of n is greater than 2.

Then, an important step is for the judgement

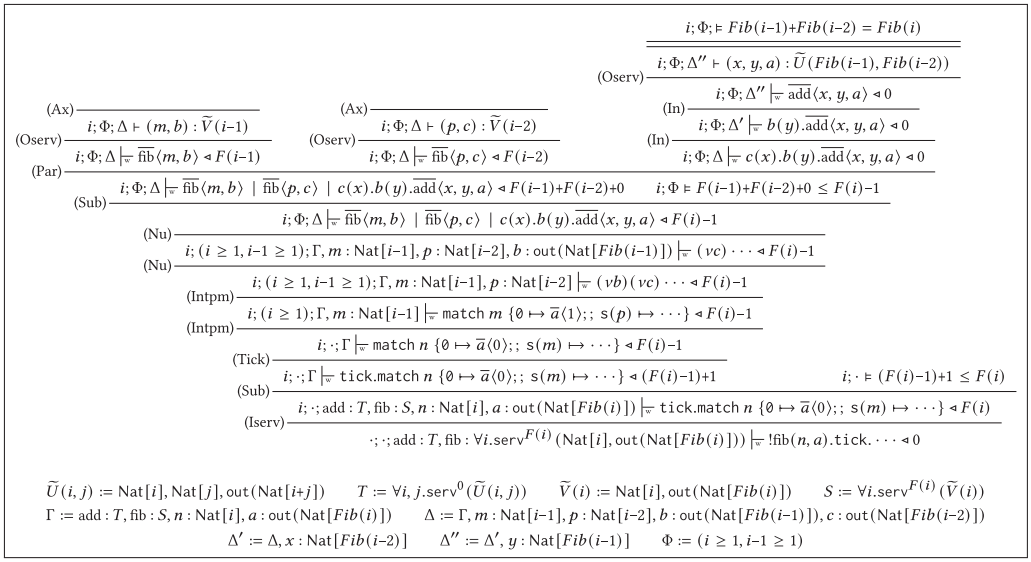
$$i;\Phi;\Delta \vdash_{\mathrm{w}} \overline{\mathrm{fib}}\langle m,b\rangle \mid \overline{\mathrm{fib}}\langle p,c\rangle \mid c(x).b(y).\overline{\mathrm{add}}\langle x,y,a\rangle \triangleleft F(i){-}1.$$

We would like to use a (Par) rule, but to do that, we need a complexity with the shape of a sum. So, we use a (Sub) rule before to transform this complexity $F(i){-}1$ into $F(i{-}1) + F(i{-}2) + 0$, which comes from the definition of F given in Example 4.1. Now that we have a sum, with the (Par) rule, we obtain three independent premises. The first two premises correspond to a recursive call to fib, and we again use polymorphism: We show that as we took (n,a) of types $\widetilde{V}(i)$, then we have (m,b) of types $\widetilde{V}(i{-}1)$ and (p,c) of types $\widetilde{V}(i{-}2)$. Thus, the first recursive call $\overline{\mathrm{fib}}\langle m,b\rangle$ is a call to fib with size $i{-}1$, and it has then complexity $F(i{-}1)$ and the second call $\overline{\mathrm{fib}}\langle p,c\rangle$ is a call to fib with size $i{-}2$ and it has then complexity $F(i{-}2)$.

Finally, the third premise corresponds to adding the results of the previous recursive calls. We first recover those results with two (In) rules, and then we have to show that $\overline{\mathrm{add}}\langle x,y,a\rangle$ is a well-typed call to add. Again, we use polymorphism, and we show that (x,y,a) has type $\widetilde{U}(Fib(i{-}1),Fib(i{-}2))$. Indeed, x has type $\mathsf{Nat}[Fib(i{-}1)]$, y has type $\mathsf{Nat}[Fib(i{-}2)]$, and so we only



(a) A Typing Without Complexity for Addition



(b) A Typing for Fibonacci (Work)

Fig. 8. Work typing of Fibonacci.

have to show that a has type $\mathtt{out}(\mathsf{Nat}[Fib(i{-}1){+}Fib(i{-}2)])$. As in Δ', we have the hypothesis $a : \mathtt{out}(\mathsf{Nat}[Fib(i)])$, we obtain this with subtyping, using the fact $i;\Phi \vDash Fib(i{-}1){+}Fib(i{-}2) = Fib(i)$.

4.5 Soundness of the Type System

We now state the properties of this typing system. We do not detail the proofs as all proofs are a simpler version than the one for span that will be described in later sections. In this type system for work, we can easily obtain some properties such as weakening and strengthening and that index variables can be substituted by any index in a typing derivation. Finally, we have that substitution in processes preserves typing. With those properties, we obtain the usual subject reduction.

Theorem 4.6 (Subject Reduction). *If* $\varphi;\Phi;\Gamma \vdash_w P \triangleleft K$ *and* $P \to_0 Q$, *then* $\varphi;\Phi;\Gamma \vdash_w Q \triangleleft K$.

Then, we also obtain the following theorem.

Theorem 4.7 (Quantitative Subject Reduction). *If $P \to_1 Q$ and $\varphi; \Phi; \Gamma \vdash_w P \triangleleft K$, then we have $\varphi; \Phi; \Gamma \vdash_w Q \triangleleft K'$ with $\varphi; \Phi \vDash K' + 1 \leq K$.*

Proof. By induction on $P \to_1 Q$. All the cases are direct, since the rule for parallel composition is the sum of complexity and the rule for ν does not change the complexity. Finally, the rule for tick gives directly this property. □

As a consequence, we almost immediately obtain that K is indeed a bound on the work of P if we have $\varphi; \Phi; \Gamma \vdash_w P \triangleleft K$. Formally, we have:

Theorem 4.8 (Work Complexity Bound). *If $\varphi; \Phi; \Gamma \vdash_w P \triangleleft K$, then, if we call n the work complexity of P, then we have $\varphi; \Phi \vDash K \geq n$. In particular, for any $\rho : \varphi \to \mathbb{N}$ such that $\rho \vDash \Phi$, we have $[\![K]\!]_\rho \geq n$.*

This complexity bound only makes sense if the set of constraint Φ is satisfiable.

Discussion. We emphasize on the fact that this soundness result is easily adaptable to similar processes and type systems for work. As stated before, we can enrich processes with other algebraic data-types and the proof can easily be adapted. An interesting consequence of this soundness theorem is that it immediately gives soundness for any subsystem. In particular, we detail in the next section a (slightly) weaker typing system where the shape of types are restricted to have an inference procedure close to the one in Reference [4].

Also note that additionally to complexity information, the type system also guarantees bounds on the size of values. For instance, if Γ contains $a : \mathsf{ch}(\mathsf{Nat}[I, J])$, then any term transmitted through the channel a during an execution will be of integer type and of size between I and J. If one was not interested in the work but only by bounds on the sizes of values, then one could consider a type system whose rules are obtained by dropping K in all rules of Figures 6(a) and 6(b).

Moreover, there is an asymmetry between servers and simple channels in our type system: for servers, complexity comes from output and thus an index to keep track of the complexity is needed in the type, whereas for simple channels, complexity comes from input. However, this asymmetry is not necessary for the work type system. Indeed, servers need to have those typing rules, as explained before because of polymorphism, but for simple channels, we have a choice: We can either keep the current type system or modify simple channel types such that they are similar to servers types (with polymorphism over indexes and an index for complexity in the type). We chose to present the current type system first, because this way, we can show this alternative choice of typing for simple channels instead of mimicking servers, and because this asymmetry is essential for the span type system because of the way we use time information. This alternative choice of typing is presented in the next section, in Figure 9.

Another interesting type system would be a non-syntax directed type-system, for which we have the choice between those two typing behaviours for simple channels. For the sake of simplicity, we avoid presenting this type system, but in fact the two choices have advantages and disadvantages and so allowing two different choices of typing can increase expressivity.

4.6 A Hint for Type Inference for Work

To infer a sized type for a process, we reduce the problem of finding a type derivation to that of satisfying a set of constraints on integers expressions. This idea has already been applied to sized type systems for functional languages [4, 29, 39] and has led to interesting results. Intuitively, this type inference procedure can be seen as a size reconstruction problem: We start from a type derivation without any size annotations, and we try to annotate it with sizes. We introduce for that variables for integer expressions, generate constraints on those variables and finally solve this set

of constraints by finding an appropriate expression for each variable. Here is a very simple and informal example in π-calculus:

$$\dfrac{\Gamma \vdash a : \mathsf{ch}(\mathsf{Nat}) \qquad \dfrac{\Gamma, n : \mathsf{Nat} \vdash b : \mathsf{ch}(\mathsf{Nat}) \qquad \dfrac{\Gamma, n : \mathsf{Nat} \vdash n : \mathsf{Nat}}{\Gamma, n : \mathsf{Nat} \vdash \mathsf{s}(n) : \mathsf{Nat}}}{\Gamma, n : \mathsf{Nat} \vdash \overline{b}\langle \mathsf{s}(n) \rangle}}{\Gamma \vdash a(n).\overline{b}\langle \mathsf{s}(n) \rangle,}$$

with $\Gamma := a : \mathsf{ch}(\mathsf{Nat}), b : \mathsf{ch}(\mathsf{Nat})$. We then reproduce this type derivation with additional variables for sizes, and some inferred constraints (for the sake of simplicity, we only consider upper bounds):

$$\dfrac{\Delta \vdash a : \mathsf{ch}(\mathsf{Nat}[0, I]) \qquad \dfrac{\Delta, n : \mathsf{Nat}[0, I] \vdash b : \mathsf{ch}(\mathsf{Nat}[0, J]) \qquad \dfrac{\Delta, n : \mathsf{Nat}[0, I] \vdash n : \mathsf{Nat}[0, I] \qquad \cdot \vDash J = I{+}1}{\Delta, n : \mathsf{Nat}[0, I] \vdash \mathsf{s}(n) : \mathsf{Nat}[0, J]}}{\Delta, n : \mathsf{Nat}[0, I] \vdash \overline{b}\langle \mathsf{s}(n) \rangle}}{\Delta \vdash a(n).\overline{b}\langle \mathsf{s}(n) \rangle,}$$

with $\Delta := a : \mathsf{ch}(\mathsf{Nat}[0, I]), b : \mathsf{ch}(\mathsf{Nat}[0, J])$. So, we only have one constraint in this typing, $J = I{+}1$. Thus, by substituting the variable J by the expression $I{+}1$, we obtain a valid typing for any expression I. Of course, in practice, constraints will be more complex, including, for example, recursive equations for an expression I when we consider servers.

In this section, we do not present the procedure for our type system, but we introduce a new system (that we call the intermediate type system) that is weaker that the previous one, but for which constraints inference will be simpler. Moreover, with this new system, we will obtain simpler constraints, essentially replacing constraints of the shape $\varphi; \Phi \vDash I \bowtie J$ by $\varphi'; \cdot \vDash I' \bowtie J'$, by removing the need for a set of constraints Φ in hypothesis.

To do this, we will mimic the work of Reference [4] for functional programs. In this work, usual types for functional programs are annotated with sizes, and in particular the arrow type $T \to U$ is given a type $\forall \widetilde{i}.T \to U$, with a polymorphism similar to the one we have for server types. After defining a restriction of those arrow types (called *canonical* types), this article describes a sound and complete inference procedure for this type system. Intuitively, in Reference [4], the canonical form of a type forces the input of functions to have the shape $\mathsf{Nat}[0, i]$ (or $\mathsf{List}[0, i](\mathcal{B})$) for some new fresh variable i. Then more complex indices (such as, e.g., $i^2{+}j$ or other expressions) can occur in types at positions that correspond to outputs.

Our intermediate type system will use those canonical forms. Then, we show that this system is a restriction of a more general system for work so that we automatically deduce the soundness of the intermediate type system (with regard to work complexity). We can then use a similar inference procedure as Reference [4], which we will not detail here. To sum up, we first generate constraints from a process such that if this set of constraints is satisfiable, then the process is typable (soundness). To show expressivity, we also want that if a process is typable, then the generated set of constraints is indeed satisfiable (completeness). With this, we know that we do not lose expressivity with the reduction to a constraint satisfaction problem. Then the second step is to give those constraints to a SMT solver and hope that it can solve it (recall that this is undecidable in general).

We begin by designing an alternative type system for work, by changing the behaviour of channels, to have a quantification for channel variables even for channel types, then we merge server and channel together. This way, we obtain a type for channel very close to the arrow type in Reference [4].

Definition 4.9. The set of types and *base types* for the intermediate type system are given by the following grammar:

$$\mathcal{B} := \mathsf{Nat}[I, J] \mid \mathsf{List}[I, J](\mathcal{B}), \qquad T := \mathcal{B} \mid \forall \widetilde{i}.\mathsf{ch}^K(\widetilde{T}) \mid \forall \widetilde{i}.\mathtt{in}^K(\widetilde{T}) \mid \forall \widetilde{i}.\mathtt{out}^K(\widetilde{T}).$$

$$\text{(Par)}\ \frac{\varphi;\Phi;\Gamma \vdash^{\text{alt}}_{\text{w}} P \triangleleft K_1 \qquad \varphi;\Phi;\Gamma \vdash^{\text{alt}}_{\text{w}} Q \triangleleft K_2}{\varphi;\Phi;\Gamma \vdash^{\text{alt}}_{\text{w}} P \mid Q \triangleleft K_1{+}K_2} \qquad \text{(Tick)}\ \frac{\varphi;\Phi;\Gamma \vdash^{\text{alt}}_{\text{w}} P \triangleleft K}{\varphi;\Phi;\Gamma \vdash^{\text{alt}}_{\text{w}} \mathtt{tick}.P \triangleleft K{+}1}$$

$$\text{(Rin)}\ \frac{\varphi;\Phi;\Gamma \vdash a : \forall \widetilde{i}.\mathsf{in}^K(\widetilde{T}) \qquad (\varphi,\widetilde{i});\Phi;\Gamma,\widetilde{v}:\widetilde{T} \vdash^{\text{alt}}_{\text{w}} P \triangleleft K}{\varphi;\Phi;\Gamma \vdash^{\text{alt}}_{\text{w}} !a(\widetilde{v}).P \triangleleft 0}$$

$$\text{(In)}\ \frac{\varphi;\Phi;\Gamma \vdash a : \forall \widetilde{i}.\mathsf{in}^K(\widetilde{T}) \qquad (\varphi,\widetilde{i});\Phi;\Gamma,\widetilde{v}:\widetilde{T} \vdash^{\text{alt}}_{\text{w}} P \triangleleft K}{\varphi;\Phi;\Gamma \vdash^{\text{alt}}_{\text{w}} a(\widetilde{v}).P \triangleleft 0}$$

$$\text{(Out)}\ \frac{\varphi;\Phi;\Gamma \vdash a : \forall \widetilde{i}.\mathsf{out}^K(\widetilde{T}) \qquad \varphi;\Phi;\Gamma \vdash \widetilde{e} : \widetilde{T}\{\widetilde{J}/\widetilde{i}\}}{\varphi;\Phi;\Gamma \vdash^{\text{alt}}_{\text{w}} \overline{a}\langle\widetilde{e}\rangle \triangleleft K\{\widetilde{J}/\widetilde{i}\}}$$

Fig. 9. Typing rules for processes for work inference.

We consider as subtyping rules the ones from Figures 7(a) and 7(b), where `serv` is replaced by `ch`. The typing rules are given by Figures 5, 6(a), and 9. Thus, the only modification with regard to the system we presented before is that channel types and server types are not distinct anymore. We then obtain the following theorem, by a proof similar to the one from the previous section:

Theorem 4.10. *If* $\varphi;\Phi;\Gamma \vdash^{alt}_{w} P \triangleleft K$, *then, if we call* n *the work complexity of* P, *we have* $\varphi;\Phi \vDash K \geq n$.

4.6.1 The Intermediate Type System. A natural way to design a type inference procedure for our system is to start with a standard algorithm that, given a process P, produces a classical type for it, and then devise a procedure that decorates this classical type derivation with sizes and complexity information, to obtain a valid derivation.

For now, we only have a type system that gives a sound bound on the complexity, described in Figure 9. However, this type system relies on constraints of the shape $\varphi;\Phi \vDash I \bowtie J$. As explained before, we would like to have simplified constraints, without hypotheses Φ, so that we can use, for example, the solver of Reference [4]. Thus, we restrict the type system it to obtain a system very close to Reference [4] allowing us to mimic the type inference procedure of this article. We call this type system the *intermediate type system*.

Definition 4.11. For this intermediate type system, we consider the following grammar:

$$\mathcal{B}_i := \mathsf{Nat}[0,I] \mid \mathsf{List}[0,I] \mid \alpha,\beta,\ldots, \qquad T_i := \mathcal{B}_i \mid \forall \widetilde{i}.\mathsf{ch}^K(\widetilde{T_i}) \mid A,B,\ldots,$$

where α,β are base type variables and A,B are type variables, that we use in the type inference procedure when the type of a value is unknown. Then, we define types in canonical form, as in Reference [4].

Definition 4.12 (Canonical Intermediate Types). A *canonical intermediate type* is a type of this shape:

$$\mathcal{B}_c := \mathsf{Nat}[0,i{+}n] \mid \mathsf{List}[0,i{+}n](\mathcal{B}_c) \mid \alpha,\beta,\ldots, \qquad T_c := \mathcal{B}_c \mid \forall i_1,\ldots,i_m.\mathsf{ch}^K(\widetilde{T_c}) \mid A,B,\ldots,$$

where i is an index variable, n an integer, and in the channel type, the index variables of K belong to $\{i_1,\ldots,i_m\}$ and m is equal to the number of base type index occurrences in $\widetilde{T_c}$, denoted $\mathtt{btocc}(\widetilde{T_c})$, and defined by:

- $\mathtt{btocc}(T_c^1,\ldots,T_c^k) = \mathtt{btocc}(T_c^1){+}\cdots{+}\mathtt{btocc}(T_c^k)$,
- $\mathtt{btocc}(\mathsf{Nat}[0,i]) = 1$,

- $\mathsf{btocc}(\mathsf{List}[0,i](\mathcal{B}_c)) = 1{+}\mathsf{btocc}(\mathcal{B}_c)$,
- $\mathsf{btocc}(\forall i_1,\ldots,i_m.\mathsf{ch}^K(\widetilde{T_c})) = \mathsf{btocc}(A) = \mathsf{btocc}(\alpha) = 0$.

Then, we also ask that for a canonical intermediate channel type, all the indices for base types corresponding to the base type index occurrences are an actual index variable (thus, $n = 0$), that they are all distinct, and in the left to right order (thus, we use all the different index variable names exactly once in base types and in a specific order). Moreover, as we are interested in an implementable procedure, a special focus is given to names of binding variables. We ask explicitly that in a canonical type, a binding name is never used twice in a type.

Example 4.13. The following type is a canonical channel type:

$$\forall i_1^1, i_2^1, i_3^1.\mathsf{ch}^{(i_1^1+i_3^1)}(\mathsf{Nat}[0,i_1^1], \forall i_1^2.\mathsf{ch}^0(\mathsf{Nat}[0,i_1^2], \mathsf{ch}^3()), \mathsf{List}[0,i_2^1](\mathsf{List}[0,i_3^1](\alpha)), A).$$

The first quantification is over three index variables, because there are exactly three positions in this type in which we can put an index variable without crossing another quantifier. Then, those three variables are indeed ordered from left to right. All the subtypes are also canonical, and notice that base type variables (A) and type variables (α) are not taken into account in the counting of index variables.

The use of canonical types is mainly to simplify the inference procedure. First, by fixing the number of variables after a quantification, we avoid having to guess the number of variables. Then, we try to put those variables at strategic locations, such that all variables give some useful information (i.e., the size of one base type). Moreover, with a canonical type, all unknown sizes are just a unique variable i and not a possibly convoluted expression, so that we can change the pattern matching rule to get rid of the constraints Φ of the previous type system, and we obtain in general simpler constraints.

Obviously, a canonical intermediate type is an intermediate type. In our intermediate type system, we will ask that all channel names have a canonical type. Moreover, in a context Γ, we will require all types to be canonical. Formally, a typing judgment has the shape $\varphi;\Gamma \vdash^{\mathrm{int}}_{\mathrm{w}} P \triangleleft K$, where Γ contains only canonical types, and there is no set of constraints Φ. Subtyping is defined as a restriction of the previous subtyping relation to intermediate types, and the type system is given by Figures 10(a) and 10(b). One can see that the invariants described above on canonical types are respected. Also, there are no subtyping rules in this type system, as canonical types do not need subtyping, and the modification on the complexity bound are internalized in the useful rules. This way, all rules are syntax-directed, which simplifies the inference procedure. Moreover, in this typing, and for the following of this section, we only consider processes with well written pattern matching, meaning that pattern matching can only be done on base type variables (otherwise, the pattern matching is not useful...). Thanks to this restriction, we can consider that the base type element in a pattern matching is a canonical type, and this helps us to get rid of the previous rule for pattern matching where we needed to add constraints in the branches. The idea is that instead of adding the constraint $i \geq 1$ to the typing judgement, we perform a substitution of i by an index of the shape $i{+}1$. Thus i now becomes the size of the predecessor (or the tail for a list). Without this set of constraints Φ in a typing, we obtain a type system closer to the one of Reference [4]. To sum up, the rules are similar to the one for $\vdash^{\mathrm{alt}}_{\mathrm{w}}$, but subtyping is internalized in the rules, and pattern matching is modified to get rid of the constraints Φ.

Now, let us show that if a process is typable with $\vdash^{\mathrm{int}}_{\mathrm{w}}$, then it is typable with $\vdash^{\mathrm{alt}}_{\mathrm{w}}$.

Theorem 4.14. *If a process P is such that $\varphi;\Gamma \vdash^{int}_{w} P \triangleleft K$, then for any Γ_{sub} obtained from Γ by substituting all type variables by actual types, we have $\varphi;\cdot;\Gamma_{sub} \vdash^{alt}_{w} P \triangleleft K$.*

$$\frac{v : T_c \in \Gamma}{\varphi; \Gamma \left|^{\text{int}}_{\text{w}}\right. v : T_c} \qquad \frac{}{\varphi; \Gamma \left|^{\text{int}}_{\text{w}}\right. 0 : \mathsf{Nat}[0,0]} \qquad \frac{\varphi; \Gamma \left|^{\text{int}}_{\text{w}}\right. e : \mathsf{Nat}[0,I]}{\varphi; \Gamma \left|^{\text{int}}_{\text{w}}\right. \mathsf{s}(e) : \mathsf{Nat}[0,I{+}1]}$$

$$\frac{}{\varphi; \Gamma \left|^{\text{int}}_{\text{w}}\right. [\,] : \mathsf{List}[0,0]} \qquad \frac{\varphi; \Gamma \left|^{\text{int}}_{\text{w}}\right. e : \mathcal{B}_i^1 \qquad \varphi; \Gamma \left|^{\text{int}}_{\text{w}}\right. e' : \mathsf{List}[0,I](\mathcal{B}_i^2) \qquad \varphi; \cdot \left|^{\text{int}}_{\text{w}}\right. \mathcal{B}_i^1, \mathcal{B}_i^2 \sqsubseteq \mathcal{B}_i}{\varphi; \Gamma \left|^{\text{int}}_{\text{w}}\right. e :: e' : \mathsf{List}[0,I{+}1](\mathcal{B}_i)}$$

(a) Intermediate Typing Rules for Expressions

$$\frac{}{\varphi; \Gamma \left|^{\text{int}}_{\text{w}}\right. 0 \triangleleft 0} \qquad \frac{\varphi; \Gamma \left|^{\text{int}}_{\text{w}}\right. P \triangleleft K_1 \qquad \varphi; \Gamma \left|^{\text{int}}_{\text{w}}\right. Q \triangleleft K_2 \qquad \varphi; \cdot \vDash K_1{+}K_2 \leq K}{\varphi; \Gamma \left|^{\text{int}}_{\text{w}}\right. P \mid Q \triangleleft K} \qquad \frac{\varphi; \Gamma, a : T_c \left|^{\text{int}}_{\text{w}}\right. P \triangleleft K}{\varphi; \Gamma \left|^{\text{int}}_{\text{w}}\right. (\nu a)P \triangleleft K}$$

$$\frac{\varphi; \Gamma \left|^{\text{int}}_{\text{w}}\right. a : \forall \widetilde{i}.\mathsf{ch}^K(\widetilde{T_c}) \qquad (\varphi, \widetilde{i}); \Gamma, \widetilde{v} : \widetilde{T_c} \left|^{\text{int}}_{\text{w}}\right. P \triangleleft K' \qquad (\varphi, \widetilde{i}); \cdot \vDash K' \leq K}{\varphi; \Gamma \left|^{\text{int}}_{\text{w}}\right. !a(\widetilde{v}).P \triangleleft 0}$$

$$\frac{\varphi; \Gamma \left|^{\text{int}}_{\text{w}}\right. a : \forall \widetilde{i}.\mathsf{ch}^K(\widetilde{T_c}) \qquad (\varphi, \widetilde{i}); \Gamma, \widetilde{v} : \widetilde{T_c} \left|^{\text{int}}_{\text{w}}\right. P \triangleleft K' \qquad (\varphi, \widetilde{i}); \cdot \vDash K' \leq K}{\varphi; \Gamma \left|^{\text{int}}_{\text{w}}\right. a(\widetilde{v}).P \triangleleft 0}$$

$$\frac{\varphi; \Gamma \left|^{\text{int}}_{\text{w}}\right. a : \forall \widetilde{i}.\mathsf{ch}^K(\widetilde{T_c}) \qquad \varphi; \Gamma \left|^{\text{int}}_{\text{w}}\right. \widetilde{e} : \widetilde{T_c}\{\widetilde{j}/\widetilde{i}\} \qquad \varphi; \cdot \vDash K\{\widetilde{j}/\widetilde{i}\} \leq K'}{\varphi; \Gamma \left|^{\text{int}}_{\text{w}}\right. \overline{a}\langle \widetilde{e} \rangle \triangleleft K'} \qquad \frac{\varphi; \Gamma \left|^{\text{int}}_{\text{w}}\right. P \triangleleft K}{\varphi; \Gamma \left|^{\text{int}}_{\text{w}}\right. \mathtt{tick}.P \triangleleft K{+}1}$$

$$\frac{\varphi; \Gamma \left|^{\text{int}}_{\text{w}}\right. v : \mathsf{Nat}[0,i] \qquad \varphi; \Gamma \left|^{\text{int}}_{\text{w}}\right. P \triangleleft K_1 \qquad \varphi; \cdot \vDash K_1 \leq K \qquad \varphi; \Gamma\{i{+}1/i\}, x : \mathsf{Nat}[0,i] \left|^{\text{int}}_{\text{w}}\right. Q \triangleleft K_2 \qquad \varphi; \cdot \vDash K_2 \leq K\{i{+}1/i\}}{\varphi; \Gamma \left|^{\text{int}}_{\text{w}}\right. \mathtt{match}\ v\ \{0 \mapsto P; ;\ \mathsf{s}(x) \mapsto Q\} \triangleleft K}$$

$$\frac{\varphi; \Gamma \left|^{\text{int}}_{\text{w}}\right. v : \mathsf{Nat}[0,i{+}n{+}1] \qquad \varphi; \Gamma \left|^{\text{int}}_{\text{w}}\right. P \triangleleft K_1 \qquad \varphi; \cdot \vDash K_1 \leq K \qquad \varphi; \Gamma, x : \mathsf{Nat}[0,i{+}n] \left|^{\text{int}}_{\text{w}}\right. Q \triangleleft K_2 \qquad \varphi; \cdot \vDash K_2 \leq K}{\varphi; \Gamma \left|^{\text{int}}_{\text{w}}\right. \mathtt{match}\ v\ \{0 \mapsto P; ;\ \mathsf{s}(x) \mapsto Q\} \triangleleft K}$$

$$\frac{\varphi; \Gamma \left|^{\text{int}}_{\text{w}}\right. v : \mathsf{List}[0,i](\mathcal{B}_i) \qquad \varphi; \Gamma \left|^{\text{int}}_{\text{w}}\right. P \triangleleft K_1 \qquad \varphi; \Gamma\{i{+}1/i\}, x : \mathcal{B}_i, y : \mathsf{List}[0,i](\mathcal{B}_i) \left|^{\text{int}}_{\text{w}}\right. Q \triangleleft K_2 \qquad \varphi \vDash K_1 \leq K; K_2 \leq K\{i{+}1/i\}}{\varphi; \Gamma \left|^{\text{int}}_{\text{w}}\right. \mathtt{match}\ v\ \{[\,] \mapsto P; ;\ x :: y \mapsto Q\} \triangleleft K}$$

$$\frac{\varphi; \Gamma \left|^{\text{int}}_{\text{w}}\right. v : \mathsf{List}[0,i{+}n{+}1](\mathcal{B}_i) \qquad \varphi; \Gamma \left|^{\text{int}}_{\text{w}}\right. P \triangleleft K_1 \qquad \varphi; \cdot \vDash K_1 \leq K \qquad \varphi; \Gamma, x : \mathcal{B}_i, y : \mathsf{List}[0,i{+}n](\mathcal{B}_i) \left|^{\text{int}}_{\text{w}}\right. Q \triangleleft K_2 \qquad \varphi; \cdot \vDash K_2 \leq K}{\varphi; \Gamma \left|^{\text{int}}_{\text{w}}\right. \mathtt{match}\ v\ \{[\,] \mapsto P; ;\ x :: y \mapsto Q\} \triangleleft K}$$

(b) Intermediate Typing Rules for Processes

Fig. 10. Intermediate typing rules.

From this, a direct corollary is that if $\varphi; \Gamma \left|^{\text{int}}_{\text{w}}\right. P \triangleleft K$, then K is a bound on the work of P.

Proof. The proof is done by induction on $\varphi; \Gamma \left|^{\text{int}}_{\text{w}}\right. P \triangleleft K$. The type variables do not cause any problem, since in a intermediate typing, if we can use a type variable, then it means that this type is never really inspected. When typing an expression, this is rather straightforward, using subtyping for expressions. Again, for typing rules for processes, many cases are straightforward just by using the subtyping rule of Figure 6(a). Thus, we only detail the interesting case.

If the typing is

$$\frac{\varphi; \Gamma \left|^{\text{int}}_{\text{w}}\right. v : \mathsf{Nat}[0,\, i] \quad \varphi; \Gamma \left|^{\text{int}}_{\text{w}}\right. P \triangleleft K_1 \quad \varphi; \cdot \vDash K_1 \leq K \quad \varphi; \Gamma\{i{+}1/i\}, x : \mathsf{Nat}[0,\, i] \left|^{\text{int}}_{\text{w}}\right. Q \triangleleft K_2 \quad \varphi; \cdot \vDash K_2 \leq K\{i{+}1/i\}}{\varphi; \Gamma \left|^{\text{int}}_{\text{w}}\right. \mathtt{match}\ v\ \{0 \mapsto P; ;\ \mathsf{s}(x) \mapsto Q\} \triangleleft K,}$$

then we would like to give the following type:

$$\frac{\varphi; \cdot; \Gamma_{sub} \left|^{\text{alt}}_{\text{w}}\right. v : \mathsf{Nat}[0,\, i] \qquad \varphi; (0 \leq 0); \Gamma_{sub} \left|^{\text{alt}}_{\text{w}}\right. P \triangleleft K \qquad \varphi; (i \geq 1); \Gamma_{sub}, x : \mathsf{Nat}[0{-}1,\, i{-}1] \left|^{\text{alt}}_{\text{w}}\right. Q \triangleleft K}{\varphi; \cdot; \Gamma_{sub} \left|^{\text{alt}}_{\text{w}}\right. \mathtt{match}\ v\ \{0 \mapsto P; ;\ \mathsf{s}(x) \mapsto Q\} \triangleleft K.}$$

From the typing $\varphi; \Gamma \left|\frac{\text{int}}{\text{w}}\right. P \triangleleft K_1$, obtaining $\varphi; (0 \leq 0); \Gamma_{sub} \left|\frac{\text{alt}}{\text{w}}\right. P \triangleleft K$ is direct by induction hypothesis. Now, by induction hypothesis, we also obtain

$$\varphi; \cdot; \Gamma_{sub}\{i{+}1/i\}, x : \text{Nat}[0, i] \left|\frac{\text{alt}}{\text{w}}\right. Q \triangleleft K_2, \qquad \varphi; \cdot \vDash K_2 \leq K\{i{+}1/i\}.$$

Then, by index substitution, we have

$$\varphi; \cdot; \Gamma_{sub}\{i{+}1/i\}\{i{-}1/i\}, x : \text{Nat}[0, i{-}1] \left|\frac{\text{alt}}{\text{w}}\right. Q \triangleleft K_2\{i{-}1/i\}, \qquad \varphi; \cdot \vDash K_2\{i{-}1/i\} \leq K\{i{+}1/i\}\{i{-}1/i\}.$$

Thus, by weakening, we obtain

$$\varphi; (i \geq 1); \Gamma_{sub}\{i{+}1/i\}\{i{-}1/i\}, x : \text{Nat}[0, i{-}1] \left|\frac{\text{alt}}{\text{w}}\right. Q \triangleleft K_2\{i{-}1/i\},$$

$$\varphi; (i \geq 1) \vDash K_2\{i{-}1/i\} \leq K\{i{+}1/i\}\{i{-}1/i\}.$$

Then, as we have $i \geq 1$, we obtain $(i{-}1){+}1 = i$. Moreover, $0{-}1 = 0$ by definition. So, by subtyping, we obtain

$$\varphi; (i \geq 1); \Gamma_{sub}, x : \text{Nat}[0{-}1, i{-}1] \left|\frac{\text{alt}}{\text{w}}\right. Q \triangleleft K.$$

We can conclude the proof as the typing above can be obtained.

The other case of pattern matching with $n > 0$ is easier. □

Thus, if we have a correct and complete procedure for this intermediate type system, then we obtain indeed a bound on the complexity thanks to the soundness result of the previous type system. Moreover, this new type system does not seem too restrictive in practice. The main problem is that we cannot have too much information about the input. For example, to have a more understandable type, we might like to assume that an input list has a length 2^i for some i. With this type system, this kind of assumption is not possible anymore, thus we would need to use the logarithm to obtain a similar reasoning. In general, we can lose some information about the input of a function that could have been useful. Subtyping also becomes very restrained, we no longer have input/output types and subtyping between canonical forms is not really useful, this is just equality everywhere. Note that in practice, because of canonical types, checking type equality is easy, since canonical forms drastically simplify the problems of α-renaming and reordering of quantifiers.

Still, with this intermediate type system, we can mimic the work in Reference [4], and we obtain a procedure that is sound and complete for this intermediate type system, and such that the inferred constraints can be fed to the solver of Reference [4]. We have not explored yet a procedure that would work directly for our original type system and be complete with regard to this system. Moreover, we do not know if the inferred constraints would be solvable in practice.

5 TYPE SYSTEM FOR PARALLEL COMPLEXITY

5.1 Definition of Span

We now define another notion of complexity taking into account parallelism. Before formally presenting the semantics, we show with some simple examples what kind of properties we would like for this parallel complexity.

First, we want a parallel complexity that works as if we had an infinite number of processors. For instance, on the process $\texttt{tick}.\mathbb{0} \mid \texttt{tick}.\mathbb{0} \mid \texttt{tick}.\mathbb{0} \mid \cdots \mid \texttt{tick}.\mathbb{0}$, we want the complexity to be 1, no matter the number of `tick` in parallel.

Moreover, reductions with a zero-cost complexity (in our setting, this should mean all reductions except when we reduce a `tick`) should not harm this maximal parallelism. For example, $a().\texttt{tick}.\mathbb{0} \mid \overline{a}\langle\rangle \mid \texttt{tick}.\mathbb{0}$ should also have complexity one, because intuitively this synchronization between the input and the output can be done independently of the `tick` on the right, and then the `tick` on the left can be reduced in parallel with the `tick` on the right.

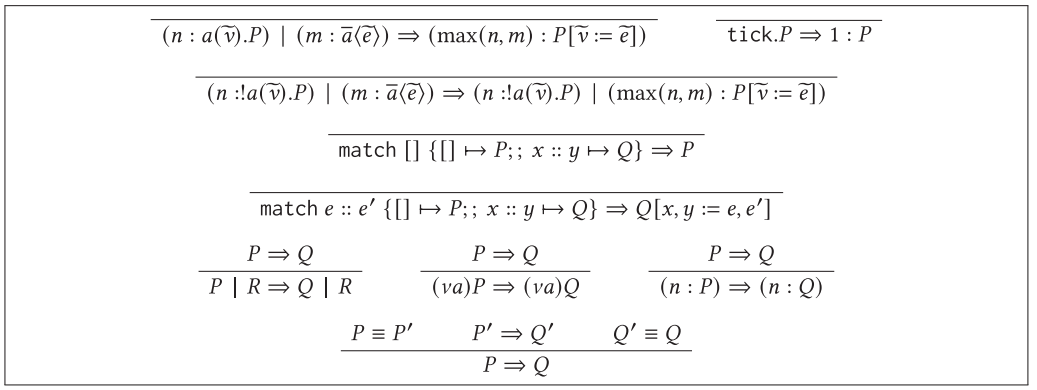


Fig. 11. Reduction rules.

Finally, as before for the work, adding a `tick` should not change the behaviour of a process. For instance, consider the process $\texttt{tick}.a().P_0 \mid a().\texttt{tick}.P_1 \mid \overline{a}\langle\rangle$, where a is not used in P_0 and P_1. This process should have the complexity $max(1+C_0, 1+C_1)$, where C_i is the cost of P_i. Indeed, there are two possible reductions, either we reduce the tick, and then we synchronize the left input with the output, and continue with P_0, or we first do the synchronization with the right input and the output, we then reduce the ticks, and finally we continue as P_1.

A possible way to define such a parallel complexity would be to adapt causal complexity [14–16]; however, we believe there is a simpler presentation for our case. We will show in Section 7 the equivalence between our notion and a kind of causal complexity. The idea for defining span has been proposed by Naoki Kobayashi (private communication). It consists in introducing a new construction for processes, $m : P$, where m is an integer. A process using this constructor will be called an *annotated process*. Intuitively, this annotated process has the meaning P *with* m *ticks before*. We can then enrich the congruence relation $\equiv$ with the following rules:

$$m : (P \mid Q) \equiv (m : P) \mid (m : Q), \qquad m : (\nu a)P \equiv (\nu a)(m : P), \qquad m : (n : P) \equiv (m + n) : P, \qquad 0 : P \equiv P.$$

This intuitively means that the ticks can be distributed over parallel composition, name creation can be done before or after ticks without changing the semantics, ticks can be grouped together, and finally zero tick is equivalent to nothing.

With this congruence relation and this new constructor, we can give a new shape to the canonical form presented in Definition 3.1.

Definition 5.1 (Canonical Form for Annotated Processes). An annotated process is in canonical form if it has the shape

$$(\nu\widetilde{a})(n_1 : G_1 \mid \cdots \mid n_m : G_m),$$

with $G_1, \ldots, G_m$ guarded annotated processes.

Note that the congruence relation above allows one to obtain this canonical form from any annotated processes. With this intuition in mind, we can then define a reduction relation $\Rightarrow$ for annotated processes. The rules are given in Figure 11. We do not detail the rules for integers as they are deducible from the ones for lists. Intuitively, this semantics works as the usual semantics for π-calculus, but when doing a synchronization, we keep the maximal annotation, and ticks are memorized in the annotations.

We then define the parallel complexity of an annotated process.

Definition 5.2 (Span). Let P be an annotated process. We define its *local complexity* $C_\ell(P)$ by:

- $C_\ell(n : P) = n{+}C_\ell(P)$,
- $C_\ell(P \mid Q) = \max(C_\ell(P), C_\ell(Q))$,
- $C_\ell((va)P) = C_\ell(P)$,
- $C_\ell(G) = 0$ if G is a guarded process.

Equivalently, $C_\ell(P)$ is the maximal integer that appears in the canonical form of P. Then, for an annotated process P, its *span* is given by $\text{span}(P) := \max\{n \mid P \Rightarrow^* Q \wedge C_\ell(Q) = n\}$ where $\Rightarrow^*$ is the reflexive and transitive closure of $\Rightarrow$.

To show that this parallel complexity is well-behaved, we give the following lemmas.

Lemma 5.3 (Congruence and Local Complexity). *Let P, Q be annotated processes such that $P \equiv Q$. Then, we have $C_\ell(P) = C_\ell(Q)$.*

Lemma 5.4 (Reduction and Local Complexity). *Let P, P' be annotated processes such that $P \Rightarrow P'$. Then, we have $C_\ell(P') \geq C_\ell(P)$.*

Those lemmas are proved by induction. The main point for the second lemma is that guarded processes have a local complexity equal to zero, thus doing a reduction will always increase this local complexity. That is why, to bound the complexity of an annotated process, we need to reduce it with $\Rightarrow$, and then we have to take the maximum local complexity over all normal forms. Moreover, this semantics respects the conditions given in the beginning of this section.

Let us now present a type system for span. We want as previously a type system such that typing a process gives us a bound on its span.

5.2 Sized Types with Time

The type system is an extension of the previous one for work. To take into account parallelism, we need a way to synchronize the time between processes in parallel, thus we will add some time information in types, as in Reference [30] or Reference [12]. Formally, we start with the types for work, and we decorate channels with an index I indicating time. Recall that in our calculus, we consider that one unit of time corresponds to the time expressed by a `tick`.

Definition 5.5. The set of types and *base types* are given by the following grammar:

$$\mathcal{B} := \mathsf{Nat}[I, J] \mid \mathsf{List}[I, J](\mathcal{B}),$$

$$T := \mathcal{B} \mid \mathsf{ch}_I(\widetilde{T}) \mid \mathsf{in}_I(\widetilde{T}) \mid \mathsf{out}_I(\widetilde{T}) \mid \forall \widetilde{i}.\mathtt{serv}_I^K(\widetilde{T}) \mid \forall \widetilde{i}.\mathtt{iserv}_I^K(\widetilde{T}) \mid \forall \widetilde{i}.\mathtt{oserv}_I^K(\widetilde{T}).$$

As before, we have channel types, server types, and input/output capabilities in those types. For a channel type or a server type, the index I is called the *time* of this type. Giving a channel name the type $\mathsf{ch}_I(\widetilde{T})$ ensures that communication on this channel should happen at a time t such that $0 \leq t \leq I$. For example, a channel name of type $\mathsf{ch}_0(\widetilde{T})$ should be used to communicate before any tick occurs. With this information, we have a bound on when the continuation P of an input $a(\widetilde{v}).P$ will be available. A server name of type $\forall \widetilde{i}.\mathtt{iserv}_I^K(\widetilde{T})$ has a slightly different meaning: it means that the inputs for this server should all be ready to receive for any time greater than I. For instance, a process $\mathtt{tick}.!a(v).P$ enforces that the type of a is $\forall \widetilde{i}.\mathtt{serv}_I^K(\widetilde{T})$ with I greater than one, as the replicated input is not ready to receive at time zero. About the free variables in a server types, $\forall \widetilde{i}.\mathtt{serv}_I^K(\widetilde{T})$, we emphasize the fact that $\widetilde{i}$ can appear in K and $\widetilde{T}$ but not in I. Indeed, the time that a server takes to become ready does not depend on the values sent to this server. However, to harmonize notations with channels, this time I is written after the quantification over $\widetilde{i}$ even if it does not depend on it.

As before, we define a notion of subtyping on those types. The rules are essentially the same as the ones in Figures 7(a) and 7(b). The only difference is that we enforce the time of a type to be invariant in subtyping.

To write the typing rules, we need some other definitions to work with time in types. The first thing we need is a way to advance time. Indeed, consider the simple example $\mathtt{tick}.a() \mid \mathtt{tick}.\overline{a}\langle\rangle$, when we consider the left branch, then after the `tick` (i.e., after one time unit), both the input and the output should be ready (contrary to work where we would still have to wait one time unit). Thus, we have to express that the time advances in all the parallel branches simultaneously, and this is done by the operator we will introduce.

Definition 5.6 (Advancing Time in Types). Given a set of index variables φ, a set of constraints Φ, a type T and an index I, we define T after I time units, denoted $\langle T\rangle_{-I}^{\varphi;\Phi}$ by:

- $\langle \mathcal{B}\rangle_{-I}^{\varphi;\Phi} = \mathcal{B}$.
- $\langle \mathsf{ch}_J(\widetilde{T})\rangle_{-I}^{\varphi;\Phi} = \mathsf{ch}_{(J-I)}(\widetilde{T})$ if $\varphi;\Phi \vDash J \geq I$. It is undefined otherwise.
  Other channel types follow exactly the same pattern.
- $\langle \forall \widetilde{i}.\mathtt{serv}_J^K(\widetilde{T})\rangle_{-I}^{\varphi;\Phi} = \forall \widetilde{i}.\mathtt{serv}_{J-I}^K(\widetilde{T})$ if $\varphi;\Phi \vDash J \geq I$.
  Otherwise, $\langle \forall \widetilde{i}.\mathtt{serv}_J^K(\widetilde{T})\rangle_{-I}^{\varphi;\Phi} = \forall \widetilde{i}.\mathtt{oserv}_{J-I}^K(\widetilde{T})$.
- $\langle \forall \widetilde{i}.\mathtt{iserv}_J^K(\widetilde{T})\rangle_{-I}^{\varphi;\Phi} = \forall \widetilde{i}.\mathtt{iserv}_{J-I}^K(\widetilde{T})$ if $\varphi;\Phi \vDash J \geq I$.
  It is undefined otherwise.
- $\langle \forall \widetilde{i}.\mathtt{oserv}_J^K(\widetilde{T})\rangle_{-I}^{\varphi;\Phi} = \forall \widetilde{i}.\mathtt{oserv}_{J-I}^K(\widetilde{T})$.

This definition can be extended to contexts, with $\langle v : T, \Gamma\rangle_{-I}^{\varphi;\Phi} = v : \langle T\rangle_{-I}^{\varphi;\Phi}, \langle\Gamma\rangle_{-I}^{\varphi;\Phi}$ if $\langle T\rangle_{-I}^{\varphi;\Phi}$ is defined. Otherwise, $\langle v : T, \Gamma\rangle_{-I}^{\varphi;\Phi} = \langle\Gamma\rangle_{-I}^{\varphi;\Phi}$. We will often omit $\varphi;\Phi$ in the notation when it is clear from the context. Recall that as the order $\leq$ on indexes is not total, $\varphi;\Phi \nvDash J \geq I$ does not mean that $\varphi;\Phi \vDash J < I$.

Let us explain a bit the definition here. For base types, there is no time indication, thus nothing happens. For simple channel types, there are two cases. Either the bound J on the time was greater than I, and we only have to subtract I from J to make time pass. Or the bound J was not greater than I, and this channel cannot be used any more after I units of time, thus we erase this channel from the context.

To understand server types, let us again look at a simple example $\mathtt{tick}.!a().P \mid \overline{a}\langle\rangle \mid \mathtt{tick}.\mathtt{tick}.\overline{a}\langle\rangle$. As explained before, the time of a server only corresponds to the time it takes for the input to be ready, thus the time I of this server should be one. The second branch $\overline{a}\langle\rangle$ should have this information, because it means it has to wait one unit of time, and the third branch $\mathtt{tick}.\mathtt{tick}.\overline{a}\langle\rangle$ should only know that the corresponding input is already ready after two time units. Moreover, by definition, after at least two time units, it should not be possible any more to redefine an input for this server, as it would break the invariant "all inputs are ready before I time units."

If we formalize this intuition, then we obtain a dissymmetry between the input and output capabilities for servers. The input capability behaves like a simple channel, if the time J is too small, we erase this capability, since we cannot define any more an input. However, the output capability is never erased, even if the time J is too small, but we still need this information (when $I \geq 0$) to estimate the time of waiting for the input to be available.

We remark that this definition of time advancing generates constraints in a type derivation. Indeed, if we want $\langle a : \mathsf{ch}_J(\widetilde{T})\rangle_{-1}^{\varphi;\Phi} = a : \mathsf{ch}_{J-1}(\widetilde{T})$, then we must have $\varphi;\Phi \vDash J \geq 1$. This will be useful to see what constraints the complexity of a process should satisfy, as we will see in examples.

We also need another definition on time information.

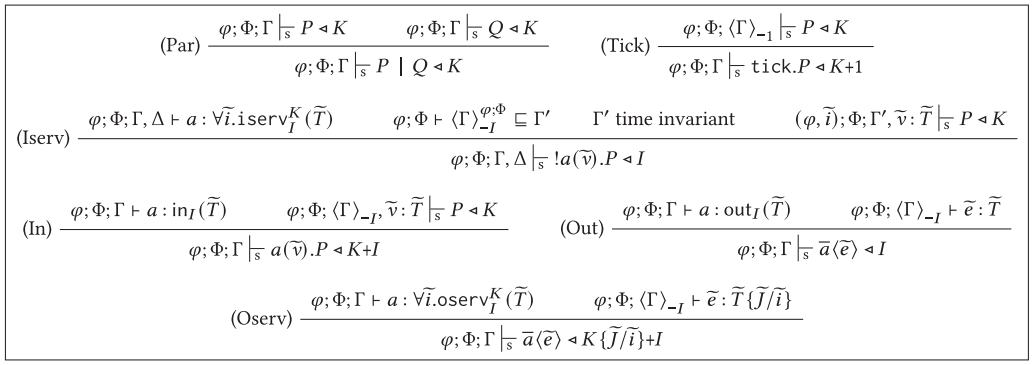


Fig. 12. Span typing rules for processes.

Definition 5.7 (Time Invariant Context). Given a set of index variables φ and a set of constraints Φ, a context Γ is said to be *time invariant* when it only contains base type variables or output server types $\forall\widetilde{i}.\mathtt{oserv}_I^K(\widetilde{T})$ with $\varphi;\Phi \vDash I = 0$.

Such a context is thus invariant by the operator $\langle\cdot\rangle_{-I}$ for any I. This is typically the kind of context that we need to define a server, as a server should not be dependent on the time it is called. We can now present the type system. Typing rules for expressions and some processes do not change; they can be found in Figures 5 and 6(a). In Figure 12, we present the remaining rules in this type system that differ from the ones in Figure 6(b). As before, a typing judgement $\varphi;\Phi;\Gamma \vdash_s P \triangleleft K$ intuitively means that under the constraints Φ, in a context Γ, a process P is typable and its span complexity is bounded by K.

The rule (Par) means that the parallel complexity is the maximum between the two processes instead of the sum (as explained in Section 4 for the (Intpm) rule, asking for the same complexity K is equivalent to taking the maximum because of the (Sub) rule). The (Tick) rule again says that complexity increases by 1, but it also decreases all time information in types by one.

Then, the (In) rule, the (Out) rule, and the (Oserv) rule are very similar to the ones for work. The only difference is this time information I. Indeed, in the span type system, as explained before, time should advance simultaneously in all parallel branches; however, when we consider a process of the shape $a(\widetilde{v}).P$, this process is not reduced immediately as it has to wait for a corresponding output to be ready. Thus, we have to estimate the time of waiting. This is estimated by this index I, and so we consider that I time units should pass before proceeding to type the continuation, as we can see with the $\langle\Gamma\rangle_{-I}$ operation.

Finally, the (Iserv) rules is far more complex that the previous one for work. Again, we have to wait those I units of time to synchronize time for all subprocesses. An important difference between input server and all the other processes is that, if an input server is ready to receive a message at time t, it should always be ready to receive a message at time $t' \geq t$, so that we can always be able to make a recursive call. Because of this, we ask that the context Γ' in the continuation is time invariant, so that this continuation does not depend on current time. So, to have this time invariant context, we integrate some weakening on context (with Δ) and an explicit subtyping rule in the premises. This is because $\langle\Gamma\rangle_{-I}^{\varphi;\Phi}$ is not time invariant in general, since the type of a contains the input capability. However, if this server has both input and output capabilities, we can give a time invariant type for a (or other servers) just by removing the input capability, which can be done by subtyping.

Finally, notice that because of the advance of time in a context, some channels name could disappear (because their time is up), thus there is a kind of "time uniqueness" for channels, contrary to the previous section on work. This will be detailed later in Section 5.3.4.

Before moving to examples, let us show what results we expect from this type system.

Theorem 5.8 (Typing and Complexity). *Let P be a process such that $\varphi;\Phi;\Gamma \vdash_s P \triangleleft K$, then $\varphi;\Phi \vDash K \geq \mathsf{span}(P)$.*

Note that this theorem talks about open processes. However, our notion of span does not behave well with open processes. For example, the process $\mathtt{match}\ v\ \{\mathtt{0} \mapsto P;;\ \mathsf{s}(x) \mapsto Q\}$ is in normal form for a variable v, thus this process has span zero. However, with the type system, we will not give it a zero complexity. This is because we will also obtain the following corollary:

Corollary 5.9 (Complexity and Open Processes). *We have:*

- *If $\varphi;\Phi;\Gamma,\widetilde{v}:\widetilde{T} \vdash_s P \triangleleft K$, then for any sequence of expressions $\widetilde{e}$ such that $\varphi;\Phi;\Gamma \vdash_s \widetilde{e}:\widetilde{T}$, then $\varphi;\Phi \vDash K \geq \mathsf{span}(P[\widetilde{v}:=\widetilde{e}])$.*
- *If $\varphi;\Phi;\Gamma \vdash_s P \triangleleft K$, then for any other annotated process Q such that $\varphi;\Phi;\Gamma \vdash_s Q \triangleleft K'$, we have $\varphi;\Phi \vDash max(K,K') \geq \mathsf{span}(P \mid Q)$.*

So, when we have a typing $\varphi;\Phi;\Gamma \vdash_s P \triangleleft K$ for an open process, one should not see K as a bound on the actual complexity on P, but rather as a bound on the complexity of this particular process in an environment respecting the type of Γ. So, in $\varphi;\Phi;v:\mathsf{Nat}[2,10] \vdash_s \mathtt{match}\ v\ \{\mathtt{0} \mapsto P;;\ \mathsf{s}(x) \mapsto Q\} \triangleleft K$, this index K is a bound on the complexity of this pattern matching under the assumption that the environment gives to v an integer value between 2 and 10.

We can then, for example, understand the rule for input by taking the judgement $\varphi;\Phi;a:\mathsf{ch}_3() \vdash_s a().\mathtt{tick}.\mathtt{0} \triangleleft 4$. This expresses that with an environment providing a message on a within 3 time units, this process terminates in 4 time units.

Discussion. We observe that if we remove all size annotations and merge server types and channel types together, then we get back the classical input/output types, and all the rules described here are admissible in the classical input/output type system for the π-calculus.

5.3 Examples

5.3.1 Input, Time, and Complexity. We first illustrate the time system on a simple generic example that will be very useful for other concrete examples. Let us consider the process $P = c().b().\overline{a}\langle\rangle$. We have the following typing for P:

$$\varphi;\Phi;a:\mathsf{out}_{I_a}(),b:\mathsf{in}_{I_b}(),c:\mathsf{in}_{I_c}() \vdash_s P \triangleleft I_a,$$

if and only if $\varphi;\Phi \vDash I_a \geq I_b \geq I_c$. Indeed, the typing derivation has the shape

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\varphi;\Phi;a:\mathsf{out}_{(I_a-I_c)-(I_b-I_c)}(),b:\mathsf{in}_0() \vdash_s \overline{a}\langle\rangle \triangleleft (I_a-I_c)-(I_b-I_c)}}{\varphi;\Phi;\langle(a:\mathsf{out}_{I_a-I_c}(),b:\mathsf{in}_{I_b-I_c}(),c:\mathsf{in}_0())\rangle_{-(I_b-I_c)} \vdash_s \overline{a}\langle\rangle \triangleleft (I_a-I_c)-(I_b-I_c)}}{\varphi;\Phi;a:\mathsf{out}_{I_a-I_c}(),b:\mathsf{in}_{I_b-I_c}(),c:\mathsf{in}_0() \vdash_s b().\overline{a}\langle\rangle \triangleleft I_a-I_c}}{\varphi;\Phi;\langle(a:\mathsf{out}_{I_a}(),b:\mathsf{in}_{I_b}(),c:\mathsf{in}_{I_c}())\rangle_{-I_c} \vdash_s b().\overline{a}\langle\rangle \triangleleft I_a-I_c}}{\varphi;\Phi;a:\mathsf{out}_{I_a}(),b:\mathsf{in}_{I_b}(),c:\mathsf{in}_{I_c}() \vdash_s c().b().\overline{a}\langle\rangle \triangleleft I_a.}$$

The constraints $\varphi;\Phi \vDash I_a \geq I_b \geq I_c$ are necessary in order for the various $\langle\cdot\rangle_{-I}$ to be defined. So, to sum up, the complexity of such a sequence of input/output is given by the time of the last

channel, and the constraints that appear are that the order of time should correspond to the order of the process sequence.

Another interesting case of this is when a is a server type and not a simple channel type: let us take $a : \mathtt{oserv}^{K_a}_{I_a}()$ in the example above. As an output capability for a server type is never erased by $\langle\cdot\rangle_{-I}$, we have fewer restrictions. In particular, we only need $\varphi;\Phi \vDash I_b \geq I_c$ and the complexity becomes $K_a{+}I_b{+}(I_a{-}I_b)$. Recall that this is not always equal to $K_a{+}I_a$ by definition of subtraction; for example, if $I_a = 0$, then we have the complexity $K_a{+}I_b$ in the end.

Finally, one can remark that if we add a `tick` constructor to this process, then the only thing it does is changing the shape of constraints. For example, if $P = \mathtt{tick}.c().\mathtt{tick}.b().\overline{a}\langle\rangle$ and a is a simple channel, then the total complexity is still I_a but the constraints on I_a, I_b and I_c become

$$\varphi;\Phi \vDash I_c \geq 1, \qquad \varphi;\Phi \vDash I_b \geq I_c{+}1, \qquad \varphi;\Phi \vDash I_a \geq I_b.$$

5.3.2 Fibonacci. Let us consider again the process P described in Figure 3. We set $G(n) = \max(1, n)$, corresponding to the function defined in Section 2, and we want to show that the server fib has a span $G(n)$. To do this, we give the typing described in Figure 13, which we detail here. This typing is similar to the one of Figure 8(b), so we detail especially the differences.

The first rule of this typing is already quite different. Indeed, in the span type system, the (Iserv) rules has a condition of time invariance on the context of the continuation. Because of this, we have to transform (with subtyping) the initial context Θ defining the servers add and fib into the time invariant context Θ', where those two servers only have output capabilities. Then, as before, we use the fact that $i;\cdot \vDash G(i) \geq 1$ for all i to rewrite it $G(i){-}1{+}1$ to do the (Tick) rule. Then, the (Tick) rule makes the time advance in the context Γ. We obtain

$$\langle\Gamma\rangle_{-1} = \Theta', n : \mathrm{Nat}[i], a : \mathrm{out}_{G(i)-1}(\mathrm{Nat}[Fib(i)]),$$

again relying on the fact that $i;\cdot \vDash G(i) \geq 1$. Afterwards, we have the two pattern matchings, and we do not detail the zero cases, as they are simple. Intuitively, with the constraints $(i \leq 0)$ or $(i \geq 1, i{-}1 \leq 0)$, we have $G(i){-}1 = 0$, so the type of a says that the channel has to send immediately, which it does. We can then focus on the main branch, where we have the constraints Φ equivalent to $i \geq 2$.

Then, after some (Nu) rules and the (Par) rule, we have to type the three premises, and all with the same complexity $G(i){-}1$. For the first premise, we use $i;\Phi \vDash G(i{-}1) \leq G(i){-}1$ to change the complexity with a (Sub) rule. Then, we use polymorphism to show that this recursive call $\overline{fib}\langle m, b\rangle$ is done on a size $i{-}1$ to obtain the complexity $G(i{-}1)$. We have a similar behaviour for the second recursive call, with size $i{-}2$. Finally, the third premise corresponds to a usual pattern described in Section 5.3.1: the complexity of this subprocess is the time of b, which is $G(i{-}1)$. And the condition $i;\Phi;G(i{-}1) \geq G(i{-}2)$ is satisfied.

5.3.3 An Example to Justify the Use of Time. To justify the use of time in types for span, and to show how we could find the time of a channel, we present here three examples of recursive calls with different behaviour. Usually, type inference for a size system reduces to satisfying a set of constraints on indices. We believe that even with time indexes on channels, type inference is still reducible to satisfying such a set of constraints. So, for the sake of simplicity, we will describe this example with constraints. We define three processes P_1, P_2, and P_3 by

$$P_l \equiv !a(n, r).\mathtt{tick}.\mathtt{match}\ n\ \{\mathtt{0} \mapsto \overline{r}\langle\rangle;;\ \mathtt{s}(m) \mapsto (\nu r')(\nu r'')(Q_l)\},$$

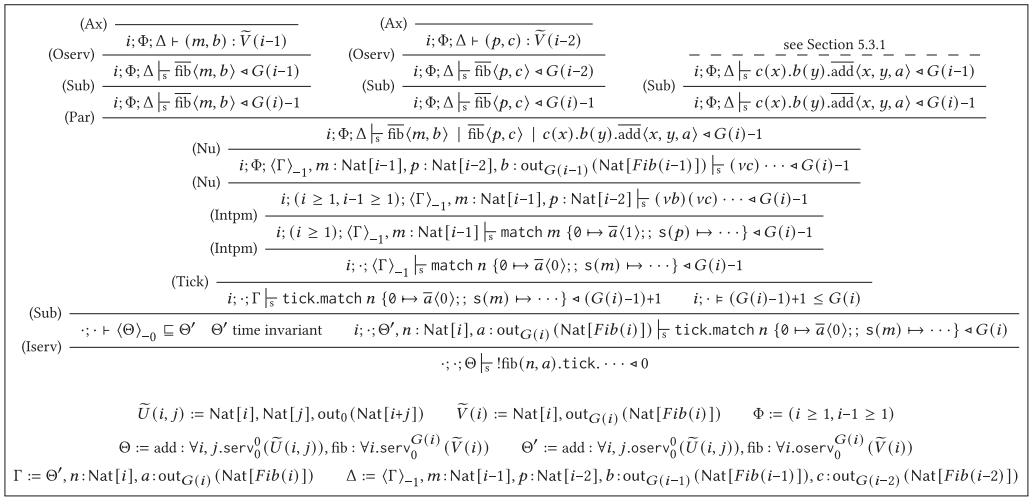


Fig. 13. A typing for Fibonacci (span).

for the following definition of Q_l:

$$Q_1 \equiv \overline{a}\langle m, r'\rangle \mid \overline{a}\langle m, r''\rangle \mid r'().r''().\overline{r}\langle\rangle,$$
$$Q_2 \equiv \overline{a}\langle m, r'\rangle \mid r'().\overline{a}\langle m, r''\rangle \mid r''().\overline{r}\langle\rangle,$$
$$Q_3 \equiv \overline{a}\langle m, r'\rangle \mid r'().(\overline{a}\langle m, r''\rangle \mid \overline{r}\langle\rangle) \mid r''().\mathbb{0}.$$

Intuitively, for P_1 the two recursive calls are done after one unit of time in parallel, and the return signal on r is done when both processes have done their return signal on r' and r''. So, this is total parallelism for the two recursive calls (the span is linear in n). For P_2, a first recursive call is done, and then the process waits for the return signal on r', and when it receives it, the second recursive call begins. So, this is totally sequential (the span is exponential in n). Finally, for P_3, we have an intermediate situation between totally parallel and totally sequential. The process starts with a recursive call. Then, it waits for the return signal on r'. When this signal arrives, it immediately starts the second recursive call and immediately does the return signal on r. So, intuitively, the second recursive call starts when all the "left" calls have been done. Note that those three servers have the same work, which is exponential in n.

So, let us type the three examples with the type system for span. For the sake of simplicity, we omit the typing of expressions, we only consider the difficult branch for the match constructors, and we focus on complexity and time. We consider the following context that is used for the three processes:

$$\Gamma \equiv a : \forall i.\mathsf{oserv}_0^{f(i)}(\mathsf{Nat}[0, i], \mathsf{ch}_{g(i)}()), n : \mathsf{Nat}[0, i], r : \mathsf{ch}_{g(i)}().$$

We have two unknown function symbols: f, which represents the complexity of the server, and g, the time for the return channel. We also use this second context:

$$\Delta \equiv \langle\Gamma\rangle_{-1}, m : \mathsf{Nat}[0, i{-}1], r' : \mathsf{ch}_{g'(i)}(), r'' : \mathsf{ch}_{g''(i)}().$$

This gives two more unknown functions, g' and g'' corresponding, respectively, to the time of r' and r'' when defined. The three processes start with the same typing:

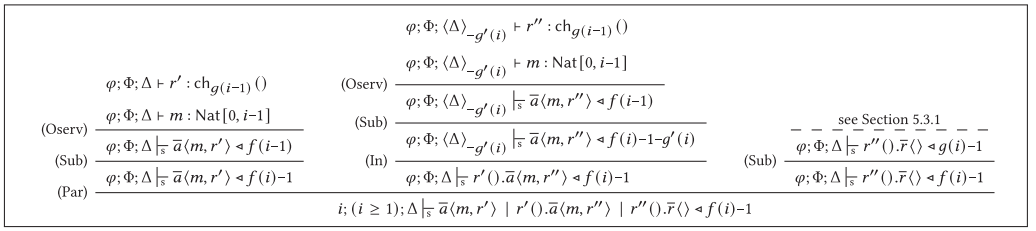


Fig. 14. A typing for Q_2.

$$\dfrac{\dfrac{\dfrac{i;\cdot \vDash f(i) \geq g(i)}{i;\cdot;\langle\Gamma\rangle_{-1} \mid\!\!\vdash_{\mathsf{s}} \overline{r}\langle\rangle \triangleleft f(i){-}1} \qquad i;i \geq 1;\Delta \mid\!\!\vdash_{\mathsf{s}} Q_l \triangleleft f(i){-}1}{i;\cdot;\langle\Gamma\rangle_{-1} \mid\!\!\vdash_{\mathsf{s}} \mathtt{match}\ n\ \{\mathtt{0} \mapsto \overline{r}\langle\rangle;;\ \mathsf{s}(m) \mapsto (\nu r')(\nu r'')(Q_l)\} \triangleleft f(i){-}1}\ (\text{Intpm})}{i;\cdot;\Gamma \mid\!\!\vdash_{\mathsf{s}} \mathtt{tick.match}\ n\ \{\mathtt{0} \mapsto \overline{r}\langle\rangle;;\ \mathsf{s}(m) \mapsto (\nu r')(\nu r'')(Q_l)\} \triangleleft f(i)}\ (\text{Tick})}{\cdot;\cdot;a : \forall i.\mathsf{serv}_0^{f(i)}(\mathsf{Nat}[0,i],\mathsf{ch}_{g(i)}()) \mid\!\!\vdash_{\mathsf{s}} P_l \triangleleft 0.}\ (\text{Iserv})$$

Because of the `tick`, we know that the complexity on the line above the root should have the shape $K{+}1$ for some K, so here we obtain immediately that $f(i) \geq 1$. In the same way, r should still be defined in $\langle\Gamma\rangle_{-1}$, hence we obtain $g(i) \geq 1$.

We now describe the various constraints obtained on the three processes, under the assumption that $i \geq 1$. For Q_1, we obtain a typing similar to Figure 13 for Fibonacci, and we derive the following constraints:

$$f(i){-}1 \geq f(i{-}1), \qquad g'(i) = g(i{-}1), \qquad g''(i) = g(i{-}1), \qquad g(i){-}1 \geq g''(i) \geq g'(i), \qquad f(i){-}1 \geq g(i){-}1.$$

The first constraint is because the total complexity $f(i){-}1$ must be greater than the complexity of the two recursive calls $f(i{-}1)$. Then, r' and r'' must have a time equal to $g(i{-}1)$ to correspond to the type of a in the outputs $\overline{a}\langle m, r'\rangle$ and $\overline{a}\langle m, r''\rangle$. Finally, the last two constraints correspond to the constraints of Section 5.3.1 and the fact that the complexity bound $f(i){-}1$ should be greater than the complexity of this subprocess, which is $g(i){-}1$, again, as in Section 5.3.1. So, we can satisfy the conditions with the following choice:

$$f(i) \equiv i{+}1, \qquad g(i) \equiv i{+}1, \qquad g'(i) \equiv g''(i) \equiv i.$$

So, as expected, the span, represented by the function f, is indeed linear.

Then, for Q_2, we describe the typing in Figure 14, where $\varphi;\Phi \equiv i;(i \geq 1)$.

Thus, we obtain the following constraints:

$$f(i){-}1 \geq f(i{-}1), \qquad g'(i) = g(i{-}1), \qquad f(i){-}1{-}g'(i) \geq f(i{-}1),$$
$$g''(i){-}g'(i) = g(i{-}1), \qquad g(i){-}1 \geq g''(i), \qquad f(i){-}1 \geq g(i){-}1.$$

The constraints on $f(i)$ are obtained because of the subtyping rules, expressed by the double lines in the type derivation. The equality constraints are obtained, because we need both r' and r'' to have the type $\mathsf{ch}_{g(i-1)}()$, and finally the constraint $g(i){-}1 \geq g''(i)$ is as in Section 5.3.1. Thus, we can take

$$f(i) \equiv 2^{i+1}{-}1, \qquad g(i) \equiv 2^{i+1}{-}1.$$

So, we indeed obtain the exponential complexity.

However, with those two examples, the time of the channel r is always equal to the complexity of the server a, and we cannot really see the usefulness of time. Still, with the next example, we obtain something more interesting. A type derivation for Q_3 is given in Figure 15.

$$
\text{(Par)}\ \dfrac{\cdots \qquad \text{(In)}\ \dfrac{\text{(Par)}\ \dfrac{\cdots \qquad \text{(Sub)}\ \dfrac{\text{(Out)}\ \dfrac{}{\varphi;\Phi;\langle\Delta\rangle_{-g'(i)} \vdash_{\mathrm{s}} \overline{r}\langle\rangle \triangleleft g(i){-}1{-}g'(i)}}{\varphi;\Phi;\langle\Delta\rangle_{-g'(i)} \vdash_{\mathrm{s}} \overline{r}\langle\rangle \triangleleft f(i){-}1{-}g'(i)}}{\varphi;\Phi;\langle\Delta\rangle_{-g'(i)} \vdash_{\mathrm{s}} \overline{a}\langle m,r''\rangle \mid \overline{r}\langle\rangle \triangleleft f(i){-}1{-}g'(i)}}{\varphi;\Phi;\Delta \vdash_{\mathrm{s}} r'().(\overline{a}\langle m,r''\rangle \mid \overline{r}\langle\rangle) \triangleleft f(i){-}1} \qquad \text{(Sub)}\ \dfrac{\text{(In)}\ \dfrac{\text{(Pzero)}\ \dfrac{}{\varphi;\Phi;\langle\Delta\rangle_{-g''(i)} \vdash_{\mathrm{s}} \mathbb{0} \triangleleft 0}}{\varphi;\Phi;\Delta \vdash_{\mathrm{s}} r''().\mathbb{0} \triangleleft g''(i)}}{\varphi;\Phi;\Delta \vdash_{\mathrm{s}} r''().\mathbb{0} \triangleleft f(i){-}1}}{i;(i \geq 1);\Delta \vdash_{\mathrm{s}} \overline{a}\langle m,r'\rangle \mid r'().(\overline{a}\langle m,r''\rangle \mid \overline{r}\langle\rangle) \mid r''().\mathbb{0} \triangleleft f(i){-}1}
$$

Fig. 15. A typing for Q_3.

We obtain the following constraints:

$$
f(i){-}1 \geq f(i{-}1), \qquad g'(i) = g(i{-}1), \qquad f(i){-}1{-}g'(i) \geq f(i{-}1),
$$

$$
g''(i){-}g'(i) = g(i{-}1), \qquad g(i){-}1 \geq g'(i), \qquad f(i){-}1{-}g'(i) \geq g(i){-}1{-}g'(i), \qquad f(i){-}1 \geq g''(i).
$$

The first four constraints are exactly the same as before, because there are the same typing derivations (represented by $\cdots$). Then, the constraint on $g(i){-}1$ is because of $\langle\cdot\rangle_{-g'(i)}$, and the other constraints come from subtyping rules. So, using the equalities, and by removing redundant inequalities, we obtain for f and g,

$$
f(i) \geq 1{+}g(i{-}1){+}f(i{-}1), \qquad g(i) \geq 1{+}g(i{-}1), \qquad f(i) \geq 1{+}2 \cdot g(i{-}1).
$$

Thus, we can take

$$
g(i) \equiv i{+}1, \qquad f(i) \equiv \frac{(i{+}1)(i{+}2)}{2}.
$$

The complexity is quadratic in n. Note that for this example, the complexity f depends directly on g, and g is given by a recursive equation independent of f. In a sense, to find the complexity, we need to find first the delay of the second recursive call. Without time indications on channel, it would not be possible to track and obtain this recurrence relation on g, and thus we could not deduce the complexity.

Note that the two first examples used channels as a return signal for a parallel computation, whereas for the last example, channels are used as a synchronization point in the middle of a computation. We believe that this flexibility of channels justifies the use of π-calculus to reason about parallel computation. Moreover, this work is a step to a more expressive type system inspired by Reference [30], taking in account concurrent behaviour. Indeed, as we will show, the current type system fails to capture some simple concurrency.

5.3.4 Limitations of the Type System. Our current type system enforces some kind of time uniqueness in channels. Indeed, take the process $a().\mathtt{tick}.\overline{a}\langle\rangle$. When trying to type this process, we obtain

$$
\text{(In)}\ \dfrac{\text{(Sub)}\ \dfrac{\cdot;\cdot \vdash \mathsf{ch}_I() \sqsubseteq \mathsf{in}_I()}{\cdot;\cdot;a : \mathsf{ch}_I() \vdash a : \mathsf{in}_I()} \qquad \text{(Tick)}\ \dfrac{\dfrac{\text{Error}}{\cdot;\cdot;\langle a : \mathsf{ch}_0()\rangle_{-1} \vdash_{\mathrm{s}} \overline{a}\langle\rangle \triangleleft 0}}{\cdot;\cdot;a : \mathsf{ch}_0() \vdash_{\mathrm{s}} \mathtt{tick}.\overline{a}\langle\rangle \triangleleft 1}}{\cdot;\cdot;a : \mathsf{ch}_I() \vdash_{\mathrm{s}} a().\mathtt{tick}.\overline{a}\langle\rangle \triangleleft I{+}1.}
$$

As by definition $\langle a : \mathsf{ch}_0()\rangle_{-1}$ is $\emptyset$, we cannot type the output on a. So, channels have strong constraints on the time they can be used. This is true especially when channels are not used linearly. Still, note that we can type a process of the shape $a().\mathbb{0} \mid \overline{a}\langle\rangle \mid \mathtt{tick}.\overline{a}\langle\rangle$, thus it is better than plain linearity on channels. This restriction limits examples of concurrent behaviours. For example, take two processes P_1 and P_2 that should be executed but not simultaneously. To do

that in a concurrent setting, we can use semaphores. In π-calculus, we could consider the process $(va)(a().P_1' \mid a().P_2' \mid \overline{a}\langle\rangle)$, where P_1' is P_1 with an output $\overline{a}\langle\rangle$ at the end, likewise for P_2'. This is a way to simulate a semaphore in π-calculus. Now, we can see that this example has the same problem as the example given above if, for instance, P_1 contains a `tick`, thus we cannot type this kind of process. Formally, this is because of our parallel composition rule:

$$(\text{Par})\ \frac{\varphi;\Phi;\Gamma \vdash_{\text{s}} P \triangleleft K \qquad \varphi;\Phi;\Gamma \vdash_{\text{s}} Q \triangleleft K}{\varphi;\Phi;\Gamma \vdash_{\text{s}} P \mid Q \triangleleft K.}$$

If we take Q equal to P, then we then obtain, from a typing $\varphi;\Phi;\Gamma \vdash_{\text{s}} P \triangleleft K$ a typing $\varphi;\Phi;\Gamma \vdash_{\text{s}} P \mid P \triangleleft K$. So, the type system considers that P and $P \mid P$ are equivalent, and this is obviously not true in general, especially with the example described above. However, in a linear setting, this is not a problem.

Still, we believe that for parallel computation, our type system should be quite expressive in practice. Indeed, as stated above, the restriction appears especially when channels are not used linearly. However, it is known that linear π-calculus in itself is expressive for parallel computation [34]. For example, classical encodings of functional programs in a parallel setting rely on the use of linear return signals, as we will see in the example for bitonic sort in Section 6. Moreover, session types can also be encoded in linear π-calculus in the presence of variant types [11, 31]. Note that to encode a calculus as the one in Reference [12], we would also need recursive types. Our calculus and its proof of soundness could be extended to variant types, but not straightforwardly to recursive types. However, we believe the results on the linear π-calculus we cited suggest that the restriction given above should not be too harmful for parallel computation.

5.4 Complexity Results

In this section, we show how to prove that our type system indeed gives a bound on the number of time reduction steps of a process. As we work with the reduction relation $\Rightarrow$ of Figure 11, we need to consider annotated processes instead of simple processes. So, we need to enrich our type system with a rule for the constructor $n : P$:

$$\frac{\varphi;\Phi;\langle\Gamma\rangle_{-n} \vdash_{\text{s}} P \triangleleft K}{\varphi;\Phi;\Gamma \vdash_{\text{s}} n : P \triangleleft K+n}.$$

As the intuition suggested, this rule is equivalent to n times the typing rule for `tick`. We can now work on the properties of our type system on annotated processes.

The procedure to prove the subject reduction for $\Rightarrow$ in this type system is intrinsically more difficult than the one for Theorem 4.6. So, from the proof of subject reduction for span, one could deduce the proof of subject reduction for work, just by forgetting the considerations with time and the constructor $n : P$ in the following proof.

5.4.1 Intermediate Lemmas. We first show some intermediate lemmas on the typing system. To begin with, we give a lemma on the link between subtyping and time advance.

Lemma 5.10. *If $\varphi;\Phi \vdash T \sqsubseteq U$, then for any I, either $\langle U\rangle_{-I}$ is undefined, or both $\langle U\rangle_{-I}$ and $\langle T\rangle_{-I}$ are defined, and $\varphi;\Phi \vdash \langle T\rangle_{-I} \sqsubseteq \langle U\rangle_{-I}$.*

Proof. The proof is by induction on the subtyping derivation. First, transitivity is direct by induction hypothesis. For non-channel type, this is direct, because time advancing does not do anything. Then, for channels that are not servers, time is invariant by subtyping, thus time advancing is either undefined for both types, either defined for both types, with the same time. For

a server type, again time is invariant by subtyping, so either both types lose their input capability, either they both keep the same capabilities. The case where $\langle U \rangle_{-I}$ is undefined and not $\langle T \rangle_{-I}$ is when T is an input/output server that loses its input capability, and U is an input server. □

Now, for the usual properties of typing systems, we have first weakening and strengthening.

Lemma 5.11 (Weakening). *Let φ, φ' be disjoint sets of index variables, Φ be a set of constraints on φ, Φ' be a set of constraints on (φ, φ'), Γ and Γ' be contexts on disjoint set of variables.*

(1) *If $\varphi; \Phi \vDash C$, then $(\varphi, \varphi'); (\Phi, \Phi') \vDash C$.*
(2) *If $\varphi; \Phi \vdash T \sqsubseteq U$, then $(\varphi, \varphi'); (\Phi, \Phi') \vdash T \sqsubseteq U$.*
(3) *If $\varphi; \Phi; \Gamma \vdash e : T$, then $(\varphi, \varphi'); (\Phi, \Phi'); \Gamma, \Gamma' \vdash e : T$.*
(4) $\langle \Gamma \rangle_{-I}^{(\varphi,\varphi');(\Phi,\Phi')} = \Delta, \Delta'$ *for some* Δ, Δ' *with* $(\varphi; \varphi'); (\Phi; \Phi') \vdash \Delta \sqsubseteq \langle \Gamma \rangle_{-I}^{\varphi;\Phi}$.
(5) *If $\varphi; \Phi; \Gamma \vdash_s P \triangleleft K$, then $(\varphi, \varphi'); (\Phi, \Phi'); \Gamma, \Gamma' \vdash_s P \triangleleft K$.*

Proof. Point 1 is a direct consequence of the definition of $\varphi; \Phi \vDash C$. Point 2 is proved by induction on the subtyping derivation, and it uses explicitly Point 1. Point 4 is a consequence of Point 1: Everything that is defined in $\langle \Gamma \rangle_{-I}^{\phi;\Phi}$ is also defined in $\langle \Gamma \rangle_{-I}^{(\phi,\phi');(\Phi,\Phi')}$, and the subtyping condition is here, since with more constraints, a server may not be changed into an output server by the advance of time. Points 3 and 5 are proved by induction on the typing derivation, and each point uses crucially the previous ones. Note that the weakening integrated in the rule for input servers is necessary to obtain Point 5. Note also that when the advance time operator is used, the weakened typing is obtained with the use of a subtyping rule. □

Formally, all lemmas should be presented in the same way, in the sense that a property such as weakening should hold first on constraints, then we can deduce it for subtyping, and finally we can deduce it for typing. However, for the sake of simplicity, we chose for the following lemmas to only present the interesting points.

Lemma 5.12 (Strengthening). *Let φ be a set of index variables, Φ be a set of constraints on φ, and C a constraint on φ such that $\varphi; \Phi \vDash C$.*

(1) $\langle \Gamma \rangle_{-I}^{\varphi;(\Phi,C)} = \langle \Gamma \rangle_{-I}^{\varphi;\Phi}$.
(2) *If $\varphi; (\Phi, C); \Gamma, \Gamma' \vdash_s P \triangleleft K$ and the variables in Γ' are not free in P, then $\varphi; \Phi; \Gamma \vdash_s P \triangleleft K$.*

We also have a property specific to size type systems, expressing that an index variable can be substituted by any index.

Lemma 5.13 (Index Substitution). *Let φ be a set of index variables and $i \notin \varphi$. Let J be an index with free variables in φ. Then,*

(1) $[\![I\{J/i\}]\!]_\rho = [\![I]\!]_{\rho[i \mapsto [\![J]\!]_\rho]}$.
(2) *If $(\varphi, i); \Phi; \Gamma \vdash_s P \triangleleft K$, then $\varphi; \Phi\{J/i\}; \Gamma\{J/i\} \vdash_s P \triangleleft K\{J/i\}$.*

We also need a lemma specific to the notion of time.

Definition 5.14 (Delaying). Given a type T and an index I, we define the *delaying* of T by I units of time, denoted $\langle T \rangle_{+I}$:

$$\langle \mathcal{B} \rangle_{+I} = \mathcal{B}, \qquad \langle \mathsf{ch}_J(\widetilde{T}) \rangle_{+I} = \mathsf{ch}_{J+I}(\widetilde{T}),$$

and for other channel and server types, the definition is in correspondence with the one on the right above. This definition can be extended to contexts.

Lemma 5.15 (Delaying). *For any index I, we have:*

(1) $\langle\langle\Gamma\rangle_{+I}\rangle_{-J} = \Delta, \Delta'$ *with* $\varphi; \Phi \vdash \Delta \sqsubseteq \langle\langle\Gamma\rangle_{-J}\rangle_{+I}$.
(2) $\langle\langle\Gamma\rangle_{+I}\rangle_{-(J+I)} = \langle\Gamma\rangle_{-J}$.
(3) *If* $\varphi; \Phi; \Gamma \models_s P \triangleleft K$, *then* $\varphi; \Phi; \langle\Gamma\rangle_{+I} \models_s P \triangleleft K{+}I$.

With this lemma, we can see that if we add a delay of I time units in the contexts for all channels, it increases the complexity by I time units, thus we see the link between time in types and the complexity. Then, we can show the usual substitution lemma.

Lemma 5.16 (Substitution). *We have:*

(1) *If* $\varphi; \Phi; \Gamma, v : T \vdash e' : U$ *and* $\varphi; \Phi; \Gamma \vdash e : T$, *then* $\varphi; \Phi; \Gamma \vdash e'[v := e] : U$.
(2) *If* $\varphi; \Phi; \Gamma, v : T \models_s P \triangleleft K$ *and* $\varphi; \Phi; \Gamma \vdash e : T$, *then* $\varphi; \Phi; \Gamma \models_s P[v := e] \triangleleft K$.

The proof is standard.

5.4.2 Subject Reduction. We now present the core theorem to have the complexity soundness: subject reduction. First, we need to show that typing is invariant by congruence.

Lemma 5.17 (Congruence and Typing). *Let* P *and* Q *be annotated processes such that* $P \equiv Q$. *Then,* $\varphi; \Phi; \Gamma \models_s P \triangleleft K$ *if and only if* $\varphi; \Phi; \Gamma \models_s Q \triangleleft K$.

Proof. We prove this by induction on $P \equiv Q$. Note that for a process P, the typing system is not syntax-directed because of the subtyping rule. However, by reflexivity and transitivity of subtyping, we can always assume that a proof has exactly *one* subtyping rule before any syntax-directed rule. Moreover, it is sufficient to prove this theorem for type derivations that do not start with a subtyping rule. Indeed, the first subtyping rule in a derivation for P can always be mimicked as it is in the derivation for Q. We show the essential base cases and the inductive steps follow directly from induction hypothesis.

- **Case** $(va)P \mid Q \equiv (va)(P \mid Q)$ with a not free in Q.
  Suppose $\varphi; \Phi; \Gamma \models_s (va)(P \mid Q) \triangleleft K$. Then the proof has the shape:

$$\dfrac{\dfrac{\dfrac{\dfrac{\pi_P}{\varphi; \Phi; \Delta, a : T' \models_s P \triangleleft K'} \quad \dfrac{\pi_Q}{\varphi; \Phi; \Delta, a : T' \models_s Q \triangleleft K'}}{\varphi; \Phi; \Delta, a : T' \models_s P \mid Q \triangleleft K'} \quad \varphi; \Phi \vdash \Gamma \sqsubseteq \Delta \quad \varphi; \Phi \vdash T \sqsubseteq T' \quad \varphi; \Phi \vDash K' \leq K}{\varphi; \Phi; \Gamma, a : T \models_s P \mid Q \triangleleft K}}{\varphi; \Phi; \Gamma \models_s (va)(P \mid Q) \triangleleft K.}$$

  Since a is not free in Q, by Lemma 5.12, from π_Q, we obtain a proof π'_Q of $\varphi; \Phi; \Delta \models_s Q \triangleleft K'$. We can then derive the following typing:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\pi_P}{\varphi; \Phi; \Delta, a : T' \models_s P \triangleleft K'} \quad \varphi; \Phi \vdash T \sqsubseteq T'}{\varphi; \Phi; \Delta, a : T \models_s P \triangleleft K'}}{\varphi; \Phi; \Delta \models_s (va)P \triangleleft K'} \quad \dfrac{\pi'_Q}{\varphi; \Phi; \Delta \models_s Q \triangleleft K'}}{\varphi; \Phi; \Delta \models_s (va)P \mid Q \triangleleft K'} \quad \varphi; \Phi \vdash \Gamma \sqsubseteq \Delta \quad \varphi; \Phi \vDash K' \leq K}{\varphi; \Phi; \Gamma \models_s (va)P \mid Q \triangleleft K.}$$

  For the converse, the proof is direct by induction hypothesis and weakening.

- **Case** $m : (P \mid Q) \equiv m : P \mid m : Q$. Suppose $\varphi; \Phi; \Gamma \vdash_s m : (P \mid Q) \triangleleft K{+}m$. Then we have

$$\dfrac{\dfrac{\dfrac{\dfrac{\pi_P}{\varphi;\Phi;\Delta \vdash_s P \triangleleft K'} \quad \dfrac{\pi_Q}{\varphi;\Phi;\Delta \vdash_s Q \triangleleft K'}}{\varphi;\Phi;\Delta \vdash_s P \mid Q \triangleleft K'} \quad \varphi;\Phi \vdash \langle\Gamma\rangle_{-m} \sqsubseteq \Delta \quad \varphi;\Phi \vDash K' \leq K}{\varphi;\Phi;\langle\Gamma\rangle_{-m} \vdash_s P \mid Q \triangleleft K}}{\varphi;\Phi;\Gamma \vdash_s m : (P \mid Q) \triangleleft K{+}m.}$$

  So, we can give the following derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{\pi_P}{\varphi;\Phi;\Delta \vdash_s P \triangleleft K'} \quad \varphi;\Phi \vdash \langle\Gamma\rangle_{-m} \sqsubseteq \Delta \quad \varphi;\Phi \vDash K' \leq K}{\varphi;\Phi;\langle\Gamma\rangle_{-m} \vdash_s P \triangleleft K}}{\varphi;\Phi;\Gamma \vdash_s m : P \triangleleft K{+}m} \quad \dfrac{\dfrac{\dfrac{\pi_Q}{\varphi;\Phi;\Delta \vdash_s Q \triangleleft K'}}{\varphi;\Phi;\langle\Gamma\rangle_{-m} \vdash_s \triangleleft K}}{\varphi;\Phi;\Gamma \vdash_s m : Q \triangleleft K{+}m}}{\varphi;\Phi;\Gamma \vdash_s m : P \mid m : Q \triangleleft K{+}m.}$$

  Now, suppose we have a typing $\varphi; \Phi; \Gamma \vdash_s m : P \mid m : Q \triangleleft K$. The typing has the shape

$$\dfrac{\dfrac{\dfrac{\dfrac{\pi_P}{\varphi;\Phi;\langle\Delta_1\rangle_{-m} \vdash_s P \triangleleft K_1}}{\varphi;\Phi;\Delta_1 \vdash_s m : P \triangleleft K_1{+}m} \quad \dfrac{\dfrac{\pi_Q}{\varphi;\Phi;\langle\Delta_2\rangle_{-m} \vdash_s Q \triangleleft K_2}}{\varphi;\Phi;\Delta_2 \vdash_s m : Q \triangleleft K_2{+}m} \quad \varphi;\Phi \vdash \Gamma \sqsubseteq \Delta_1;\Gamma \sqsubseteq \Delta_2; K_1{+}m \leq K; K_2{+}m \leq K}{\varphi;\Phi;\Gamma \vdash_s m : P \triangleleft K \quad \varphi;\Phi;\Gamma \vdash_s m : Q \triangleleft K}}{\varphi;\Phi;\Gamma \vdash_s m : P \mid m : Q \triangleleft K.}$$

  By Lemma 5.10, from $\Gamma \sqsubseteq \Delta_1$, we obtain $\Gamma = \Theta_0, \Theta_1$ with $\langle\Theta_1\rangle_{-m} \sqsubseteq \langle\Delta_1\rangle_{-m}$. In the same way, $\Gamma = \Theta'_0, \Theta'_1$ with $\langle\Theta'_1\rangle_{-m} \sqsubseteq \langle\Delta_2\rangle_{-m}$. Moreover, we have easily $\varphi; \Phi \vDash K \geq m$. Thus, by the Lemma 5.11 (weakening) for π_P and π_Q, we obtain

$$\dfrac{\dfrac{\dfrac{\dfrac{\pi_P^w}{\varphi;\Phi;\langle\Theta_0\rangle_{-m}, \langle\Delta_1\rangle_{-m} \vdash_s P \triangleleft K_1}}{\varphi;\Phi;\langle\Gamma\rangle_{-m} \vdash_s P \triangleleft K{-}m} \quad \dfrac{\dfrac{\dfrac{\pi_Q^w}{\varphi;\Phi;\langle\Theta'_0\rangle_{-m}, \langle\Delta_2\rangle_{-m} \vdash_s Q \triangleleft K_2}}{\varphi;\Phi;\langle\Gamma\rangle_{-m} \vdash_s Q \triangleleft K{-}m}}{\varphi;\Phi;\langle\Gamma\rangle_{-m} \vdash_s P \mid Q \triangleleft K{-}m}}{\varphi;\Phi;\Gamma \vdash_s m : (P \mid Q) \triangleleft K.}$$

  This concludes this case.

- **Case** $m : (\nu a)P \equiv (\nu a)(m : P)$.
  Suppose that $\varphi; \Phi; \Gamma \vdash_s m : (\nu a)P \triangleleft K{+}m$. Then, the proof has the shape

$$\dfrac{\dfrac{\dfrac{\dfrac{\pi_P}{\varphi;\Phi;\Delta, a : T \vdash_s P \triangleleft K'}}{\varphi;\Phi;\Delta \vdash_s (\nu a)P \triangleleft K'} \quad \varphi;\Phi \vdash \langle\Gamma\rangle_{-m} \sqsubseteq \Delta \quad \varphi;\Phi \vDash K' \leq K}{\varphi;\Phi;\langle\Gamma\rangle_{-m} \vdash_s (\nu a)P \triangleleft K}}{\varphi;\Phi;\Gamma \vdash_s m : (\nu a)P \triangleleft K{+}m.}$$

  Recall that, by Lemma 5.15, $\langle\langle T\rangle_{+m}\rangle_{-m} = \langle T\rangle_{-0} = T$. So, we have

$$\dfrac{\dfrac{\dfrac{\dfrac{\pi_P}{\varphi;\Phi;\Delta, a : T \vdash_s P \triangleleft K'} \quad \varphi;\Phi \vdash \langle\Gamma\rangle_{-m} \sqsubseteq \Delta \quad \varphi;\Phi \vDash K' \leq K \quad \varphi;\Phi \vdash T \sqsubseteq T}{\varphi;\Phi;\langle\Gamma\rangle_{-m}, a : T \vdash_s P \triangleleft K}}{\varphi;\Phi;\Gamma, a : \langle T\rangle_{+m} \vdash_s m : P \triangleleft K{+}m}}{\varphi;\Phi;\Gamma \vdash_s (\nu a)(m : P) \triangleleft K{+}m.}$$

For the converse, suppose that $\varphi;\Phi;\Gamma \vdash_s (\nu a)m : P \triangleleft K$. Then the typing has the shape

$$\dfrac{\dfrac{\dfrac{\pi_P}{\varphi;\Phi;\langle(\Delta,\, a:T')\rangle_{-m} \vdash_s P \triangleleft K'}}{\varphi;\Phi;\Delta,\, a:T' \vdash_s m : P \triangleleft K'+m} \qquad \varphi;\Phi \vdash \Gamma \sqsubseteq \Delta \qquad \varphi;\Phi \vdash T \sqsubseteq T' \qquad \varphi;\Phi \vDash K'+m \leq K}{\dfrac{\varphi;\Phi;\Gamma,\, a:T \vdash_s m : P \triangleleft K}{\varphi;\Phi;\Gamma \vdash_s (\nu a)m : P \triangleleft K.}}$$

By an abuse of notation, let us write $\langle T'\rangle_{-m}$ to denote $\langle T'\rangle_{-m}$ if it is defined, and any other type otherwise. Then, we have the derivation (with possibly a weakened version of π_P)

$$\dfrac{\dfrac{\dfrac{\dfrac{\pi_P^w}{\varphi;\Phi;\langle\Delta\rangle_{-m},\, a:\langle T'\rangle_{-m} \vdash_s P \triangleleft K'}}{\varphi;\Phi;\langle\Delta\rangle_{-m} \vdash_s (\nu a)P \triangleleft K'}}{\varphi;\Phi;\Delta \vdash_s m : (\nu a)P \triangleleft K'+m} \qquad \varphi;\Phi \vdash \Gamma \sqsubseteq \Delta \qquad \varphi;\Phi \vDash K'+m \leq K}{\varphi;\Phi;\Gamma \vdash_s m : (\nu a)P \triangleleft K.}$$

This concludes all the base cases. We can then prove the lemma by considering the remaining cases for $P \equiv Q$. Symmetry, transitivity and contextual congruence are direct by induction hypothesis. □

Now that we have Lemma 5.17, we can work up to the congruence relation. So, we proceed to show subject reduction.

Theorem 5.18 (Subject Reduction). *If* $\varphi;\Phi;\Gamma \vdash_s P \triangleleft K$ *and* $P \Rightarrow Q$, *then* $\varphi;\Phi;\Gamma \vdash_s Q \triangleleft K$.

Let us show this Theorem. We do this by induction on $P \Rightarrow Q$. Let us first remark that when considering the typing of P, again the first subtyping rule has no importance, since we can always start the typing of Q with the exact same subtyping rule. We now proceed by doing the case analysis on the rules of Figure 11. Again, we only present the interesting cases.

- **Case** $(n :!a(\widetilde{v}).P) \mid (m : \overline{a}\langle\widetilde{e}\rangle) \Rightarrow (n :!a(\widetilde{v}).P) \mid (\max(n,m) : P[\widetilde{v} := \widetilde{e}])$. Consider the typing $\varphi;\Phi;\Gamma \vdash_s (n :!a(\widetilde{v}).P) \mid (m : \overline{a}\langle\widetilde{e}\rangle) \triangleleft K$. The first rule is the rule for parallel composition, then the proof is split into the two following subtrees:

$$\dfrac{\dfrac{\varphi;\Phi;\Delta_0,\, \Delta_0' \vdash a : \forall\widetilde{i}.\mathtt{iserv}^{K_0'}_{I_0}(\widetilde{T_0}) \qquad \dfrac{\pi_P}{(\varphi,\, \widetilde{i});\Phi;\Theta_0,\, \widetilde{v}:\widetilde{T_0} \vdash_s P \triangleleft K_0'} \qquad \varphi;\Phi \vdash \langle\Delta_0\rangle_{-I_0} \sqsubseteq \Theta_0, \text{ time invariant}}{\varphi;\Phi;\Delta_0,\, \Delta_0' \vdash_s !a(\widetilde{v}).P \triangleleft I_0 \qquad \varphi;\Phi \vdash \langle\Gamma_0\rangle_{-n} \sqsubseteq \Delta_0,\, \Delta_0';I_0 \leq K_0}}{\dfrac{\varphi;\Phi;\langle\Gamma_0\rangle_{-n} \vdash_s !a(\widetilde{v}).P \triangleleft K_0}{\dfrac{\varphi;\Phi;\Gamma_0 \vdash_s n :!a(\widetilde{v}).P \triangleleft K_0+n \qquad \varphi;\Phi \vdash \Gamma \sqsubseteq \Gamma_0;K_0+n \leq K}{\varphi;\Phi;\Gamma \vdash_s n :!a(\widetilde{v}).P \triangleleft K,}}}$$

$$\dfrac{\dfrac{\dfrac{\varphi;\Phi;\Delta_1 \vdash a : \forall\widetilde{i}.\mathtt{oserv}^{K_1'}_{I_1}(\widetilde{T_1}) \qquad \dfrac{\pi_e}{\varphi;\Phi;\langle\Delta_1\rangle_{-I_1} \vdash \widetilde{e} : \widetilde{T_1}\{\widetilde{J}/\widetilde{i}\}}}{\varphi;\Phi;\Delta_1 \vdash_s \overline{a}\langle\widetilde{e}\rangle \triangleleft I_1+K_1'\{\widetilde{J}/\widetilde{i}\}} \qquad \varphi;\Phi \vdash \langle\Gamma_1\rangle_{-m} \sqsubseteq \Delta_1;I_1+K_1'\{\widetilde{J}/\widetilde{i}\} \leq K_1}{\dfrac{\varphi;\Phi;\langle\Gamma_1\rangle_{-m} \vdash_s \overline{a}\langle\widetilde{e}\rangle \triangleleft K_1}{\dfrac{\varphi;\Phi;\Gamma_1 \vdash_s m : \overline{a}\langle\widetilde{e}\rangle \triangleleft K_1+m \qquad \varphi;\Phi \vdash \Gamma \sqsubseteq \Gamma_1;K_1+m \leq K}{\varphi;\Phi;\Gamma \vdash_s m : \overline{a}\langle\widetilde{e}\rangle \triangleleft K.}}}$$

The first subtree can be used exactly as it is to type the server in the right part of the reduction relation. Furthermore, as the name a is used as an input and as an output, the original type in Γ for this name must be a server type $\forall \widetilde{i}.\mathsf{serv}_I^{K_a}(\widetilde{T})$. As this server does not lose its input capacity after the time decrease, we also know that $\varphi;\Phi \vDash I \geq n$. So, by subtyping, we have

$$\varphi;\Phi \vDash I_0 = I\text{–}n, \qquad (\varphi,\widetilde{i});\Phi \vdash \widetilde{T} \sqsubseteq \widetilde{T_0}, \qquad (\varphi,\widetilde{i});\Phi \vDash K_0' \leq K_a,$$
$$\varphi;\Phi \vDash I_1 = I\text{–}m, \qquad (\varphi,\widetilde{i});\Phi \vdash \widetilde{T_1} \sqsubseteq \widetilde{T}, \qquad (\varphi,\widetilde{i});\Phi \vDash K_a \leq K_1'.$$

There are now two cases to consider: Either $\varphi;\Phi \vDash I \geq m$ or $\varphi;\Phi \nvDash I \geq m$. The most difficult case between the two is $\varphi;\Phi \nvDash I \geq m$, hence we only detail this case.

Suppose that $\varphi;\Phi \nvDash I \geq m$. Note that as we know $\varphi;\Phi \vDash I \geq n$, it means that $m > n$. Moreover, as $\varphi;\Phi \vDash I_1 = I - m$, then $\varphi;\Phi \vDash I_1{+}m = \max(I,m)$. We still have

$$(\varphi,\widetilde{i});\Phi \vdash \widetilde{T_1} \sqsubseteq \widetilde{T_0}, \qquad (\varphi,\widetilde{i});\Phi \vDash K_0' \leq K_1'.$$

Thus, by subtyping, from π_P, we can obtain a proof of $(\varphi,\widetilde{i});\Phi;\Theta_0,\widetilde{v} : \widetilde{T_1} \vdash_{\mathrm{s}} P \triangleleft K_1'$. By Lemma 5.13, we have a proof of $\varphi;\Phi\{\widetilde{J}/\widetilde{i}\};\Theta_0\{\widetilde{J}/\widetilde{i}\},\widetilde{v} : \widetilde{T_1}\{\widetilde{J}/\widetilde{i}\} \vdash_{\mathrm{s}} P \triangleleft K_1'\{\widetilde{J}/\widetilde{i}\}$. As $\widetilde{i}$ only appears in $\widetilde{T_1}$ and K_1', we obtain a proof of $\varphi;\Phi;\Theta_0,\widetilde{v} : \widetilde{T_1}\{\widetilde{J}/\widetilde{i}\} \vdash_{\mathrm{s}} P \triangleleft K_1'\{\widetilde{J}/\widetilde{i}\}$.

Now, by using several times Lemma 5.10, we have

$$\varphi;\Phi \vdash \langle\Gamma\rangle_{-(n+I_0)} \sqsubseteq \epsilon_0, \langle\Delta_0\rangle_{-I_0} \sqsubseteq \epsilon_0, \Theta_0, \qquad \varphi;\Phi \vdash \langle\Gamma\rangle_{-(m+I_1)} \sqsubseteq \epsilon_1, \langle\Delta_1\rangle_{-I_1},$$

for some ϵ_0, ϵ_1. Moreover, we know that Θ_0 is time invariant. let us call $J = max(I_0{+}n, I_1{+}m)$, we have by again decreasing in the first subtyping relation:

$$\varphi;\Phi \vdash \langle\Gamma\rangle_{-J} \sqsubseteq \epsilon_2, \Theta_0,$$

for some ϵ_2. Note that we have $\varphi;\Phi \vDash J = max(I,m) = I_1{+}m$, since $\varphi;\Phi \vDash I_0{+}n = I$ and $\varphi;\Phi \vDash I_1{+}m = max(I,m)$.

By Lemma 5.11, we can obtain two proofs, with the subtyping rule:

$$\varphi;\Phi;\langle\Gamma\rangle_{-J},\widetilde{v} : \widetilde{T_1}\{\widetilde{J}/\widetilde{i}\} \vdash_{\mathrm{s}} P \triangleleft K_1\{\widetilde{J}/\widetilde{i}\}, \qquad \varphi;\Phi;\langle\Gamma\rangle_{-J} \vdash \widetilde{e} : \widetilde{T_1}\{\widetilde{J}/\widetilde{i}\}.$$

Thus, by the substitution lemma (Lemma 5.16), we have $\varphi;\Phi;\langle\Gamma\rangle_{-J} \vdash_{\mathrm{s}} P[\widetilde{v} := \widetilde{e}] \triangleleft K_1\{\widetilde{J}/\widetilde{i}\}$. Recall that in this case, $\max(n,m) = m$. We can obtain the following typing using the associated typing rule:

$$\varphi;\Phi;\langle\Gamma\rangle_{-(J-m)} \vdash_{\mathrm{s}} m : P[\widetilde{v} := \widetilde{e}] \triangleleft m{+}K_1\{\widetilde{J}/\widetilde{i}\}.$$

Then, by delaying (Lemma 5.15), we have $\varphi;\Phi;\langle\langle\Gamma\rangle_{-(J-m)}\rangle_{+(J-m)} \vdash_{\mathrm{s}} m : P[\widetilde{v} := \widetilde{e}] \triangleleft J{+}K_1\{\widetilde{J}/\widetilde{i}\}$, and $\Gamma = \epsilon_0', \epsilon_1'$ with $\varphi;\Phi \vdash \epsilon_1' \sqsubseteq \langle\langle\Gamma\rangle_{-(J-m)}\rangle_{+(J-m)}$. Recall that $\varphi;\Phi \vDash J = I_1{+}m$ and $\varphi;\Phi \vDash I_1{+}m{+}K_1\{\widetilde{J}/\widetilde{i}\} \leq K$. Thus, again by subtyping and weakening, we obtain

$$\varphi;\Phi;\Gamma \vdash_{\mathrm{s}} \max(n,m) : P[\widetilde{v} := \widetilde{e}] \triangleleft K.$$

This concludes this case. Note that many notations in this case are somewhat complicated, because we only know that $\varphi;\Phi \vDash I \geq m$ is false, but it does not immediately mean that $\varphi;\Phi \vDash m > I$, because the relation on indexes is not complete. So, we have to take that into account when writing substraction.

The case of simple channel synchronization works in the same way.

- **Case** $\mathtt{match}\ e :: e'\ \{[] \mapsto P;;\ x :: y \mapsto Q\} \Rightarrow Q[x,y := e,e']$. This case is more difficult than its counterpart for integers, thus we only detail this case and the one for integers can easily be deduced from this one. Moreover, this case is also more difficult than the case for $\ni$. Assume given a derivation $\varphi;\Phi;\Gamma \vdash_{\mathrm{s}} \mathtt{match}\ e :: e'\ \{[] \mapsto P;;\ x :: y \mapsto Q\} \triangleleft K$. Then the proof has the shape

$$\dfrac{\dfrac{\dfrac{\pi_e}{\varphi;\Phi;\Delta \vdash e : \mathcal{B}'} \quad \dfrac{\pi_{e'}}{\varphi;\Phi;\Delta \vdash e' : \mathsf{List}[I', J'](\mathcal{B}')}}{\varphi;\Phi;\Delta \vdash e :: e' : \mathsf{List}[I'{+}1, J'{+}1](\mathcal{B}')} \quad \varphi;\Phi \vdash \Gamma \sqsubseteq \Delta; \mathsf{List}[I'{+}1, J'{+}1](\mathcal{B}') \sqsubseteq \mathsf{List}[I, J](\mathcal{B})}{\varphi;\Phi;\Gamma \vdash e :: e' : \mathsf{List}[I, J](\mathcal{B})} \quad \pi_Q}{\varphi;\Phi;\Gamma \models_{\mathrm{s}} \mathtt{match}\ e :: e'\ \{[] \mapsto P;;\ x :: y \mapsto Q\} \triangleleft K.}$$

Where we ignore the branch for P and π_Q proves $\varphi;(\Phi, J \geq 1);\Gamma, x:\mathcal{B}, y:\mathsf{List}[I{-}1, J{-}1](\mathcal{B}) \models_{\mathrm{s}} Q \triangleleft K$. Subtyping gives us the following information:

$$\varphi;\Phi \vDash I \leq I'{+}1, \qquad \varphi;\Phi \vDash J'{+}1 \leq J, \qquad \varphi;\Phi \vdash \mathcal{B}' \sqsubseteq \mathcal{B}.$$

From this, we can deduce the following constraints:

$$\varphi;\Phi \vDash J \geq 1, \qquad \varphi;\Phi \vDash I{-}1 \leq I', \qquad \varphi;\Phi \vDash J' \leq J - 1.$$

Thus, with the subtyping rule and the proofs π_e and $\pi_{e'}$, we obtain

$$\varphi;\Phi;\Gamma \vdash e : \mathcal{B}, \qquad \varphi;\Phi;\Gamma \vdash e' : \mathsf{List}[I{-}1, J{-}1](\mathcal{B}).$$

Then, by Lemma 5.12, from π_Q, we obtain a proof of $\varphi;\Phi;\Gamma, x:\mathcal{B}, y:\mathsf{List}[I{-}1, J{-}1](\mathcal{B}) \models_{\mathrm{s}} Q \triangleleft K$. By the substitution lemma (Lemma 5.16), we obtain $\varphi;\Phi;\Gamma \models_{\mathrm{s}} Q[x, y := e, e'] \triangleleft K$. This concludes this case.

- **Case** $P \Rightarrow Q$ with $P \equiv P'$, $P' \Rightarrow Q'$, and $Q \equiv Q'$. Suppose that $\varphi;\Phi;\Gamma \models_{\mathrm{s}} P \triangleleft K$. By Lemma 5.17, we have $\varphi;\Phi;\Gamma \models_{\mathrm{s}} P' \triangleleft K$. By induction hypothesis, we obtain $\varphi;\Phi;\Gamma \models_{\mathrm{s}} Q' \triangleleft K$. Then, again by Lemma 5.17, we have $\varphi;\Phi;\Gamma \models_{\mathrm{s}} Q \triangleleft K$. This concludes this case.

This concludes the proof of Theorem 5.18.

Now that we have the subject reduction for $\Rightarrow$, we can easily deduce a more generic form of Theorem 5.8.

Theorem 5.19. *Let P be an annotated process such that $\varphi;\Phi;\Gamma \models_{\mathrm{s}} P \triangleleft K$. Then, $\varphi;\Phi \vDash K \geq \mathsf{span}(P)$.*

Proof. By Theorem 5.18, all reductions from P using $\Rightarrow$ preserve the typing. Moreover, for any process Q, if we have a typing $\varphi;\Phi;\Gamma \models_{\mathrm{s}} Q \triangleleft K$, then $\varphi;\Phi \vDash K \geq C_\ell(Q)$. Indeed, a constructor $n : P$ incurs an increment of the complexity of n both in typing and in the definition of $C_\ell(Q)$, and for parallel composition the typing imposes a complexity greater than the maximum as in the definition for $C_\ell(Q)$. Thus, for any process Q reachable from P, we have $\varphi;\Phi \vDash K \geq C_\ell(Q)$, and K is indeed a bound on the span. □

Corollary 5.9, which has been stated before, is then obtained with the substitution lemma and the rule for parallel composition.

6 AN EXAMPLE: BITONIC SORT

As an example for this type system, we show how to obtain the bound on a common parallel algorithm: bitonic sort [1]. The particularity of this sorting algorithm is that it admits a parallel complexity in $O(log(n)^2)$. We will show here that our type system allows one to derive this bound for the algorithm, just as the paper-and-pen analysis. Actually, we consider here a version for lists, which is not optimal for the number of operations, but we obtain the usual number of comparisons. For the sake of simplicity, we consider the case of lists of size a power of 2. Let us briefly sketch the ideas of this algorithm. For a formal description, see Reference [1].

- A *bitonic sequence* is either a sequence composed of an increasing sequence followed by a decreasing sequence (e.g., References [2, 5, 7, 8, 19, 23]), or a cyclic rotation of such a sequence (e.g., References [2, 5, 7, 8, 19, 23]).

- The algorithm uses two main functions, `bmerge` and `bsort`.
- `bmerge` takes a bitonic sequence and recursively sorts it as follows:
  Assume $s = [a_0, \ldots, a_{n-1}]$ is a bitonic sequence such that $[a_0, \ldots, a_{n/2-1}]$ is increasing and $[a_{n/2} \ldots, a_{n-1}]$ is decreasing, then we consider
  $s_1 = [\min(a_0, a_{n/2}), \min(a_1, a_{n/2+1}) \ldots, \min(a_{n/2-1}, a_{n-1})]$ the list of minima and
  $s_2 = [\max(a_0, a_{n/2}), \max(a_1, a_{n/2+1}) \ldots, \max(a_{n/2-1}, a_{n-1})]$ the list of maxima.
  Then, we have s_1 and s_2 are bitonic and satisfy $\forall x \in s_1, \forall y \in s_2, x \leq y$.
  `bmerge` then applies recursively to s_1 and s_2 to produce a sorted sequence.
- `bsort` takes a sequence and recursively sorts it. It starts by separating the sequence in two. Then, it recursively sorts the first sequence in increasing order, and the second sequence in decreasing order. With this, we obtain a bitonic sequence that can be sorted with `bmerge`.

We will encode this algorithm in π-calculus with a Boolean type. As said before, our results can easily be extended to support Boolean with a conditional constructor.

First, we suppose that a server for comparison `lessthan` is already implemented. We start with `bcompare` such that given two lists of same length, it creates the list of minimum and the list of maximum (s_2 and s_1 described above). The process is given in Figure 16 and intuitively, it compares the two top elements of the lists, puts the minimum in the left list and the maximum in the right list, and then proceeds recursively.

Then, to present the processes for bitonic sort, let us use the macro $\texttt{let}\ \widetilde{v} = f(\widetilde{e})\ \texttt{in}\ P$ to represent $(va)(\overline{f}\langle \widetilde{e}, a\rangle \mid a(\widetilde{v}).P)$, with the meaning that if $\overline{f}$ is a server representing a function f, then this process indeed corresponds to computing f on $\widetilde{e}$, and putting the output in the variables $\widetilde{v}$ in the continuation P. We will only use it with f that has complexity 0, then the associated typing rule corresponds to adding $\widetilde{v}$ in the context with types corresponding to the outputs of the function f. We also use a generalized pattern matching, allowing us to match on a list with one element, which could formally be done by two successive pattern matching. We also assume that we have a function for concatenation of lists and a function `partition` taking a list of size $2n$, and giving two lists corresponding to the first n elements and the last n elements. Then, the process for bitonic sort is given in Figure 16. The server bmerge takes a list, splits it into two lists l_1 and l_2, computes the list of minima p_1 and the list of maxima p_2, applies recursively to those lists, and then concatenates them in increasing order if *up* is true, and in decreasing order otherwise. And finally, bsort takes a list, separates it into two lists, sorts the first one in increasing order and the second one in decreasing order, and then merges them using the previous function.

We present here intuitively the typing. To begin with, we suppose that `lessthan` is given the server type $\texttt{oserv}_0^0(\mathcal{B}, \mathcal{B}, \mathsf{ch}_0(\mathsf{Bool}))$, saying that this is a server ready to be called, and it takes in input a channel that is used to return the Boolean value. With this, we can give to `bcompare` the following server type:

$$\forall i.\mathsf{serv}_0^1(\mathsf{List}[0,i](\mathcal{B}), \mathsf{List}[0,i](\mathcal{B}), \mathsf{out}_1(\mathsf{List}[0,i](\mathcal{B}), \mathsf{List}[0,i](\mathcal{B}))).$$

The important things to notice are that this server has complexity 1, and the channel taken in input has a time 1. A sketch of this typing is given in Figure 17(a), where we use abbreviations for the name of server (*lt* for *lessthan*, *bc* for *bcompare*...). The cases of empty lists are not detailed, but they are easy. In the non-empty case, for the ν constructor, we must give a type to the channels b and c. We use

$$b : \mathsf{ch}_1(\mathsf{List}[0,i-1](\mathcal{B}), \mathsf{List}[0,i-1](\mathcal{B})), \qquad c : \mathsf{ch}_1(\mathsf{Bool}).$$

And we can then type the various processes in parallel.

- For the call to `bcompare`, the arguments have the expected type, and this call has complexity 1, because of the type of `bcompare`.

```
!bcompare(l1,l2,a). match(l1) {
  [] ↦ ā⟨l1,l2⟩ ;;
  x::l1' ↦ match(l2) {
    [] ↦ ā⟨l1,l2⟩ ;;
    y::l2' ↦ (νb)(νc)(
    bcompare⟨l1',l2',b⟩ | tick.lessthan⟨x,y,c⟩
    | b(lm,lM).c(z).if z then ā⟨x::lm,y::lM⟩ else ā⟨y::lm,x::lM⟩
    )
  }
}
!bmerge(up,l,a). match(l) {
  [] ↦ ā⟨l⟩ ;;
  [y] ↦ ā⟨l⟩ ;;
  _ ↦ let (l1,l2) = partition(l) in (νb)(νc)(νd)(
  bcompare⟨l1,l2,b⟩ | b(p1,p2).(bmerge⟨up,p1,c⟩ | bmerge⟨up,p2,d⟩)
  | c(q1).d(q2). if up then let l' = q1 @ q2 in ā⟨l'⟩ else let l' = q2 @ q1 in ā⟨l'⟩
  )
}

!bsort(up,l,a). match(l) {
  [] ↦ ā⟨l⟩ ;;
  [y]↦ ā⟨l⟩ ;;
  _ ↦ let (l1,l2) = partition(l) in (νb)(νc)(νd)(
  bsort⟨tt,l1,b⟩ | bsort⟨ff,l2,c⟩
  | b(q1).c(q2).let q = q1 @ q2 in bmerge⟨up,q,d⟩ | d(p).ā⟨p⟩
  )
}
```

Fig. 16. Bitonic sort.

- For the process $\texttt{tick}.\overline{\texttt{lessthan}}\langle x,y,c\rangle$, the tick enforces a decreasing of time 1 in the context. This modifies in particular the time of c, that becomes 0. Thus, we can do the call to `lessthan` as everything is well-typed.
- Finally, for the last process, we have in the two branches a shape $b(\cdots).c(\cdots).\overline{a}\langle\cdots\rangle$. So, by Section 5.3.1, as all those three channels b, c and a have a time equal to 1, we have a complexity 1 for this typing.

So, we can indeed give this server type to `bcompare`, and thus we can call this server and it generates a complexity of 1.

Then, the main point in the typing of the two remaining servers is to find a solution to a recurrence relation for the complexity of server types. In the typing of `bmerge`, we suppose given a list of size smaller than 2^i, and we choose both the complexity of this type and the time of the channel a equal to an index $f(i)$ as in Section 5.3.3. So, it means we choose for `bmerge` the type:

$$\forall i.\mathsf{serv}_0^{f(i)}(\mathsf{Bool},\mathsf{List}[0,2^i](\mathcal{B}),\mathsf{out}_{f(i)}(\mathsf{List}[0,2^i](\mathcal{B}))).$$

Then, the typing given in Figure 17(b) gives us the following conditions, for the three branches, when $2^i \geq 2$ (or equivalently, $i \geq 1$):

$$f(i) \geq 1, \qquad f(i) \geq 1{+}f(i{-}1), \qquad f(i) \geq f(i{-}1) \geq f(i{-}1).$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{i;i\geq 1;\Delta\vdash(l_1',l_2',b):\widetilde{T}(i{-}1)}(\text{Ax})}{i;i\geq 1;\Delta\models_s\overline{bc}\langle l_1',l_2',b\rangle\triangleleft 1}(\text{Oserv})}{\cdots}(\text{Par}) \quad \dfrac{\dfrac{\dfrac{\dfrac{}{i;i\geq 1;\langle\Delta\rangle_{-1}\vdash(x,y,c):(\mathcal{B},\mathcal{B},\mathsf{ch}_0(\mathsf{Bool}))}(\text{Ax})}{i;i\geq 1;\langle\Delta\rangle_{-1}\models_s\overline{lt}\langle x,y,c\rangle\triangleleft 0}(\text{Oserv})}{i;i\geq 1;\Delta\models_s\mathtt{tick}.\overline{lt}\langle x,y,c\rangle\triangleleft 1}(\text{Tick}) \quad \cdots \text{ See Section 5.3.1 } \cdots}{i;i\geq 1;\Delta\models_s\overline{bc}\langle l_1',l_2',b\rangle\mid\mathtt{tick}.\overline{lt}\langle x,y,c\rangle\mid c(z).\mathtt{if}\ \cdots\triangleleft 1}(\text{Par})}{i;i\geq 1;\Delta\models_s\overline{bc}\langle l_1',l_2',b\rangle\mid\mathtt{tick}.\overline{lt}\langle x,y,c\rangle\mid c(z).\mathtt{if}\ \cdots\triangleleft 1}(\text{Nu})}{i;i\geq 1;\Gamma',(l_1,l_2,a):\widetilde{T}(i),x:\mathcal{B},y:\mathcal{B},l_1':\mathsf{List}[0,i{-}1](\mathcal{B}),l_2':\mathsf{List}[0,i{-}1](\mathcal{B})\models_s(vb)(vc)\cdots\triangleleft 1}(\text{Listpm})}{i;i\geq 1;\Gamma',(l_1,l_2,a):\widetilde{T}(i),x:\mathcal{B},l_1':\mathsf{List}[0,i{-}1](\mathcal{B})\models_s\mathtt{match}\ l_2\ \{[]\mapsto\cdots;;\ y::l_2'\mapsto\cdots\}\triangleleft 1}(\text{Listpm})}{i;\cdot;\Gamma',(l_1,l_2,a):\widetilde{T}(i)\models_s\mathtt{match}\ l_1\ \{[]\mapsto\cdots;;\ x::l_1'\mapsto\cdots\}\triangleleft 1 \quad \Gamma'\text{ time invariant} \quad \cdot;\cdot;\vdash\langle\Gamma\rangle_{-0}\sqsubseteq\Gamma'}(\text{Iserv})}{\cdot;\cdot;\Gamma\models_s!bc(l_1,l_2,a)\ldots.\triangleleft 0}$$

$$\widetilde{T}(i)\coloneqq\mathsf{List}[0,i](\mathcal{B}),\mathsf{List}[0,i](\mathcal{B}),\mathsf{out}_1(\mathsf{List}[0,i](\mathcal{B}),\mathsf{List}[0,i](\mathcal{B}))$$

$$\Gamma\coloneqq lt:\mathsf{oserv}_0^0(\mathcal{B},\mathcal{B},\mathsf{ch}_0(\mathsf{Bool})),bc:\forall i.\mathsf{serv}_0^1(\widetilde{T}(i)) \qquad \Gamma'\coloneqq lt:\mathsf{oserv}_0^0(\mathcal{B},\mathcal{B},\mathsf{ch}_0(\mathsf{Bool})),bc:\forall i.\mathsf{oserv}_0^1(\widetilde{T}(i))$$

$$\Delta\coloneqq\Gamma',(l_1,l_2,a):\widetilde{T}(i),x:\mathcal{B},y:\mathcal{B},l_1':\mathsf{List}[0,i{-}1](\mathcal{B}),l_2':\mathsf{List}[0,i{-}1](\mathcal{B}),b:\mathsf{out}_1(\mathsf{List}[0,i{-}1](\mathcal{B}),\mathsf{List}[0,i{-}1](\mathcal{B})),c:\mathsf{ch}_1(\mathsf{Bool})$$

(a) Type Derivation for Bitonic Comparison.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{i;\Phi;\Delta\vdash(l_1,l_2,b):\widetilde{T}(2^{i-1})}(\text{Ax})}{i;\Phi;\Delta\models_s\overline{bc}\langle l_1,l_2,b\rangle\triangleleft 1}(\text{Oserv})}{i;\Phi;\Delta\models_s\overline{bc}\langle l_1,l_2,b\rangle\triangleleft f(i)}(\text{Sub}) \quad \dfrac{\cdots \text{ See Section 5.3.1 } \cdots}{i;\Phi;\Delta\models_s b(p_1,p_2).\cdots\triangleleft 1{+}f(i{-}1)}(\text{Sub}) \quad \cdots \text{ see Section 5.3.1 } \cdots}{i;\Phi;\Delta\models_s\overline{bc}\langle l_1,l_2,b\rangle\mid b(p_1,p_2).\cdots\mid c(q_1).\cdots\triangleleft f(i)}(\text{Par})}{i;\Phi;\Gamma',(up,l,a):\widetilde{U}(i),l_1:\mathsf{List}[0,2^{i-1}](\mathcal{B}),l_2:\mathsf{List}[0,2^{i-1}](\mathcal{B})\models_s(vb)(vc)(vd)\cdots\triangleleft f(i)}(\text{Nu})}{i;2^i\geq 2;\Gamma',(up,l,a):\widetilde{U}(i)\models_s\mathtt{let}\ (l_1,l_2)=\mathtt{partition}(l)\ \mathtt{in}\ \cdots\triangleleft f(i)}(\text{Let})}{i;\cdot;\Gamma',(up,l,a):\widetilde{U}(i)\models_s\mathtt{match}\ l\ \{\cdots\}\triangleleft f(i) \quad \Gamma'\text{ time invariant} \quad \cdot;\cdot;\vdash\langle\Gamma\rangle_{-0}\sqsubseteq\Gamma'}(\text{Iserv})}{\cdot;\cdot;\Gamma\models_s!bm(up,l,a)\ldots.\triangleleft 0}(\text{Listpm})$$

$$\widetilde{T}(i)\coloneqq\mathsf{List}[0,i](\mathcal{B}),\mathsf{List}[0,i](\mathcal{B}),\mathsf{out}_1(\mathsf{List}[0,i](\mathcal{B}),\mathsf{List}[0,i](\mathcal{B})) \qquad \widetilde{U}(i)\coloneqq\mathsf{Bool},\mathsf{List}[0,2^i](\mathcal{B}),\mathsf{out}_{f(i)}(\mathsf{List}[0,2^i](\mathcal{B}))$$

$$\Gamma\coloneqq bc:\forall i.\mathsf{oserv}_0^1(\widetilde{T}(i)),bm:\forall i.\mathsf{serv}_0^{f(i)}(\widetilde{U}(i)) \qquad \Gamma'\coloneqq bc:\forall i.\mathsf{oserv}_0^1(\widetilde{T}(i)),bm:\forall i.\mathsf{oserv}_0^{f(i)}(\widetilde{U}(i)) \qquad \Phi\coloneqq 2^i\geq 2$$

$$\Delta\coloneqq\Gamma',(up,l,a):\widetilde{U}(i),l_1:\mathsf{List}[0,2^{i-1}](\mathcal{B}),l_2:\mathsf{List}[0,2^{i-1}](\mathcal{B}),b:\mathsf{out}_1(\mathsf{List}[0,2^{i-1}](\mathcal{B}),\mathsf{List}[0,2^{i-1}](\mathcal{B})),c,d:\mathsf{out}_{f(i-1)}(\mathsf{List}[0,2^{i-1}](\mathcal{B}))$$

(b) Type Derivation for Bitonic Merge

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{i;\Phi;\Delta\vdash(\mathtt{tt},l_1,b):\widetilde{V}(i{-}1)}(\text{Ax})}{i;\Phi;\Delta\models_s\overline{bs}\langle\mathtt{tt},l_1,b\rangle\triangleleft g(i{-}1)}(\text{Oserv})}{i;\Phi;\Delta\models_s\overline{bs}\langle\mathtt{tt},l_1,b\rangle\triangleleft g(i)}(\text{Sub}) \quad \dfrac{\cdots \text{ See Section 5.3.1 } \cdots}{i;\Phi;\Delta\models_s b(q_1).\cdots\triangleleft g(i{-}1){+}f(i)}(\text{Sub}) \quad \cdots \text{ See Section 5.3.1 } \cdots}{i;\Phi;\Delta\models_s\overline{bs}\langle\mathtt{tt},l_1,b\rangle\mid\overline{bs}\langle\mathtt{ff},l_2,c\rangle\mid b(q_1).\cdots\mid d(p).\overline{a}\langle p\rangle\triangleleft g(i)}(\text{Par})}{i;\Phi;\Gamma',(up,l,a):\widetilde{V}(i),l_1:\mathsf{List}[0,2^{i-1}](\mathcal{B}),l_2:\mathsf{List}[0,2^{i-1}](\mathcal{B})\models_s(vb)(vc)(vd)\cdots\triangleleft g(i)}(\text{Nu})}{i;2^i\geq 2;\Gamma',(up,l,a):\widetilde{V}(i)\models_s\mathtt{let}\ (l_1,l_2)=\mathtt{partition}(l)\ \mathtt{in}\ \cdots\triangleleft g(i)}(\text{Let})}{i;\cdot;\Gamma',(up,l,a):\widetilde{V}(i)\models_s\mathtt{match}\ l\ \{\cdots\}\triangleleft g(i) \quad \Gamma'\text{ time invariant} \quad \cdot;\cdot;\vdash\langle\Gamma\rangle_{-0}\sqsubseteq\Gamma'}(\text{Iserv})}{\cdot;\cdot;\Gamma\models_s!bs(up,l,a)\ldots.\triangleleft 0}(\text{Listpm})$$

$$\widetilde{U}(i)\coloneqq\mathsf{Bool},\mathsf{List}[0,2^i](\mathcal{B}),\mathsf{out}_{f(i)}(\mathsf{List}[0,2^i](\mathcal{B})) \qquad \widetilde{V}(i)\coloneqq\mathsf{Bool},\mathsf{List}[0,2^i](\mathcal{B}),\mathsf{out}_{g(i)}(\mathsf{List}[0,2^i](\mathcal{B}))$$

$$\Gamma\coloneqq bm:\forall i.\mathsf{oserv}_0^{f(i)}(\widetilde{U}(i)),bs:\forall i.\mathsf{serv}_0^{g(i)}(\widetilde{V}(i)) \qquad \Gamma'\coloneqq bm:\forall i.\mathsf{oserv}_0^{f(i)}(\widetilde{U}(i)),bs:\forall i.\mathsf{oserv}_0^{g(i)}(\widetilde{V}(i)) \qquad \Phi\coloneqq 2^i\geq 2$$

$$\Delta\coloneqq\Gamma',(up,l,a):\widetilde{V}(i),l_1:\mathsf{List}[0,2^{i-1}](\mathcal{B}),l_2:\mathsf{List}[0,2^{i-1}](\mathcal{B}),b,c:\mathsf{out}_{g(i-1)}(\mathsf{List}[0,2^{i-1}])(\mathcal{B}),d:\mathsf{out}_{(g(i-1)+f(i))}(\mathsf{List}[0,2^i])(\mathcal{B})$$

(c) Type Derivation for Bitonic Sort

Fig. 17. Bitonic sort.

The first inequality comes from the (Sub) rule for the first branch, the second inequality comes from the (Sub) rule for the second branch, and the third inequality comes from the condition that the time of a should be greater than the time of d, which should be greater than the time of c

in the third branch, as explained in Section 5.3.1. So, we can take $f(i) = i$, and thus bmerge has logarithmic complexity.

In the same way, for bsort, we chose the type

$$\forall i.\mathsf{serv}_0^{g(i)}(\mathsf{Bool}, \mathsf{List}[0, 2^i](\mathcal{B}), \mathsf{out}_{g(i)}(\mathsf{List}[0, 2^i](\mathcal{B}))).$$

The typing is given in Figure 17(c), which gives us the following conditions, again for the three parallel branches, when $2^i \geq 2$ (equivalently, $i \geq 1$):

$$g(i) \geq g(i{-}1), \qquad g(i) \geq g(i{-}1){+}f(i), \qquad g(i) \geq g(i{-}1){+}f(i).$$

The first inequality comes from the (Sub) rule for the first branch, the second inequality comes from the (Sub) rule for the second branch, where $g(i{-}1){+}f(i)$ is the complexity of $b(q_1).\cdots$ (time of b and c summed with the call to bmerge), and the last inequality comes from the third branch, as the time of a should be greater than the time of d. So, if we take $f(i) = i$ as previously, then we have

$$i \geq 1 \text{ implies } g(i) \geq g(i{-}1){+}i.$$

Thus, we can take $g(i)$ in $O(i^2)$, and we obtain in the end that bitonic sort is indeed in $O(log(n)^2)$ on a list of size n.

Note that in this example, the type system gives recurrence relations corresponding to the usual recurrence relations we would obtain with a complexity analysis by hand. Here, the recurrence relation is only on f or g, because channel names are only used as return channels, hence their time is always equal to the complexity of the server that uses them. In general, this is not the case; as we saw before in Section 5.3.3, we can obtain mutually recurrent relations when defining a server.

7 SPAN AND CAUSAL COMPLEXITY

In this section, we present how our notion of span can be linked with the causal complexity of the literature, as we believe it can be of interest to show that both annotated processes and causal complexity are well-behaved notions of parallel complexity in the π-calculus.

7.1 Presentation of Causal Complexity

We present here a notion of causal complexity inspired by other works [14–16]. We explained before with the canonical form in Definition 3.1 that a process can be described by a set of names and a multiset of guarded processes, when working up to congruence. For causal complexity, we consider more structure for processes. The idea is to see a process as a set of names and a binary tree where leaves are guarded processes and a node means parallel composition. Formally, instead of using the previous congruence relation, we use the *tree congruence*. This is defined as the least congruence relation $\equiv_t$ closed under

$$(va)(vb)P \equiv_t (vb)(va)P,$$

$$(va)(P \mid Q) \equiv_t (va)P \mid Q \text{ (when } a \text{ is not free in } Q), \qquad (va)(P \mid Q) \equiv_t P \mid (va)Q \text{ (when } a \text{ is not free in } P).$$

This tree congruence can indeed move names as before, but it preserves the tree-shape of a process. With this intuition, we can redefine the semantics to preserve this tree structure. For this, we annotate the reduction relation by a *location*.

Definition 7.1 (Locations and Positions). The sets of *positions* and *locations* are given by the following grammars:

$$p := \epsilon \mid 0 \cdot p \mid 1 \cdot p, \qquad \ell := p; \mathtt{tick} \mid p; \mathsf{pm} \mid \tau\langle p, p' \rangle.$$

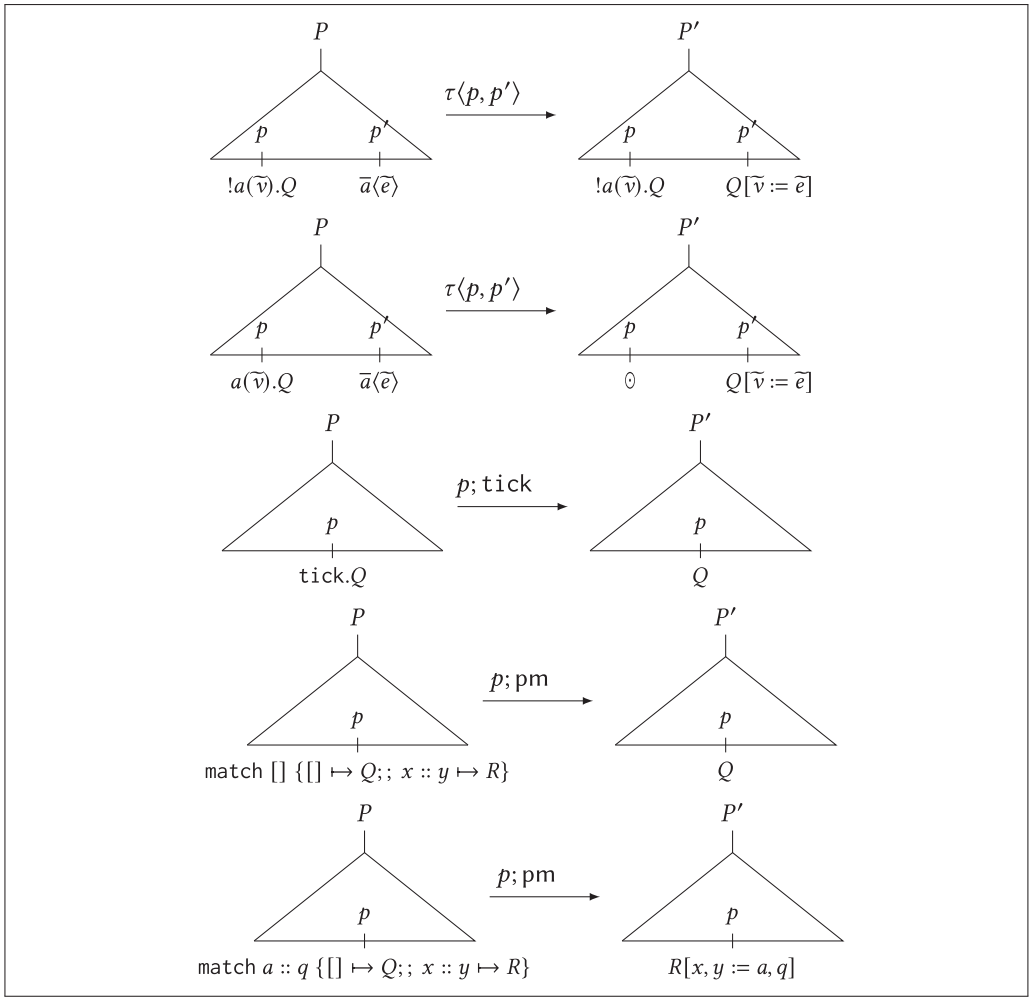


Fig. 18. Semantics for causal complexity.

The intuition is that a reduction $P \xrightarrow{p;\texttt{tick}} P'$ removed the top `tick` of the guarded process of P at position p in the tree. A reduction $P \xrightarrow{\tau\langle p,p'\rangle} P'$ is a communication between the input process at position p and the output process at position p'. Finally, the action $p;\mathsf{pm}$ is for the position of a pattern-matching reduction.

The rules are described in Figure 18. For example, from a process P seen as a tree with at position p a replicated input $!a(\widetilde{v}).Q$ and at position p' an output $\overline{a}\langle\widetilde{e}\rangle$, we obtain the process P', the same tree as P, with at position p the same replicated input but at position p', we have the tree corresponding to the process $Q[\widetilde{v} := \widetilde{e}]$. The location $\tau\langle p, p'\rangle$ indicates at which positions the modification happened, and what was the action performed. In the same way, we can define a reduction annotated by a `tick`, or a match. An alternative presentation of this semantics, closer to the one in Reference [16], with a labelled transition system, is also possible. As for the standard reduction in π-calculus, both semantics (the one with the congruence and the one with the labelled transition system) are equivalent. We chose to present this one, because it is easier to use in proofs.

The goal of this semantics is first to preserve the tree structure in a reduction step, and second to remember when doing a reduction step where the modification occurs exactly in the tree. Then, we can define a causality relation between locations. The idea is that a location ℓ *causes* a location ℓ' (denoted $\ell \prec_c \ell'$) when the reduction step at location ℓ' could not have happened without the reduction step at location ℓ.

Definition 7.2. The causality relation $\ell \prec_c \ell'$ between locations is defined by:

- $p;\mathtt{tick} \prec_c p';\mathtt{tick}$ when p is a prefix of p',
- $p;\mathtt{tick} \prec_c \tau\langle p_0, p_1\rangle$ when p is a prefix of p_0 or p_1,
- $\tau\langle p_0, p_1\rangle \prec_c p;\mathtt{tick}$ when p_1 is a prefix of p,
- $\tau\langle p_0, p_1\rangle \prec_c \tau\langle p_0', p_1'\rangle$ when p_1 is a prefix of p_0' or p_1'.

As for pm locations, they behave as a `tick` location. By extension, we will sometime say that a location $\ell \prec_c p$ when $\ell \prec_c p;\mathtt{tick}$. Intuitively, this definition comes from where the continuation of a reduction is. For example, for the subprocess $\mathtt{tick}.Q$ at position p, the continuation Q after a reduction is a subtree with its root at position p. Thus, any position in this tree for Q has p as a prefix, and so any reduction happening in Q has p as a prefix in its location.

The main interest of causal complexity is that this notion of causality can be adapted to account for different behaviours. For example, in Reference [16], the causality relation is different. For instance, they consider that linear communications (on non replicated input) do not propagate causality in their work. Here, we chose this causality relation to show the equivalence with annotated processes, and in this sense, our notion of span is a particular case of causal complexity with this choice of causality relation.

The important point in this causality relation we define is that a τ location causes another location ℓ when the output position is a prefix of the positions in ℓ. Indeed, for a communication with a non-replicated input, the input position becomes a $\mathbb{0}$ thus it cannot cause anything, and for a communication with a replicated input, we consider that two calls to the same replicated input are independent one from the other. Then, the important point is that two reductions with independent locations could be done in any order, it would not change the final tree.

With this definition of causality, intuitively we can define causal complexity of a computation as the maximal number of tick in all the chains of causality in this computation.

Definition 7.3 (Computation). A *computation* from a process P is a sequence $(P_i, \ell_i)_{i\leq N}$ such that $P_0 = P$ and $P_i \xrightarrow{\ell_i} P_{i+1}$ for $i < N$.

Definition 7.4 (Chain of Causality). In a computation $(P_i, \ell_i)_{i\leq N}$, we say that ℓ_i *depends* on ℓ_j, noted $\ell_i \prec \ell_j$ when $i < j$ and $\ell_i \prec_c \ell_j$. Then a *chain of causality* in this computation is a chain $\ell_{i_1} < \ell_{i_2} < \cdots < \ell_{i_m}$.

Definition 7.5 (Causal Complexity). The causal complexity of a computation $(P_i, \ell_i)_{i\leq N}$ is given by the maximal number of tick locations in all the chains of causality. Finally, the causal complexity of a process P, denoted $\mathrm{CausC}(P)$ is defined as the maximal causal complexity over all computations from P.

We can now prove the equivalence between the notions of parallel with annotated processes complexity and causal complexity for this particular causality relation.

7.2 $\mathbf{Span}(P) \geq \mathbf{CausC}(P)$

In this section, we show that span is an upper bound on causal complexity. Formally, we prove the following lemma.

Lemma 7.6. *Let P be a process. Let $(P_i, \ell_i)_{i \leq N}$ be a computation from P with causal complexity K. Then,* $\mathrm{span}(P) \geq K$.

To do that, we show that we can do the same computation with our semantics with annotated processes. Let us take a process P seen as a tree. By definition, each leaf of P is a guarded process G. For each leaf, we replace the guarded process G by $0 : G$, and we call P' this annotated process. By the definition of congruence for annotated processes, we have $P \equiv P'$. Then, we will work with this tree representation for annotated processes.

Definition 7.7 (Tree Representation of Annotated Processes). We consider annotated trees such that a set of names is given at the beginning, nodes represent parallel composition and leaves are processes of the shape $n : G$ with G guarded. Such a tree indeed represents an annotated process.

Then, we say that an annotated process P' seen as a tree is an *annotation* of a process P seen as a tree if P' and P have exactly the same shape, and each leaf G of P is a leaf $n : G$ for P'.

So, by definition P' is an annotation of P. We can then prove the following lemma:

Lemma 7.8 (Causal Reduction and Annotation). *Suppose that $P \xrightarrow{\ell} Q$. Then, for any P' annotation of P, we have $P' \Rightarrow Q'$ where Q' is an annotation of Q.*

Moreover, we have:

- *If $\ell = p; \mathtt{tick}$, then Q' has the same annotation as P' for all leaves at a position p' such that p is not a prefix of p'. For all the other leaves, Q' has the annotation $n + 1$ where n is the annotation for the leaf at position p in P'.*
- *If $\ell = p; \mathrm{pm}$, then Q' has the same annotation as P' for all leaves at a position p' such that p is not a prefix of p'. For all the other leaves, Q' has the annotation n where n is the annotation for the leaf at position p in P'.*
- *If $\ell = \tau\langle p, p' \rangle$, then Q' has the same annotation as P' for all leaves at position q such that p' is not a prefix of q. For all the other leaves, Q' has the annotation $\max(n, m)$ where n is the annotation for the leaf at position p in P' and m is the annotation for the leaf at position p' in P'.*

Proof. The proof is done by case analysis on the rules of Figure 18. All cases are rather direct. The idea is to always keep the same tree shape in the reduction $\Rightarrow$, and then to make the names go up and the annotations go down after doing this reduction using the congruence rules. Formally, the names can go up with $((\nu a)P) \mid Q \equiv (\nu a)(P \mid Q)$ (always possible by α-renaming) and $n : ((\nu a)P) \equiv (\nu a)(n : P)$, and then the annotations can go down with $n : (P \mid Q) \equiv n : P \mid n : Q$. Then, with the shape of the reduction relation $\Rightarrow$, we can indeed see that the annotations indeed correspond to the one given in this lemma. □

With this lemma, we can start from the annotation P' of P, and simulate the computation. Now, we only need to show that the annotations correspond to the number of ticks in a chain of causality. Formally, we prove the following lemma.

Lemma 7.9. *Let $(P'_i, \ell_i)_{i \leq N}$ be the computation given by the previous lemma from an original computation $(P_i, \ell_i)_{i \leq N}$. Then, for all $i \leq N$, the annotation of a guarded process G at position p in P'_i is an upper bound of the maximal number of* `tick` *locations in all chains of causality for the locations $\ell_0, \ldots, \ell_{i-1}$ such that the last location ℓ of this chain satisfies $\ell \prec_c p$*

We prove this by induction on i.

- This is true for $i = 0$, since P' is P annotated with zeros everywhere.

- Let $i < N$. Suppose that this is true for P'_i, the annotation of P_i. Let us look at the reduction $P_i \xrightarrow{\ell_i} P_{i+1}$.
  - If $\ell_i = p;\mathtt{tick}$, then by induction hypothesis, the annotation n for the tick corresponds to the maximal number of `tick` locations in all chains of causality for the locations $\ell_0, \dots, \ell_{i-1}$ that are also in causality $\prec_c$ with p. Let us look at P'_{i+1} given by lemma 7.8. For all the positions in P'_{i+1} with p not a prefix, dependency did not change, since $\ell_i = p;\mathtt{pm}$ does not cause those positions. As annotations did not change either, the hypothesis is still correct. For the new positions in the tree with p as a prefix, all the annotations are n. All chains of causality with the locations $\ell_0, \dots, \ell_i$ are either chains that do not contain ℓ_i and that caused p and so $n+1$ is a bound, because n is a bound by induction hypothesis, either they contain ℓ_i and in this case it was a chain in causal relation with p, thus $n+1$ is a bound as n is a bound on the causality between ℓ_0 and ℓ_{i-1} and the last location ℓ_i adds one to the complexity. The case for pattern matching is a simpler version of this case.
  - If $\ell_i = \tau\langle p, p'\rangle$, then by induction hypothesis, the annotation n for the input and m for the output are bounds on some chains of causality on $\ell_0, \dots, \ell_{i-1}$. Let us look at P'_{i+1} given by Lemma 7.8. The only interesting positions are the one with p' as a prefix. All the annotations for those positions are $\max(n, m)$. The only new chain of causality on $\ell_0, \dots, \ell_i$ that end with a location that causes those positions are the ones that finish with ℓ_i. In this case, either they cause ℓ_i, because they cause p or because they cause p'. In both cases, n or m was a bound on the number of ticks by induction hypothesis. So, $\max(n, m)$ is indeed a bound on the number of ticks for all those chains.

This concludes the proof. In the end, the annotation in position p in P_N is a bound on chains of causality that also cause p. Moreover, for any chain of causality, this chain causes its last position (or output position by definition of causality). Thus, all chains of causality are bounded by at least one of the annotations, so the maximum over all annotations is a bound on the causal complexity. This directly gives us that span is an upper bound on causal complexity.

7.3 CausC(P) ≥ Span(P)

Let us work on the converse. To do that, we will restrict a bit the congruence $\equiv$ for annotated processes and expand the semantics $\Rightarrow$ to work with trees. As before, we can define a tree congruence $\equiv_t$ for annotated processes, with the base rules

$$(va)(vb)P \equiv_t (vb)(va)P,$$

$$(va)(P \mid Q) \equiv_t (va)P \mid Q \text{ (when } a \text{ is not free in } Q), \qquad (va)(P \mid Q) \equiv_t P \mid (va)Q \text{ (when } a \text{ is not free in } P),$$

$$n : (P \mid Q) \equiv n : P \mid n : Q, \qquad n : (m : P) \equiv (n + m) : P, \qquad (va)(n : P) \equiv n : ((va)P), \qquad 0 : P \equiv P.$$

Then, we define the semantics $\Rightarrow_t$ exactly as before but with trees instead of simple processes in parallel. An example is given in Figure 19.

As before, any annotated process can be written in a tree representation as in Definition 7.7 using the tree congruence rule $\equiv_t$ for annotated processes. So, from this, it is rather direct that this semantics defined with tree $\Rightarrow_t$ is equivalent to the previously defined $\Rightarrow$, in the sense that they give the same complexity. (It relies in particular on the fact that congruence does not change the span). Now, we can work on this span with tree representation. Formally, we want to prove:

Lemma 7.10. *If P is a process without annotation, with $P(\Rightarrow_t)^*Q$ and $C_\ell(Q) = K$, then there is a computation $(P_i, \ell_i)_{i \le N}$ from P with causal complexity greater than K.*

So, by definition of causal complexity and span, this will indeed show that $\mathrm{CausC}(P) \ge \mathrm{span}(P)$.

As we only need to prove this lemma for processes without annotation, we can take an additional hypothesis on annotated processes: We consider in the following that annotations appear

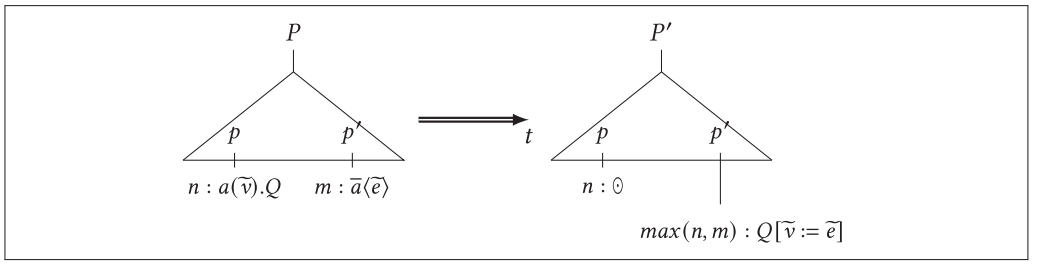


Fig. 19. Tree semantics for span.

exclusively at the "top-level" of a process, so there is no annotation in the subprocess P in $a(\tilde{v}).P$, in $\texttt{tick}.P$, in $!a(\tilde{v}).P$ or in the subprocesses of a pattern matching. We can take this hypothesis, because if a process P satisfies this hypothesis and $P \Rightarrow_t Q$, then Q also satisfies this hypothesis. Of course, processes without annotation satisfy this hypothesis. Still, in this definition, we do not consider $0 : P$ as a real annotation, since it can be removed by congruence. We start by the following definition.

Definition 7.11 (Removing Annotations). Let P be a tree representation of an annotated process. We define forget(P) as the tree P where leaves $n : G$ are replaced by G.

Note that this definition only makes sense because of the previous hypothesis; otherwise, we would need to apply recursively the forgetful function to G. We then easily show the following lemma.

Lemma 7.12 (Forget Reductions). *Suppose that* $P \Rightarrow_t Q$. *Then, we have* $\text{forget}(P) \xrightarrow{\ell} \text{forget}(Q)$ *for some location* ℓ.

Proof. This is a direct consequence of our definition of $\Rightarrow_t$. Note that the reduction $\texttt{tick}.P \Rightarrow_t 1 : P$, or more generally $n : \texttt{tick}.P \Rightarrow_t (n+1) : P$ here corresponds indeed to removing a tick with the forgetful function. □

With this lemma, we can start from forget(P) and simulate the reduction $(\Rightarrow_t)^*$. Now, we only need to show that the annotations are bounded by the number of ticks in a chain of dependency. We show the following lemma:

Lemma 7.13. *If* P *is a process without annotation, and if* $P(\Rightarrow_t)^*Q$, *then there is a computation* $(P_i, \ell_i)_{i \leq N}$ *from* $P = \text{forget}(P)$ *to* forget(Q). *Moreover, for each leaf* $n : G$ *at position* p *in* Q, *there is a chain of causality with at least* n *ticks that ends with a location* ℓ *such that* $\ell \prec_c p$.

Note that from this lemma, we can immediately deduce Lemma 7.10. We prove this by induction on $P(\Rightarrow_t)^*Q$:

- If this relation is the reflexive one, and $P = Q$, then this is direct, because all annotations are equal to 0.
- Now, suppose we have $P(\Rightarrow_t)^*R \Rightarrow_t Q$. By induction hypothesis, there is a computation $(P_i, \ell_i)_{i \leq N}$ from forget(P) to forget(R), with the expected chains of causality. We now proceed by case analysis on $R \Rightarrow Q$.
  - If this reduction is a tick reduction at position p, then we have $\text{forget}(R) \xrightarrow{p;\texttt{tick}} \text{forget}(Q)$. Let us take a leaf $n : G$ at position p' of Q. If p is not a prefix of p', then this leaf was also in R. So, by induction hypothesis, we obtain the desired chain of causality. If p is a prefix of p', then n–1 was the annotation for the position p in R. So, by induction hypothesis

for R, there is a chain of causality with at least $n-1$ ticks that ends with a location ℓ such that $\ell \prec_c p$. By definition, it means that this last location $\ell \prec_c p; \texttt{tick}$. So, this gives us a chain of causality with at least n ticks that ends with a location that causes p' in Q. This concludes this case. The case of pattern matching is a simpler version of this case.

– If this reduction is a synchronization with input at position p and output at position p', then we have $\text{forget}(R) \xrightarrow{\tau\langle p,p'\rangle} \text{forget}(Q)$. Let us take a leaf $n : G$ at position q of Q. We only consider the interesting case: p' is a prefix of q. Then, we have $n = \max(n_0, n_1)$ with n_0 the annotation for the input position p in R and n_1 the annotation for the output position p' in R. Let us say, by symmetry, that n_0 is the maximum between those two. So, by induction hypothesis for R, there is a chain of causality with at least n_0 ticks that ends with a location ℓ such that $\ell \prec_c p$. By definition, it means that this last location $\ell \prec_c \tau\langle p, p'\rangle$. So, this gives us a chain of causality with at least n_0 ticks that ends with a location that causes q in Q. This concludes this case.

This concludes the proof.

We have indeed obtained that causal complexity is an upper bound of span. From this, we have the equivalence between causal complexity and our definition of span.

8 RELATED WORK

An analysis of the complexity of parallel functional programs based on types has been carried out in Reference [26]. Their system can analyse the work and the span (called depth in this article), and makes use of amortized complexity analysis, which allows one to obtain sharp bounds. However, the kind of parallelism they analyse is limited to parallel composition. So, on the one hand, we are considering a more general model of parallelism, and on the other hand, we are not taking advantage of amortized analysis as they do. Reference [20] proposes a complexity analysis of parallel functional programs written in interaction nets, a graph-based language derived from linear logic. Their analysis is based on sized types. However, their model is also quite different from ours as interaction nets do not provide name-passing.

Other works, like Reference [2], tackle the problem of analysing the parallel complexity of a distributed system by building a distributed flow graph and searching for a path of maximal cost in this graph. Another approach to analyse loops with concurrency in an actor-based language is done by *rely-guarantee reasoning* [3]. Those approaches give interesting results on some classes of systems, but they cannot be directly applied to the π-calculus language we are considering, with dynamic creation of processes and channels. Moreover, they do not offer the same compositionality as analysis based on type systems. Reference [19] studies distributed systems that are comparable to those of Reference [2], and analyses their complexity by means of a behaviour type system. In a second step the types are used to run an analysis that returns complexity bounds. Thus, this approach is more compositional than that of Reference [2], but still does not apply to our π-calculus language.

Let us now turn to related works in the setting of π-calculus or process calculi. To our knowledge, the first work to study parallel complexity in π-calculus by types was given by Kobayashi [30], as another application of his type system for deadlock freedom, further developed in other papers [33]. In his setting, channels are typed with *usages*, which are simple CCS-like processes to describe the behaviour of a channel. To carry out complexity analysis, those usages are annotated by two time informations, *obligation* and *capability*. The obligation level is the time at which a channel is ready to perform an action, and the capability level is the time at which it successfully finds a communication partner. We believe that when they are not infinite, the sum of those levels is related to our own time annotation of channels. The definition of parallel complexity in this

work differs from ours, as it loses some non-deterministic paths and the extension with dependent types is suggested but not detailed. It is not clear to us if everything can be adapted to reason only about our parallel complexity, but we plan to study it in future work. More recently, Das et al. in References [12, 13] proposed a type system with temporal session types to capture several parallel cost models with the use of a *tick* constructor. Our usage of time was inspired by their types with the usual *next* modality of temporal logic, but in this paper they also use the *always* and *eventually* modalities to gain expressivity. We believe that because our usage of time is more permissive, those modalities would not be useful in our calculus. Because of session-types, they have linearity for the use of data-types such as lists, but they obtain deadlock-freedom contrary to our calculus. Moreover, they provide decidable operations to simplify the use of their types, such as subtyping, but they do not define dependent types nor sized types that are useful to treat data-types. Still, they provide a significant number of examples to show the expressivity of their type system. In a recent paper [8], a framework was introduced for the cost analysis in multiparty session types. It shares with our approach the use of sized types but it is targeted at communication protocols, which is slightly different from our problematic. It would be interesting however to carry out a closer comparison with our setting.

The methodology of our work is inspired by implicit computational complexity, which aims at characterizing complexity classes by means of dedicated programming languages, mainly in the sequential setting, for instance, by providing languages for FPTIME functions. Some results have been adapted to the concurrent case, but mainly for the work complexity or for other languages than the π-calculus, e.g., References [10, 17, 35] (the first reference is for a higher-order π-calculus). Reference [16] is closer to our setting as it defines a notion of causal complexity in π-calculus and gives a type system characterizing processes with polynomial complexity. However, contrary to those works, we do not restrict to a particular complexity class (like FPTIME) and we handle the case of the span.

Technically, the types we use are inspired from linear dependent types [9]. Those are one of the many variants of sized types, which were introduced in Reference [29].

9 CONCLUSION AND PERSPECTIVES

We believe that the salient contributions of this article are the following:

- a definition of the span in π-calculus by means of a simple reduction semantics (reduction of annotated processes),
- a sized type system for π-calculus, which is here employed for analysing the complexity, but which we think could be used for other applications,
- a natural proof method for establishing the complexity soundness of a type system for span, consisting in proving a subject reduction property for the annotated processes.

We see several possible future directions to this work:

- Type inference: We plan to investigate how type inference could be automatized or partially automatized for the span type system. We will study typing by constraint generation and explore whether existing off-the-shelf solvers or new procedures could allow one to solve these constraints. We have shown here (Section 4.6) that the case of work generates a set of constraints close to those in Reference [4]. However, the case of span could require more sophisticated reasoning because of the strong distinction between servers and channels with the advancing of time.
- We have mentioned that our type system for span is not adapted to analyse some concurrent systems such as the simple example of the semaphore (Section 5.3). However, we believe

that a type system based on an adaptation of usages [30, 32, 33] could be promising for this purpose.
- It would be challenging to examine whether similar type systems could be developed to account for some other complexity properties, for instance, to extract the number of parallel processes needed to achieve the span.

ACKNOWLEDGMENT

We are grateful to Naoki Kobayashi for suggesting the definition of annotated processes and their reduction that we use in this article.

REFERENCES

[1] Selim G. Akl. 2011. *Encyclopedia of Parallel Computing*. Springer, Boston, MA, 139–146.
[2] Elvira Albert, Jesús Correas, Einar Broch Johnsen, and Guillermo Román-Díez. 2015. Parallel cost analysis of distributed systems. In *Proceedings of the 22nd International Symposium on Static Analysis (SAS'15) (Lecture Notes in Computer Science, Vol. 9291)*. Springer, 275–292.
[3] Elvira Albert, Antonio Flores-Montoya, Samir Genaim, and Enrique Martin-Martin. 2017. Rely-guarantee termination and cost analyses of loops with concurrent interleavings. *J. Autom. Reason.* 59, 1 (2017), 47–85.
[4] Martin Avanzini and Ugo Dal Lago. 2017. Automating sized-type inference for complexity analysis. *Proc. ACM Program. Lang.* 1 (2017), 43.
[5] Patrick Baillot and Alexis Ghyselen. 2021. Types for complexity of parallel computation in pi-calculus. In *Proceedings of the 30th European Symposium on Programming Languages and Systems (ESOP'21) (LNCS, Vol. 12648)*. Springer, 59–86.
[6] Patrick Baillot and Virgile Mogbil. 2004. Soft lambda-Calculus: A language for polynomial time computation. In *Proceedings of the 7th International Conference on Foundations of Software Science and Computation Structures (FOSSACS'04) (LNCS, Vol. 2987)*. Springer, 27–41.
[7] Patrick Baillot and Kazushige Terui. 2004. Light types for polynomial time computation in lambda-calculus. In *Proceedings of the 19th IEEE Symposium on Logic in Computer Science (LICS'04)*. IEEE Computer Society, 266–275.
[8] David Castro-Perez and Nobuko Yoshida. 2020. CAMP: Cost-aware multiparty session protocols. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 155:1–155:30.
[9] Ugo Dal Lago and Marco Gaboardi. 2011. Linear dependent types and relative completeness. In *Proceedings of the 26th Annual IEEE Symposium on Logic in Computer Science (LICS'11), Proceedings*. IEEE, 133–142.
[10] Ugo Dal Lago, Simone Martini, and Davide Sangiorgi. 2016. Light logics and higher-order processes. *Math. Struct. Comput. Sci.* 26, 6 (2016), 969–992.
[11] Ornela Dardha, Elena Giachino, and Davide Sangiorgi. 2017. Session types revisited. *Info. Comput.* 256 (2017), 253–286.
[12] Ankush Das, Jan Hoffmann, and Frank Pfenning. 2018. Parallel complexity analysis with temporal session types. *Proc. ACM Program. Lang.* 2, ICFP (2018), 91:1–91:30.
[13] Ankush Das, Jan Hoffmann, and Frank Pfenning. 2018. Work analysis with resource-aware session types. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS'18)*. ACM, 305–314.
[14] Pierpaolo Degano, Fabio Gadducci, and Corrado Priami. 2003. Causality and replication in concurrent processes. In *Perspectives of System Informatics*. Springer, Berlin, 307–318.
[15] Pierpaolo Degano and Corrado Priami. 1995. Causality for mobile processes. In *Automata, Languages, and Programming*. Springer, Berlin, 660–671.
[16] Romain Demangeon and Nobuko Yoshida. 2018. Causal computational complexity of distributed processes. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS'18)*. ACM, 344–353.
[17] Paolo Di Giamberardino and Ugo Dal Lago. 2015. On session types and polynomial time. *Math. Struct. Comput. Sci.* 1 (2015).
[18] Marco Gaboardi, Jean-Yves Marion, and Simona Ronchi Della Rocca. 2008. A logical account of pspace. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'08)*. ACM, 121–131.
[19] Elena Giachino, Einar Broch Johnsen, Cosimo Laneve, and Ka I. Pun. 2016. Time complexity of concurrent programs—A technique based on behavioural types. In *Proceedings of the 12th International Conference on Formal Aspects of Component Software (FACS'15) (Lecture Notes in Computer Science, Vol. 9539)*. Springer, 199–216.
[20] Stéphane Gimenez and Georg Moser. 2016. The complexity of interaction. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'16)*. 243–255.
[21] Emmanuel Hainry, Jean-Yves Marion, and Romain Péchoux. 2013. Type-based complexity analysis for fork processes. In *Proceedings of the 16th International Conference on Foundations of Software Science and Computation Structures (FOSSACS'13) (Lecture Notes in Computer Science, Vol. 7794)*. Springer, 305–320.

[22] Robert Harper. 2012. *Practical Foundations for Programming Languages*. Cambridge University Press.
[23] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. 2012. Multivariate amortized resource analysis. *ACM Trans. Program. Lang. Syst.* 34, 3 (2012), 14:1–14:62.
[24] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. 2012. Resource aware ML. In *Proceedings of the 24th International Conference on Computer Aided Verification (CAV'12) (Lecture Notes in Computer Science, Vol. 7358)*. Springer, 781–786.
[25] Jan Hoffmann and Martin Hofmann. 2010. Amortized resource analysis with polynomial potential. In *Proceedings of the 19th European Symposium on Programming on Programming Languages and Systems (ESOP'10) (Lecture Notes in Computer Science, Vol. 6012)*. Springer, 287–306.
[26] Jan Hoffmann and Zhong Shao. 2015. Automatic static cost analysis for parallel programs. In *Programming Languages and Systems*. Springer, Berlin, 132–157.
[27] Martin Hofmann. 2003. Linear types and non-size-increasing polynomial time computation. *Info. Comput.* 183, 1 (2003), 57–85.
[28] Martin Hofmann and Steffen Jost. 2003. Static prediction of heap space usage for first-order functional programs. In *Proceedings of the 30th SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'03)*. ACM, 185–197.
[29] John Hughes, Lars Pareto, and Amr Sabry. 1996. Proving the correctness of reactive systems using sized types. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 410–423.
[30] Naoki Kobayashi. 2002. A type system for lock-free processes. *Info. Comput.* 177, 2 (2002), 122 – 159.
[31] Naoki Kobayashi. 2003. Type systems for concurrent programs. In *Formal Methods at the Crossroads: From Panacea to Foundational Support*. Springer, 439–453.
[32] Naoki Kobayashi. 2005. Type-based information flow analysis for the π-calculus. *Acta Informatica* 42, 4–5 (2005), 291–347.
[33] Naoki Kobayashi. 2006. A new type system for deadlock-free processes. In *Proceedings of the International Conference on Concurrency Theory*. Springer, 233–247.
[34] Naoki Kobayashi, Benjamin C. Pierce, and David N. Turner. 1999. Linearity and the pi-calculus. *ACM Trans. Program. Lang. Syst.* 21, 5 (Sep. 1999), 914–947.
[35] Antoine Madet and Roberto M. Amadio. 2011. An elementary affine λ-calculus with multithreading and side effects. In *Proceedings of the 10th International Conference on Typed Lambda Calculi and Applications (TLCA'11) (Lecture Notes in Computer Science, Vol. 6690)*. Springer, 138–152.
[36] Jean-Yves Marion. 2011. A type system for complexity flow analysis. In *Proceedings of the 26th Annual IEEE Symposium on Logic in Computer Science, (LICS'11)*. IEEE Computer Society, 123–132.
[37] Robin Milner, Joachim Parrow, and David Walker. 1992. A calculus of Mobile processes, I. *Info. Comput.* 100, 1 (1992), 1–40.
[38] Davide Sangiorgi and David Walker. 2003. *The Pi-calculus: A Theory of Mobile Processes*. Cambridge University Press.
[39] Pedro B. Vasconcelos. 2008. *Space cost analysis using sized types*. Ph.D. Dissertation. University of St. Andrews.

Received March 2021; revised September 2021; accepted November 2021