Sized Types with Usages for Parallel Complexity of Pi-Calculus Processes

Patrick Baillot
Univ Lyon, CNRS, ENS de Lyon, Universite Claude-Bernard Lyon 1, LIP, F-69342, France

Alexis Ghyselen
Univ Lyon, CNRS, ENS de Lyon, Universite Claude-Bernard Lyon 1, LIP, F-69342, France

Naoki Kobayashi
The University of Tokyo, Japan

Abstract

We address the problem of analysing the complexity of concurrent programs written in Pi-calculus. We are interested in parallel complexity, or span, understood as the execution time in a model with maximal parallelism. A type system for parallel complexity has been recently proposed by the first two authors but it is too imprecise for non-linear channels and cannot analyse some concurrent processes. Aiming for a more precise analysis, we design a type system which builds on the concepts of sized types and usages. The sized types allow us to parametrize the complexity by the size of inputs, and the usages allow us to achieve a kind of rely-guarantee reasoning on the timing each process communicates with its environment. We prove that our new type system soundly estimates the parallel complexity, and show through examples that it is often more precise than the previous type system of the first two authors.

2012 ACM Subject Classification Theory of computation → Type structures; Theory of computation → Process calculi; Software and its engineering → Software verification

Keywords and phrases Type Systems, Pi-calculus, Process Calculi, Complexity Analysis, Usages, Sized Types

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2021.34

Related Version *Extended Version*: https://arxiv.org/abs/2104.07293

Funding This work was partially supported by the LABEX MILYON (ANR-10-LABX-0070) of Universite de Lyon and JSPS KAKENHI Grant Number JP20H05703.

Acknowledgements We would like to thank anonymous referees for useful comments.

1 Introduction

Static analysis of complexity is a classic topic of program analysis, and various approaches to the complexity analysis, including type-based ones [3, 6, 17–19, 21], have been studied so far. The complexity analysis of concurrent programs has been, however, much less studied.

In this paper, we are interested in analysing the parallel complexity (also called *span*) of the π-calculus, i.e., the maximal parallelized execution time under the assumption that an unlimited number of processors are available [4]. The parallel complexity should be parametrized by the size of inputs. Following the success of previous studies on the complexity analysis of sequential programs [3, 6, 17–19, 21] and those on the analysis of other properties on concurrent programs (e.g. [11, 13]), deadlock-freedom and livelock-freedom (e.g. [22, 27, 30]; see [24] for a survey), we take a type-based approach.

The first two authors [4] have actually proposed a type-based analysis of the parallel complexity already. However, as stressed by the authors, even though their type system for span is useful for analysing the complexity of some parallel programs, it fails to type-

check some examples of common concurrent programs, like semaphores. It is based on a combination of sized types and input/output types, in order to account suitably for the behaviour of channels w.r.t. reception and emission.

In the present paper we design a type system for span which can deal with a much wider range of concurrent computation patterns including the semaphores. For that, we take inspiration from the notion of *type usage*, which has been introduced and explored in [27, 30], initially to guarantee absence of deadlock during execution. Type usages are a generalization of input/output types, and describe how each channel is used for input and output. Unlike the original notion of usages [27, 30], our usages are annotated with time intervals, which are used for a kind of rely-guarantee reasoning, like "assuming that a message from the environment arrives during the time interval $[I_1, J_1]$, the process sends back a message during the interval $[I_2, J_2]$"; such reasoning is crucial for analyzing the parallel complexity precisely and in a compositional manner. We formalize the type system with usages and prove that it soundly estimates the parallel complexity.

Contributions. The contributions of this paper are as follows. (i) The formalization of the new type system for parallel complexity built on the new notion of usages: our usages are quite different from the original ones, and properly defining them (including operations on time intervals, usage reductions, and the notion of reliable usages) is non-trivial. (ii) The proofs of type preservation and complexity soundness: thanks to the careful definition of new usages, the proofs are actually quite natural, despite the expressiveness of the type system. (iii) Examples to demonstrate the precision and expressive power of our new type system.

Paper outline. We introduce in Sect. 2 the π-calculus and the notion of parallel complexity we consider. Sect. 3 is devoted to the definition of types with usages. Then in Sect. 4 we prove the main result of this paper, the complexity soundness, and provide some examples. Finally, related work is discussed in Sect. 5.

2 The Pi-calculus with Semantics for Span

In this section, we review the definitions of the π-calculus and its parallel complexity [4].

2.1 Syntax and Standard Semantics for Pi-Calculus

We consider a synchronous π-calculus, with a constructor `tick` that generates the time complexity. The sets of *variables*, *expressions* and *processes* are defined by:

$$\begin{aligned} v \text{ (variables)} &:= x, y, z \mid a, b, c \qquad e \text{ (expressions)} := v \mid \mathtt{0} \mid \mathtt{s}(e) \\ P \text{ (processes)} &:= 0 \mid (P \mid Q) \mid a(\widetilde{v}).P \mid !a(\widetilde{v}).P \mid \overline{a}\langle\widetilde{e}\rangle.P \mid (\nu a)P \\ &\quad \mid \mathtt{match}\ e\ \{\mathtt{case}\ \mathtt{0} \mapsto P; \mathtt{case}\ \mathtt{s}(x) \mapsto Q\} \mid \mathtt{tick}.P \end{aligned}$$

We use x, y, z as meta-variables for integer variables, and a, b, c as those for channel names. The notation $\widetilde{v}$ stands for a sequence of variables $v_1, v_2, \ldots, v_k$. Similarly, $\widetilde{e}$ denotes a sequence of expressions. We work up to α-renaming, and write $P[\widetilde{v} := \widetilde{e}]$ to denote the substitution of $\widetilde{e}$ for the free variables $\widetilde{v}$ in P. For simplicity, we only consider integers as base types below, but the results can be generalized to other algebraic data-types such as lists or booleans.

Intuitively, $P \mid Q$ stands for the parallel composition of P and Q. The process $a(\widetilde{v}).P$ 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 $a(\widetilde{v}).P$:

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

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

$$\frac{}{\texttt{match 0 } \{\texttt{case 0} \mapsto P; \texttt{case s}(x) \mapsto Q\} \Rightarrow P}$$

$$\frac{}{\texttt{match s}(e) \ \{\texttt{case 0} \mapsto P; \texttt{case s}(x) \mapsto Q\} \Rightarrow Q[x := e]}$$

$$\frac{P \Rightarrow Q}{P \mid R \Rightarrow Q \mid R} \qquad \frac{P \Rightarrow Q}{(\nu a)P \Rightarrow (\nu a)Q} \qquad \frac{P \Rightarrow Q}{(n : P) \Rightarrow (n : Q)}$$

$$\frac{P \equiv P' \qquad P' \Rightarrow Q' \qquad Q' \equiv Q}{P \Rightarrow Q}$$

Figure 1 Reduction Rules for Annotated Processes.

it behaves like an infinite number of $a(\widetilde{v}).P$ in parallel. The process $\overline{a}\langle\widetilde{e}\rangle.P$ represents an output: it sends a sequence of expressions $\widetilde{e}$ on the channel a, and continues as P. We often omit 0 and just write $\overline{a}\langle\rangle$ for $\overline{a}\langle\rangle.0$. A process $(\nu a)P$ dynamically creates a new channel name a and then proceeds as P. We also have standard pattern matching on data types, and finally, the `tick` constructor incurs a cost of one in complexity but has no semantic relevance. This constructor is the only source of time complexity in a program. As the similar tick constructor in [9], it can represent different cost models and is more general than counting the number of reduction steps. For example, by adding `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. We can also count the number of reduction steps, by adding `tick` after each input and pattern matching.

As usual, the structural congruence $\equiv$ is defined as the least congruence containing:

$$P \mid 0 \equiv P \qquad P \mid Q \equiv Q \mid P \qquad P \mid (Q \mid R) \equiv (P \mid Q) \mid R$$

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

Note that the last rule can always be applied from right to left by α-renaming. By associativity, we will often write parallel composition for any number of processes and not only two.

2.2 Parallel Complexity: The Span

We now review the definition of span [4]. We add a process construct $m : P$, 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*. The congruence relation $\equiv$ is then enriched with the following relations:

$$0 : P \equiv P \qquad m : (P \mid Q) \equiv (m : P) \mid (m : Q)$$

$$m : (\nu a)P \equiv (\nu a)(m : P) \qquad m : (n : P) \equiv (m + n) : P$$

So, zero tick is equivalent to nothing and ticks can be distributed over parallel composition as expressed by the second relation. Name creation can be done before or after ticks without changing the semantics and finally ticks can be grouped together.

The rules for the reduction relation $\Rightarrow$ are given in Figure 1. This semantics works as the usual semantics for π-calculus, but when doing a synchronization, only the maximal annotation is kept, and ticks are memorized in the annotations. Span is then defined by:

▶ **Definition 1** (Parallel Complexity). *The* local complexity $\mathcal{C}_\ell(P)$ *of an annotated process* P *is defined by:*

$$\mathcal{C}_\ell(n : P) = n + \mathcal{C}_\ell(P) \qquad \mathcal{C}_\ell(P \mid Q) = \max(\mathcal{C}_\ell(P), \mathcal{C}_\ell(Q))$$

$$\mathcal{C}_\ell((\nu a)P) = \mathcal{C}_\ell(P) \qquad \mathcal{C}_\ell(P) = 0 \text{ otherwise}$$

The global parallel complexity *(or* span*) of* P *is given by* $\max\{n \mid P \Rightarrow^* Q \wedge \mathcal{C}_\ell(Q) = n\}$ *where* $\Rightarrow^*$ *is the reflexive and transitive closure of* $\Rightarrow$.

▶ **Example 2.** Let $P := \mathtt{tick}.a().\mathtt{tick}.\overline{a}\langle\rangle \mid \mathtt{tick}.a().\mathtt{tick}.\overline{a}\langle\rangle \mid \overline{a}\langle\rangle$. Then, we have:

$$\begin{aligned} P &\Rightarrow^2 1 : (a().\mathtt{tick}.\overline{a}\langle\rangle) \mid 1 : (a().\mathtt{tick}.\overline{a}\langle\rangle) \mid 0 : \overline{a}\langle\rangle \Rightarrow 1 : (a().\mathtt{tick}.\overline{a}\langle\rangle) \mid 1 : (\mathtt{tick}.\overline{a}\langle\rangle) \\ &\Rightarrow 1 : (a().\mathtt{tick}.\overline{a}\langle\rangle) \mid 2 : \overline{a}\langle\rangle \Rightarrow 2 : (\mathtt{tick}.\overline{a}\langle\rangle) \Rightarrow 3 : \overline{a}\langle\rangle \end{aligned}$$

Thus, the process has at least complexity 3. As all the other possible choices we could have made in the reduction steps are similar, the process has exactly complexity 3.

The following example motivates our introduction of usages in the next section.

▶ **Example 3** (Motivating Example). Let $P := a().\mathtt{tick}.\overline{a}\langle\rangle$. Then, the complexity of $P \mid P \mid P \mid \cdots \mid P \mid \overline{a}\langle\rangle$ is equal to the number of P in parallel.

3 Types with Usages

The goal of our work is to design a type system for processes such that if $\Gamma \vdash Q \triangleleft K$ then K is a bound on the complexity of Q, as in [4]. The analysis of [4] was not precise enough: in fact, the process P in Example 3 was not typable. The main idea to tackle this problem is to use the notion of usages to represent the channel-wise behaviour of processes. Usages have been used for deadlock-freedom analysis [23, 27, 30], but our notion of usages significantly differs from the original one, as discussed below.

3.1 Indices

First, we define integer indices, which are used to keep track of the size of values in a process.

▶ **Definition 4.** *Let* $\mathcal{V}$ *be a countable set of index variables, usually denoted by* i,j *or* k. *The set of* indices, *representing integers in* $\mathbb{N}_\infty = \mathbb{N} \cup \{\infty\}$, *is given by:*

$$I, J := I_{\mathbb{N}} \mid \infty \qquad I_{\mathbb{N}} := i \mid f(I_{\mathbb{N}}, \ldots, I_{\mathbb{N}})$$

Here, $i \in \mathcal{V}$. *The symbol* f *is an element of a given set of function symbols containing, for example, integers constants as nullary operators, addition and multiplication. We also assume the subtraction as a function symbol, with* $n - m = 0$ *when* $m \geq n$. *Each function symbol* f *of arity* $\mathtt{ar}(f)$ *comes with an interpretation* $[\![f]\!] : \mathbb{N}^{\mathtt{ar}(f)} \to \mathbb{N}$.

Given an index valuation $\rho : \mathcal{V} \to \mathbb{N}$, we extend the interpretation of function symbols to indices, noted $[\![I]\!]_\rho$, as expected; $[\![I]\!]_\rho$ ranges over $\mathbb{N}_\infty$. For an index I, we write $I\{J_{\mathbb{N}}/i\}$ for the index obtained by replacing the occurrences of i in I with $J_{\mathbb{N}}$. Note that $\infty\{J_{\mathbb{N}}/i\} = \infty$.

▶ **Definition 5** (Constraints on Indices). *Let* $\varphi \subset \mathcal{V}$ *be a finite 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* $\mathbb{N}_\infty$. *Usually, we take* $\bowtie \in \{\leq, <, =, \neq\}$. *A finite set of constraints is denoted* Φ.

For a finite 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 $\rho \vDash C$. We will also use some extended operations on indices I, J; for example, we may use $\infty + J = \infty$ or other such equations.

3.2 Usages

We use *usages* to express the channel-wise behaviour of a process. Usages are a kind of CCS processes [29] on a single channel, where each action is annotated with two time intervals. The set of usages, ranged over by U and V, is given by:

$$U, V ::= 0 \mid (U|V) \mid \alpha^{A_o}_{J_c}.U \mid\ !U \mid U + V \qquad \alpha := \mathtt{In} \mid \mathtt{Out}$$

$$A_o, B_o ::= [I, J] \qquad J_c, I_c ::= J \mid [I, J]$$

Given a set of index variables φ and a set of constraints Φ, for an interval $[I, J]$, we always require that $\varphi; \Phi \vDash I \leq J$. For an interval $A_o = [I, J]$, we denote $\mathsf{Left}(A_o) = I$ and $\mathsf{Right}(A_o) = J$. In the original notion of usages [23, 27, 30], A_o and J_c were just numbers. The extension to intervals plays an important role in our analysis. Note that J_c may be a single index J, but this single index J should be understood as the interval $[-\infty, J]$.

Intuitively, a channel with usage 0 is not used at all. A channel of usage $U \mid V$ can be used according to U and V possibly in parallel. The usage $\mathtt{In}^{A_o}_{J_c}.U$ describes a channel that may be used for input, and then used according to U. The two intervals A_o and J_c, called *obligation* and *capacity* respectively, are used to achieve a kind of assume-guarantee reasoning. The obligation A_o indicates a *guarantee* that if the channel is indeed used for input, then the input should become ready during the interval A_o. The capacity J_c indicates the *assumption* that if the environment performs a corresponding output, that output will be provided during the time interval J_c after the input becomes ready. For example, if a channel a has usage $\mathtt{In}^{[1,1]}_{J_c}.0$, then the process $\mathtt{tick}.a().0$ conforms to the usage, but $a().0$ and $\mathtt{tick}.\mathtt{tick}.a().0$ do not. Similarly, $\mathtt{Out}^{A_o}_{J_c}.U$ has the same meaning but for output. The usage $!U$ denotes the usage U that can be replicated infinitely, and $U + V$ denotes a non-deterministic choice between the usages U and V. This is useful for example in a case of pattern matching where a channel can be used very differently in the two branches.

Recall that the obligation and capacity intervals in usages express a sort of assume-guarantee reasoning. We thus require that the assume-guarantee reasoning in a usage is "consistent" (or *reliable*, in the terminology of usages). For example, the usage $\mathtt{In}^{[0,0]}_{[1,1]} \mid \mathtt{Out}^{[1,1]}_{0}$ is reliable because (i) the part $\mathtt{In}^{[0,0]}_{[1,1]}$ assumes that a corresponding output will become ready at time 1, and the other part $\mathtt{Out}^{[1,1]}_{0}$ indeed guarantees that and moreover, (ii) $\mathtt{Out}^{[1,1]}_{0}$ assumes that a corresponding input will be ready by the time the output becomes ready, and the part $\mathtt{In}^{[0,0]}_{[1,1]}$ guarantees that. In contrast, the usage $\mathtt{In}^{[0,0]}_{[1,1]} \mid \mathtt{Out}^{[2,2]}_{0}$ is problematic because, although the part $\mathtt{In}^{[0,0]}_{[1,1]}$ assumes that an output will be ready at time 1, $\mathtt{Out}^{[2,2]}_{0}$ provides the output only at time 2. The consistency on assume-guarantee reasoning must hold during the whole computation; for example, in the usage $\mathtt{In}^{[0,0]}_{[0,0]}.\mathtt{In}^{[0,0]}_{[1,1]} \mid \mathtt{Out}^{[0,0]}_{[0,0]}.\mathtt{Out}^{[2,2]}_{0}$, the first input/output pair is fine, but the usage expressing the next communication: $\mathtt{In}^{[0,0]}_{[1,1]} \mid \mathtt{Out}^{[2,2]}_{0}$ is problematic. To properly define the reliability of usages during the whole computation, we first prepare a reduction semantics for usages, by viewing usages as CCS processes.

▶ **Definition 6** (Congruence for Usages). *The relation $\equiv$ is defined as the least congruence relation closed under:*

$$U \mid 0 \equiv U \qquad U \mid V \equiv V \mid U \qquad U \mid (V \mid W) \equiv (U \mid V) \mid W$$

$$!0 \equiv 0 \qquad !U \equiv !U \mid U \qquad !(U \mid V) \equiv !U \,|\, !V \qquad !!U \equiv !U$$

Before giving the reduction semantics, we introduce some notations.

▶ **Definition 7** (Operations on Usages). *We define the operations $\oplus$, $\sqcup$, and $+$ by:*

$$A_o \oplus J = [0, \mathsf{Left}(A_o) + J] \qquad A_o \oplus [I, J] = [\mathsf{Right}(A_o) + I, \mathsf{Left}(A_o) + J]$$
$$[I, J] \sqcup [I', J'] = [\max(I, I'), \max(J, J')] \qquad [I, J] + [I', J'] = [I + I', J + J']$$

Note that $\oplus$ is an operation that takes an obligation interval and a capacity and returns an interval. The operations $\sqcup$ (max) and $+$ are just pointwise extensions of the operations for indices. The intuition on $\oplus$ is explained later when we define the reduction relation.

The delaying operation $\uparrow^{A_o} U$ on usages is defined by:

$$\uparrow^{A_o} 0 = 0 \qquad \uparrow^{A_o} (U \mid V) = \uparrow^{A_o} U \mid \uparrow^{A_o} V \qquad \uparrow^{A_o} (U + V) = \uparrow^{A_o} U + \uparrow^{A_o} V$$

$$\uparrow^{A_o} \alpha^{B_o}_{J_c}.U = \alpha^{A_o + B_o}_{J_c}.U \qquad \uparrow^{A_o} (!U) = !(\uparrow^{A_o} U)$$

We also define $[I, J] + J_c$ and thus $\uparrow^{J_c} U$ by extending the operation with: $[I, J] + J' = [I, J + J']$.

Intuitively, a usage $\uparrow^{A_o} U$ corresponds to the usage U delayed by a time approximated by the interval A_o.

The reduction relation is given by the rules of Figure 2. The first rule means that to reduce a usage, we choose one input and one output, and then we trigger the communication between them. This communication occurs and does not lead to an error when the capacity of an action indeed corresponds to a bound on the time the dual action is defined. This is given by the relation $A_o \subseteq B_o \oplus J_c$. As an example, let us suppose that $B_o = [1, 3]$, and the time for which the output becomes ready is in fact 2, then the capacity J_c says that after two units of time, the synchronization should happen in the interval J_c. So, if we take $J_c = [5, 7]$ for example, then if t is the time for which the dual input becomes ready, we must have $t \in [2 + 5, 2 + 7]$. This should be true for any time value in B_o, so we want that $\forall t' \in [1, 3], \forall t \in A_o, t \in [t' + 5, t' + 7]$, and this is equivalent to $A_o \subseteq B_o \oplus [5, 7] = [8, 8]$. Indeed, 8 is the only time that is in the three intervals $[6, 8]$, $[7, 9]$ and $[8, 10]$. The case where $J_c = J$ is a single index occurs when t can be smaller than t', and in this case we only ask that the upper bound is correct: $\forall t' \in B_o, \forall t \in A_o, t \leq t' + J$.

If the bound is incorrect, we trigger an error: see the second rule. In the case everything went well, the continuation is delayed by an approximation of the time when this communication occurs (see $\uparrow^{A_o \sqcup B_o}$ in the first rule). In the rules for $U + V$, a reduction step in usages can also make a non-deterministic choice.

An error in a usage reduction means that the assume-guarantee reasoning was inconsistent. Based on this intuition, we define the notion of reliablity.

▶ **Definition 8** (Reliability). *A usage U is* reliable *under $\varphi; \Phi$ if $U \not\longrightarrow^* \mathtt{err}$.*

▶ **Example 9.** Consider the usage $U := \mathtt{In}^{[1,1]}_1.\mathtt{Out}^{[1,1]}_0 \mid \mathtt{In}^{[1,1]}_1.\mathtt{Out}^{[1,1]}_0 \mid \mathtt{Out}^{[0,0]}_{[1,1]}$. The only possible reduction sequence (with symmetry) is:

$$U \longrightarrow \mathtt{Out}^{[2,2]}_0 \mid \mathtt{In}^{[1,1]}_1.\mathtt{Out}^{[1,1]}_0 \longrightarrow \mathtt{Out}^{[3,3]}_0.$$

$$\frac{\varphi;\Phi \vDash B_o \subseteq A_o \oplus I_c \qquad \varphi;\Phi \vDash A_o \subseteq B_o \oplus J_c}{\varphi;\Phi \vdash \mathtt{In}^{A_o}_{I_c}.U \mid \mathtt{Out}^{B_o}_{J_c}.V \longrightarrow \uparrow^{A_o \sqcup B_o}(U \mid V)} \qquad \frac{\varphi;\Phi \nvDash (B_o \subseteq A_o \oplus I_c \wedge A_o \subseteq B_o \oplus J_c)}{\varphi;\Phi \vdash \mathtt{In}^{A_o}_{I_c}.U \mid \mathtt{Out}^{B_o}_{J_c}.V \longrightarrow \mathtt{err}}$$

$$\frac{}{\varphi;\Phi \vdash U + V \longrightarrow U} \qquad \frac{}{\varphi;\Phi \vdash U + V \longrightarrow V} \qquad \frac{\varphi;\Phi \vdash U \longrightarrow U' \qquad U' \neq \mathtt{err}}{\varphi;\Phi \vdash U \mid V \longrightarrow U' \mid V}$$

$$\frac{\varphi;\Phi \vdash U \longrightarrow \mathtt{err}}{\varphi;\Phi \vdash U \mid V \longrightarrow \mathtt{err}} \qquad \frac{U \equiv U' \qquad \varphi;\Phi \vdash U' \longrightarrow V' \qquad V' \equiv V}{\varphi;\Phi \vdash U \longrightarrow V}$$

Figure 2 Reduction Rules for Usages.

For the first step, we have indeed $[1,1] \subseteq [0,0] \oplus [1,1] = [1,1]$ and $[0,0] \subseteq [1,1] \oplus 1 = [0,2]$. Note that the capacity $[0,1]$ instead of 1 for the input would not have worked since $[1,1] \oplus [0,1] = [1,2]$. Thus, the usage U is reliable. It corresponds, for example, to the usage of the channel a in the process P given in Example 2: $\mathtt{tick}.a().\mathtt{tick}.\overline{a}\langle\rangle \mid \mathtt{tick}.a().\mathtt{tick}.\overline{a}\langle\rangle \mid \overline{a}\langle\rangle$. The obligation $[1,1]$ corresponds to waiting for exactly one tick. Then, the capacities say that once they are ready, the two inputs will indeed communicate before one time unit for any reduction. And at the end, we obtain an output available at time 3, and this output has no communication. One can see that those capacities and obligations indeed give the complexity of this process. Thus, we will ask in the type system that all usages are reliable, and so the time indications will give some complexity bounds on the behaviour of a channel. ⌟

▶ **Example 10.** We give an example of a non-reliable usage. To the previous example, let us add another input in parallel

$$U := \mathtt{In}_1^{[1,1]}.\mathtt{Out}_0^{[1,1]} \mid \mathtt{In}_1^{[1,1]}.\mathtt{Out}_0^{[1,1]} \mid \mathtt{In}_1^{[1,1]}.\mathtt{Out}_0^{[1,1]} \mid \mathtt{Out}_{[1,1]}^{[0,0]}$$

We have: $U \longrightarrow^* \mathtt{Out}_0^{[3,3]} \mid \mathtt{In}_1^{[1,1]}.\mathtt{Out}_0^{[1,1]} \longrightarrow \mathtt{err}$, because $[1,1] \oplus 1 = [0,2]$, so the capacity here is not a good assumption. However, the following variant of the usage:

$$U := \mathtt{In}_2^{[1,1]}.\mathtt{Out}_0^{[1,1]} \mid \mathtt{In}_2^{[1,1]}.\mathtt{Out}_0^{[1,1]} \mid \mathtt{In}_2^{[1,1]}.\mathtt{Out}_0^{[1,1]} \mid \mathtt{Out}_{[1,1]}^{[0,0]}$$

is reliable. This example shows how reliability adapts to parallel composition. ⌟

We introduce another relation $U \sqsubseteq V$ called the *subusage* relation, which will be used later to define the subtyping relation. It is defined by the rules of Figure 3. The relation $U \sqsubseteq V$ intuitively means that any channel of usage U may also be used according to V. For example, $U \sqsubseteq 0$ says that we may not use a channel (usage equal to 0). Recall that an obligation and a capacity express a guarantee and an assumption respectively. The last but one rule says that it is safe to strengthen the guarantee and weaken the assumption. We use the relation $I_c \leq J_c$ to denote the relation $\subseteq$ on intervals, where a single index J is considered as the interval $[-\infty, J]$. The last rule can be understood as follows. The part $\uparrow^{A_o+J_c} V$ says that a channel may be used according to V only after the interval $A_o + J_c$. Since the action $\alpha^{A_o}_{J_c}$ is indeed finished during the interval $A_o + J_c$, we can move V to under the guard of $\alpha^{A_o}_{J_c}$. This last rule is especially useful for substitution, as explained in the example below.

▶ **Example 11.** Consider the process:

$$P := a(r).r().b() \mid \overline{a}\langle b\rangle$$

Let us give usages to b and r; here we omit time annotations for the sake of simplicity. We have $U_r = \mathtt{In}$ and $U_b = \mathtt{In} \mid U_r$ Indeed, r is used only once as an input, and b is used as an input on the left, and it is sent to be used as r on the right. Thus, after a reduction step we

$$\frac{}{\varphi;\Phi \vdash U \sqsubseteq 0} \qquad \frac{i \in \{1;2\}}{\varphi;\Phi \vdash U_1 + U_2 \sqsubseteq U_i} \qquad \frac{\varphi;\Phi \vdash U \sqsubseteq U'}{\varphi;\Phi \vdash U + V \sqsubseteq U' + V}$$

$$\frac{\varphi;\Phi \vdash V \sqsubseteq V'}{\varphi;\Phi \vdash U + V \sqsubseteq U + V'} \qquad \frac{\varphi;\Phi \vdash U \sqsubseteq U'}{\varphi;\Phi \vdash U \mid V \sqsubseteq U' \mid V} \qquad \frac{\varphi;\Phi \vdash U \sqsubseteq U'}{\varphi;\Phi \vdash !U \sqsubseteq !U'}$$

$$\frac{U \equiv U' \qquad \varphi;\Phi \vdash U' \sqsubseteq V' \qquad V \equiv V'}{\varphi;\Phi \vdash U \sqsubseteq V} \qquad \frac{\varphi;\Phi \vdash U \sqsubseteq U' \qquad \varphi;\Phi \vdash U' \sqsubseteq U''}{\varphi;\Phi \vdash U \sqsubseteq U''}$$

$$\frac{\varphi;\Phi \vdash U \sqsubseteq U'}{\varphi;\Phi \vdash \alpha^{A_o}_{J_c}.U \sqsubseteq \alpha^{A_o}_{J_c}.U'} \qquad \frac{\varphi;\Phi \vDash B_o \subseteq A_o \qquad \varphi;\Phi \vDash I_c \leq J_c}{\varphi;\Phi \vdash \alpha^{A_o}_{I_c}.U \sqsubseteq \alpha^{B_o}_{J_c}.U}$$

$$\frac{}{\varphi;\Phi \vdash (\alpha^{A_o}_{J_c}.U) \mid (\uparrow^{A_o+J_c} V) \sqsubseteq \alpha^{A_o}_{J_c}.(U \mid V)}$$

Figure 3 Subusage.

$$\frac{\varphi;\Phi \vDash I' \leq I \qquad \varphi;\Phi \vDash J \leq J'}{\varphi;\Phi \vdash \mathsf{Nat}[I,J] \sqsubseteq \mathsf{Nat}[I',J']} \qquad \frac{\varphi;\Phi \vdash \widetilde{T} \sqsubseteq \widetilde{T'} \qquad \varphi;\Phi \vdash \widetilde{T'} \sqsubseteq \widetilde{T} \qquad \varphi;\Phi \vdash U \sqsubseteq V}{\varphi;\Phi \vdash \mathtt{ch}(\widetilde{T})/U \sqsubseteq \mathtt{ch}(\widetilde{T'})/V}$$

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

Figure 4 Subtyping Rules for Usage Types.

obtain $P \to b().b()$ where b has usage $U'_b = \mathtt{In}.\mathtt{In}$. So, the channel b had usage U_b in P, but it ended up being used according to U'_b; that is valid since we have the subusage relation $U_b \sqsubseteq U'_b$.

3.3 Type System

We extend ordinary types for the π-calculus with usages.

▶ **Definition 12** (Usage Types). *We define* types *by the following grammar:*

$$T, S ::= \mathsf{Nat}[I,J] \mid \mathtt{ch}(\widetilde{T})/U \mid \forall \widetilde{i}.\mathtt{srv}^K(\widetilde{T})/U.$$

The type $\mathsf{Nat}[I,J]$ describes an integer n such that $I \leq n \leq J$. Channels are classified into *server channels* (or just *servers*) and *simple channels*. All the inputs on a server channel must be replicated (as in $!a(\widetilde{v}).P$), while no input on a simple channel can be replicated. The type $\mathtt{ch}(\widetilde{T})/U$ describes a simple channel that is used for transmitting values of type $\widetilde{T}$ according to usage U. For example, $\mathtt{ch}(\mathsf{Nat}[I,J])/U$ is the type of channels used according to U for transmitting integers in the interval $[I,J]$. The type $\forall \widetilde{i}.\mathtt{srv}^K(\widetilde{T})/U$ describes a server channel that is used for transmitting values of type $\widetilde{T}$ according to usage U; the superscript K, which we call the *complexity* of a server, is an interval. It denotes the cost incurred when a server is invoked. Note that the server type allows polymorphism on index variables $\widetilde{i}$.

The subtyping relation $T \sqsubseteq T'$, which means that a value of type T can also be used as a value of type T', is defined by the rules of Figure 4.

We extend operations on usages to partial operations on types and typing contexts with $\Gamma = v_1 : T_1, \ldots, v_n : T_n$. The delaying of a type $\uparrow^{A_o} T$ is defined as the delaying of the usage for a channel or a server type, and it does nothing on integers. We also say that a type is *reliable* when it is an integer type, or when it is a server or channel type with a reliable usage. We define following operations:

$$\frac{v : T \in \Gamma}{\varphi; \Phi; \Gamma \vdash v : T} \qquad \frac{}{\varphi; \Phi; \Gamma \vdash 0 : \mathsf{Nat}[0,0]} \qquad \frac{\varphi; \Phi; \Gamma \vdash e : \mathsf{Nat}[I,J]}{\varphi; \Phi; \Gamma \vdash \mathtt{s}(e) : \mathsf{Nat}[I+1, J+1]}$$

$$\frac{\varphi; \Phi; \Delta \vdash e : T' \qquad \varphi; \Phi \vdash \Gamma \sqsubseteq \Delta \qquad \varphi; \Phi \vdash T' \sqsubseteq T}{\varphi; \Phi; \Gamma \vdash e : T}$$

Figure 5 Typing Rules for Expressions.

▶ **Definition 13.** *The parallel composition $T \mid T'$ is defined by:*

$$\mathsf{Nat}[I,J] \mid \mathsf{Nat}[I,J] = \mathsf{Nat}[I,J] \qquad \mathit{ch}(\widetilde{T})/U \mid \mathit{ch}(\widetilde{T})/V = \mathit{ch}(\widetilde{T})/(U \mid V)$$

$$\forall\widetilde{i}.\mathit{srv}^K(\widetilde{T})/U \mid \forall\widetilde{i}.\mathit{srv}^K(\widetilde{T})/V = \forall\widetilde{i}.\mathit{srv}^K(\widetilde{T})/(U \mid V)$$

▶ **Definition 14** (Replication of Type)**.** *The replication of a type $!T$ is defined by:*

$$!\mathsf{Nat}[I,J] = \mathsf{Nat}[I,J] \qquad !\mathit{ch}(\widetilde{T})/U = \mathit{ch}(\widetilde{T})/(!U) \qquad !\forall\widetilde{i}.\mathit{srv}^K(\widetilde{T})/U = \forall\widetilde{i}.\mathit{srv}^K(\widetilde{T})/(!U)$$

The (partial) operations on types defined above are extended pointwise to contexts. For example, for $\Gamma = v_1 : T_1, \ldots, v_n : T_n$ and $\Delta = v_1 : T'_1, \ldots, v_n : T'_n$, we define $\Gamma \mid \Delta = v_1 : T_1 \mid T'_1, \ldots, v_n : T_n \mid T'_n$. Note that this is defined just if Γ and Δ agree on the typing of integers and associate the same types (excluding usage) to names.

▶ **Definition 15.** *Given a capacity J_c and an interval $K = [K_1, K_2]$, we define $J_c; K$ by;*

$$J; [K_1, K_2] = [0, J + K_2] \qquad [\infty, \infty]; [K_1, K_2] = [0, 0] \qquad [I_{\mathbb{N}}, J]; [K_1, K_2] = [0, J + K_2]$$

Intuitively, $J_c; K$ represents the complexity of an input/output process when the input/output has capacity J_c and the complexity of the continuation is K. $J_c = [\infty, \infty]$ means the input/output will never succeed (because there is no corresponding output/input); hence the complexity is 0. A case where this is useful is given later in Example 22. Otherwise, an upper-bound is given by $J + K_2$ (the time spent for the input/output to succeed, plus K_2). The lower-bound is 0, since the input/output may be blocked forever.

The type system is given in Figures 5 and 6. The typing rules for expressions are standard ones for sized types.

A type judgment for processes is of the form $\varphi; \Phi; \Gamma \vdash P \triangleleft [I, J]$ where φ denotes the set of index variables, Φ is a set of constraints on index variables, and J is a bound on the parallel complexity of P under those constraints. This complexity bound J can also be seen as a bound on the open complexity of a process, that is to say the complexity of P in an environment corresponding to the types in Γ. For example, a channel with usage $\mathtt{In}_5^{[1,1]}$ alone cannot be reduced, as it is only used as an input. So, the typing $\cdot; \cdot; a : \mathtt{ch}()/\mathtt{In}_5^{[1,1]} \vdash \mathtt{tick}.a() \triangleleft [1, 6]$ says that in an environment that may provide an output on the channel a within the time interval $[1, 1] \oplus 5 = [0, 6]$, this process has a complexity bounded by 6. Similarly, the lower bound I is a lower bound on the parallel complexity of P. But in practice, this lower bound is often too imprecise.[1]

The (par) rule separates a context into two parts, and the complexity is the maximum over the two complexities, both for lower bound and upper bound. The (tick) rule shows the addition of a tick implies a delay of $[1, 1]$ in the context and the complexity. The (nu)

[1] This is because in the definition of $J_C; K$ in Definition 15, we pessimistically take into account the possibility that each input/output may be blocked forever. We can avoid the pessimistic estimation of the lower-bound by incorporating information about lock-freedom [23, 25].

$$\text{(zero)}\ \frac{}{\varphi;\Phi;\Gamma \vdash 0 \triangleleft [0,0]} \qquad \text{(par)}\ \frac{\varphi;\Phi;\Gamma \vdash P \triangleleft K_1 \qquad \varphi;\Phi;\Delta \vdash Q \triangleleft K_2}{\varphi;\Phi;\Gamma \mid \Delta \vdash P \mid Q \triangleleft K_1 \sqcup K_2}$$

$$\text{(tick)}\ \frac{\varphi;\Phi;\Gamma \vdash P \triangleleft K}{\varphi;\Phi;\uparrow^{[1,1]}\Gamma \vdash \mathtt{tick}.P \triangleleft K + [1,1]} \qquad \text{(ich)}\ \frac{\varphi;\Phi;\Gamma, a : \mathtt{ch}(\widetilde{T})/U, \widetilde{v} : \widetilde{T} \vdash P \triangleleft K}{\varphi;\Phi;\uparrow^{J_c}\Gamma, a : \mathtt{ch}(\widetilde{T})/\mathtt{In}_{J_c}^{[0,0]}.U \vdash a(\widetilde{v}).P \triangleleft J_c; K}$$

$$\text{(iserv)}\ \frac{(\varphi, \widetilde{i});\Phi;\Gamma, a : \forall \widetilde{i}.\mathtt{srv}^K(\widetilde{T})/U, \widetilde{v} : \widetilde{T} \vdash P \triangleleft K}{\varphi;\Phi;\uparrow^{J_c}!\Gamma, a : \forall \widetilde{i}.\mathtt{srv}^K(\widetilde{T})/!\mathtt{In}_{J_c}^{[0,0]}.U \vdash !a(\widetilde{v}).P \triangleleft [0,0]}$$

$$\text{(och)}\ \frac{\varphi;\Phi;\Gamma', a : \mathtt{ch}(\widetilde{T})/V \vdash \widetilde{e} : \widetilde{T} \qquad \varphi;\Phi;\Gamma, a : \mathtt{ch}(\widetilde{T})/U \vdash P \triangleleft K}{\varphi;\Phi;\uparrow^{J_c}(\Gamma \mid \Gamma'), a : \mathtt{ch}(\widetilde{T})/\mathtt{Out}_{J_c}^{[0,0]}.(V \mid U) \vdash \overline{a}\langle \widetilde{e} \rangle.P \triangleleft J_c; K}$$

$$\text{(oserv)}\ \frac{\varphi;\Phi;\Gamma', a : \forall \widetilde{i}.\mathtt{srv}^K(\widetilde{T})/V \vdash \widetilde{e} : \widetilde{T}\{\widetilde{I_{\mathbb{N}}}/\widetilde{i}\} \qquad \varphi;\Phi;\Gamma, a : \forall \widetilde{i}.\mathtt{srv}^K(\widetilde{T})/U \vdash P \triangleleft K'}{\varphi;\Phi;\uparrow^{J_c}(\Gamma \mid \Gamma'), a : \forall \widetilde{i}.\mathtt{srv}^K(\widetilde{T})/\mathtt{Out}_{J_c}^{[0,0]}.(V \mid U) \vdash \overline{a}\langle \widetilde{e} \rangle.P \triangleleft J_c; (K' \sqcup K\{\widetilde{I_{\mathbb{N}}}/\widetilde{i}\})}$$

$$\text{(if)}\ \frac{\varphi;\Phi;\Gamma \vdash e : \mathsf{Nat}[I,J] \qquad \varphi;\Phi, I \leq 0;\Gamma \vdash P \triangleleft K \qquad \varphi;\Phi, J \geq 1;\Gamma, x : \mathsf{Nat}[I\text{–}1, J\text{–}1] \vdash Q \triangleleft K}{\varphi;\Phi;\Gamma \vdash \mathtt{match}\ e\ \{\mathtt{case}\ 0 \mapsto P; \mathtt{case}\ \mathtt{s}(x) \mapsto Q\} \triangleleft K}$$

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

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

Figure 6 Typing Rules for Processes.

rule imposes that all names must have a reliable usage when they are created. In order to type a channel with the (ich) rule, the channel must have an input usage, with obligation $[0,0]$. Note that with the subusage relation, we have $\mathtt{In}_{J_c}^{A_o} \sqsubseteq \mathtt{In}_{J_c}^{[0,0]}$ if and only if $A_o = [0, I]$ for some I. So, this typing rule imposes that the lower-bound guarantee is correct, but the rule is not restrictive for upper-bound. This rule induces a delay of J_c in both context and complexity. Indeed, in practice this input does not happen immediately as we need to wait for output. This is where the assumption on when this output is ready, given by the capacity, is useful. The rule for output (och) is similar. For a server, the rule for input (iserv) is similar to (ich) in principle but differs in the way complexity is managed. Indeed, as a replicated input is never modified nor erased through a computation, giving it a non-zero complexity would harm the precision of the type system. Moreover, if this server represents for example a function on an integer with linear complexity, then the complexity of this server depends on the size of the integer it receives; that is why the complexity is transferred to the output rule on server, as one can see in the rule (oserv). Indeed, this rule (oserv) is again similar to (och) but the complexity of a call to the server is added in the rule. As we have polymorphism on servers, in order to type an output we need to find an instantiation on the indices $\widetilde{i}$, which is denoted by $\widetilde{I_{\mathbb{N}}}$ in this rule. Finally, the (if) rule is the only rule that modifies the set of constraints, and it gives information on the values the sizes can take. As explained in Example 21, those constraints are crucial in our sized type system. Note that contexts are not separated in this rule. So, for both branches, it means that the usage of channels must be the same. However, because we have the choice usage $(U + V)$, in practice we can use different usages in those two branches.

▶ **Example 16.** The typing derivation of the process in Example 2 is given in Figure 7. Note that the process $(\mathtt{tick}.a().\mathtt{tick}.\overline{a}\langle\rangle \mid \mathtt{tick}.a().\mathtt{tick}.\overline{a}\langle\rangle \mid \mathtt{tick}.a().\mathtt{tick}.\overline{a}\langle\rangle \mid \overline{a}\langle\rangle$ is also typable, in the same way, using the following usage (recall Example 10).

$$U := \mathtt{In}_2^{[1,1]}.\mathtt{Out}_0^{[1,1]} \mid \mathtt{In}_2^{[1,1]}.\mathtt{Out}_0^{[1,1]} \mid \mathtt{In}_2^{[1,1]}.\mathtt{Out}_0^{[1,1]} \mid \mathtt{Out}_{[1,1]}^{[0,0]}$$

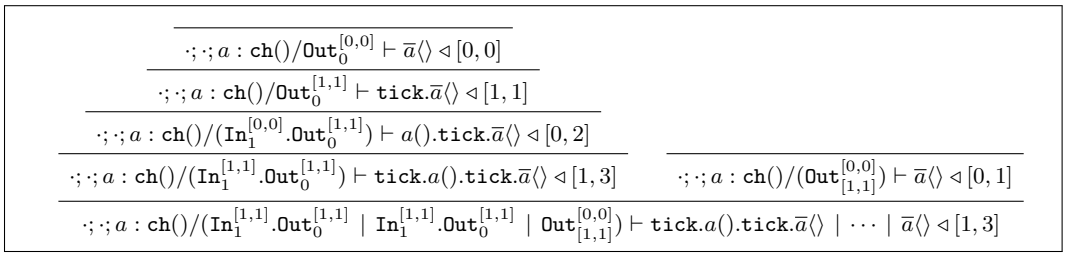

Figure 7 Typing of Example 2.

We thus obtain the complexity bound $[1,4]$.

An example for the use of servers and sizes is given later, in Example 21, as well as a justification for the use of intervals for obligations and capacities, in Example 20.

4 Soundness and Examples

The proof of soundness relies on subject reduction. In order to work on the parallel reduction relation $\Rightarrow$, we need to consider annotated processes. We introduce the following typing rule, for the annotation, as a generalization of the rule for `tick`.

$$\frac{\varphi;\Phi;\Gamma \vdash P \triangleleft K}{\varphi;\Phi;\uparrow^{[m,m]}\Gamma \vdash m : P \triangleleft K + [m,m]}$$

4.1 Subject Reduction and Soundness

In the Appendix B.2, we describe some intermediate lemmas needed for the soundness proof, namely weakening, strengthening and then substitution lemmas for index and expressions.

Let us first introduce a notation for subject reduction:

▶ **Definition 17** (Reduction for Contexts). *We say that a context Γ reduces to a context Γ' under $\varphi;\Phi$, denoted $\varphi;\Phi \vdash \Gamma \longrightarrow^* \Gamma'$ when one of the following holds:*

- $\Gamma = \Delta, a : \mathit{ch}(\widetilde{T})/U \qquad \varphi;\Phi \vdash U \longrightarrow^* U' \qquad \Gamma' = \Delta, a : \mathit{ch}(\widetilde{T})/U'$
- $\Gamma = \Delta, a : \forall \widetilde{i}.\mathit{srv}^K(\widetilde{T})/U \qquad \varphi;\Phi \vdash U \longrightarrow^* U' \qquad \Gamma' = \Delta, a : \forall \widetilde{i}.\mathit{srv}^K(\widetilde{T})/U'$

So, Γ' is Γ after some reduction steps but only in a unique usage. We obtain immediately that if all types in Γ are reliable then all types in Γ' are also reliable by definition of reliability.

The subject reduction property is stated as follows; see Appendix B.5 for a proof.

▶ **Theorem 18** (Subject Reduction). *If $\varphi;\Phi;\Gamma \vdash P \triangleleft K$ with all types in Γ reliable and $P \Rightarrow Q$ then there exists Γ' with $\varphi;\Phi \vdash \Gamma \longrightarrow^* \Gamma'$ and $\varphi;\Phi;\Gamma' \vdash Q \triangleleft K$.*

The following is the main soundness theorem.

▶ **Theorem 19.** *Let P be an annotated process and n be its global parallel complexity. Then, if $\varphi;\Phi;\Gamma \vdash P \triangleleft [I,J]$ with all types in Γ reliable, then we have $\varphi;\Phi \vDash J \geq n$. Moreover, if Γ does not contain any integers variables, we have $\varphi;\Phi \vDash I \leq n$.*

Proof. By Theorem 18, all reductions from P using $\Rightarrow$ conserve the typing. The context may be reduced too, but as a reduction step does not harm reliability, we can still apply the subject reduction through all the reduction steps of $\Rightarrow$. Moreover, for a process Q, if we have a typing $\varphi;\Phi;\Gamma \vdash Q \triangleleft [I,J]$, then $J \geq \mathcal{C}_\ell(Q)$. Thus, J is indeed a bound on the parallel

complexity by definition. As for the lower bound, one can see that we do not always have $I \leq \mathcal{C}_\ell(Q)$ because of the processes $\mathtt{tick}.Q'$ and $\mathtt{match}\ e\ \{\mathtt{case}\ \mathtt{0} \mapsto Q_1; \mathtt{case}\ \mathtt{s}(x) \mapsto Q_2\}$. However, those two processes are not in normal form for $\Rightarrow$, because $\mathtt{tick}.Q' \Rightarrow 1 : Q'$ and as there are no integer variables in Γ, the pattern matching can also be reduced. Thus, from a process Q we can find Q' such that $Q \Rightarrow Q'$ and Q' has no such processes of this shape on the top. And then, we obtain $I \leq \mathcal{C}_\ell(Q')$ which is smaller than the parallel complexity of Q by definition. ◀

4.2 Examples

Below we give several examples to demonstrate the expressive power of our type system.

▶ **Example 20** (Intervals)**.** To see the need for an interval capacity, consider the following process:

$$a().\overline{b}\langle\rangle \mid \mathtt{match}\ e\ \{\mathtt{case}\ \mathtt{0} \mapsto \overline{a}\langle\rangle; \mathtt{case}\ \mathtt{s}(x) \mapsto \mathtt{tick}.\overline{a}\langle\rangle\}$$

Depending on the value of e (which may be statically unknown), an output on a may be available at time 0 or 1. Thus, the input usage on a should have a capacity interval $[0, 1]$. As a result, the obligation of the output usage on b should also be an interval $[0, 1]$.

Now, one may think that we can assume that lower-bounds are always 0 and omit lower-bounds, since we are mainly interested in an *upper-bound* of the parallel complexity. Information about lower-bounds is, however, actually required for precise reasoning on upper-bounds. For example, consider the process

$$a().\overline{b}\langle\rangle \mid \mathtt{tick}.\overline{a}\langle\rangle.b()$$

With intervals, a have the usage $\mathtt{In}_{[1,1]}^{[0,0]} \mid \mathtt{Out}_0^{[1,1]}$ and so b has the usage $\mathtt{Out}_{[0,0]}^{[1,1]} \mid \mathtt{In}_{[0,0]}^{[1,1]}$, and the parallel complexity of the process can be precisely inferred to be 1.

If we set lower-bounds to 0 and assign to a the usage $\mathtt{In}_{[0,1]}^{[0,0]} \mid \mathtt{Out}_0^{[0,1]}$, then the usage of b can only be: $\mathtt{Out}_1^{[0,1]} \mid \mathtt{In}_1^{[0,1]}$. Note that according to the imprecise usage of a, the output on b may become ready at time 0 and then have to wait for one time unit until the input on b becomes ready; thus, the capacity of the output on b is 1, instead of $[0, 0]$. An upper-bound of the parallel complexity would therefore be inferred to be $1 + 1 = 2$ (because the usages tell us that the lefthand side process may wait for one time unit at a, and then for another time unit at b), which is too imprecise.

We remark that this problem does not come for an inappropriate definitions of usages with only upper-bound in our work. Indeed, by adapting the usage type system given in [23], we would have the same imprecision. In the same way, trying to give a notion of reliability that makes the usage $\mathtt{Out}_0^{[0,1]} \mid \mathtt{In}_0^{[0,1]}$ reliable would lead to an unsound type system, as it would make the subusage relation less flexible, which is essential for soundness. ⌟

Let us also present how sizes and polymorphism over indices in servers can type processes defined by replication such as the factorial. Please note that by taking inspiration from the typing in [4], using the type representation given in the Appendix A, more complicated examples of parallel programs such as the bitonic sort could be typed in our setting with a good complexity bound.

▶ **Example 21** (Factorial)**.** Assume a function on expressions $\mathtt{mult} : \mathsf{Nat}[I, J] \times \mathsf{Nat}[I', J'] \to \mathsf{Nat}[I * I', J * J']$. In practice, this should be encoded as a server in π-calculus, but for simplicity, we consider it as a function. We will describe the factorial and count the number

$$P := !f(n,r).\texttt{match}\ n\ \{\texttt{case}\ 0 \mapsto \overline{r}\langle 1\rangle; \texttt{case}\ \texttt{s}(m) \mapsto (\nu r')(\overline{f}\langle m, r'\rangle \mid r'(x).\texttt{tick}.\overline{r}\langle mult(n,x)\rangle)\}$$

$$\dfrac{\dfrac{}{i;\cdot;n:\mathsf{Nat}[i] \vdash n:\mathsf{Nat}[i]} \quad \dfrac{\dfrac{}{i;(i\le 0)\vDash i! = 1}}{i;i\le 0;f:T',n:\mathsf{Nat}[i],r:\mathtt{ch}(\mathsf{Nat}[i!])/\mathtt{Out}_0^{[i,i]} \vdash \overline{r}\langle 1\rangle \lhd [0,i]}\ \pi_1}{\dfrac{i;\cdot;f:T',n:\mathsf{Nat}[i],r:\mathtt{ch}(\mathsf{Nat}[i!])/\mathtt{Out}_0^{[i,i]} \vdash \texttt{match}\ n\ \{\texttt{case}\ 0 \mapsto \overline{r}\langle 1\rangle; \texttt{case}\ \texttt{s}(m) \mapsto \cdots\} \lhd [0,i]}{\cdot;\cdot;f:T \vdash !f(n,r).\texttt{match}\ n\ \{\texttt{case}\ 0 \mapsto \overline{r}\langle 1\rangle; \texttt{case}\ \texttt{s}(m) \mapsto \cdots\} \lhd [0,0]}}$$

with the main branch of π_1 being:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{(i;i\ge 1)\vDash i*(i-1)! = i!}}{i;i\ge 1;n:\mathsf{Nat}[i],x:\mathsf{Nat}[(i-1)!] \vdash mult(n,x):\mathsf{Nat}[i!]}}{i;i\ge 1;n:\mathsf{Nat}[i],x:\mathsf{Nat}[(i-1)!],r:\mathtt{ch}(\mathsf{Nat}[i!])/\mathtt{Out}_0^{[0,0]} \vdash \overline{r}\langle mult(n,x)\rangle \lhd [0,0]}}{i;i\ge 1;n:\mathsf{Nat}[i],x:\mathsf{Nat}[(i-1)!],r:\mathtt{ch}(\mathsf{Nat}[i!])/\mathtt{Out}_0^{[1,1]} \vdash \texttt{tick}.\overline{r}\langle mult(n,x)\rangle \lhd [1,1]}}{\cdots \qquad i;i\ge 1;n:\mathsf{Nat}[i],r':S_2,r:\mathtt{ch}(\mathsf{Nat}[i!])/\mathtt{Out}_0^{[i,i]} \vdash r'(x).\cdots \lhd [0,i]}}{i;i\ge 1;n:\mathsf{Nat}[i],m:\mathsf{Nat}[i-1],r:\mathtt{ch}(\mathsf{Nat}[i!])/\mathtt{Out}_0^{[i,i]},f:T',r':S \vdash \overline{f}\langle m,r'\rangle \mid r'(x).\cdots \lhd [0,i]}}{i;i\ge 1;n:\mathsf{Nat}[i],m:\mathsf{Nat}[i-1],r:\mathtt{ch}(\mathsf{Nat}[i!])/\mathtt{Out}_0^{[i,i]},f:T' \vdash (\nu r')\cdots \lhd [0,i]}$$

Figure 8 Representation and Typing of Factorial.

of multiplications with `tick`. We write $\mathsf{Nat}[I]$ to denote $\mathsf{Nat}[I,I]$. We use the usual notation $I!$ to represent the factorial function in indices. The process representing factorial and its typing derivation are given in Figure 8. The following type T denotes:

$$\forall i.\mathtt{srv}^{[0,i]}(\mathsf{Nat}[i],\mathtt{ch}(\mathsf{Nat}[i!])/\mathtt{Out}_0^{[i,i]})/(!\mathtt{In}_\infty^{[0,0]}.\mathtt{Out}_0^{[0,\infty]})$$

Denoting a server taking an integer as input, and a return channel on which the factorial of this integer is sent, in i units of time. The usage of this server describes that it can be called anytime. This type is reliable and it would be reliable even if composed with any kind of output $\mathtt{Out}_0^{A_o}$ if we want to call this server. Let:

$$T' = \forall i.\mathtt{srv}^{[0,i]}(\mathsf{Nat}[i],\mathtt{ch}(\mathsf{Nat}[i!])/\mathtt{Out}_0^{[i,i]})/\mathtt{Out}_0^{[0,\infty]})$$

$$S = \mathtt{ch}(\mathsf{Nat}[(i-1)!])/(\mathtt{Out}_0^{[i-1,i-1]} \mid \mathtt{In}_{[i-1,i-1]}^{[0,0]}) = S_1 \mid S_2$$

where S_1 and S_2 are obtained by the expected separation of the usage. This type S is reliable under $(i);(i\ge 1)$. Thus, we give the typing described in Figure 8. From the type of f, we see on its complexity $[0,i]$ that it does at most a linear number of multiplications. Note that the constraints that appear in a match are useful since without them, we could not prove $i;(i\le 0)\vDash i! = 1$ and $i;(i\ge 1)\vDash i*(i-1)! = i!$. Moreover, polymorphism over indices is necessary in order to find that the recursive call is made on a strictly smaller size $i-1$. ⌟

Let us now justify the use of this operator $J_c;K$ in order to treat complexity.

▶ **Example 22** (Deadlock). Let us consider the process $P := (\nu a)(\nu b)(a().\mathtt{tick}.\overline{b}\langle\rangle \mid b().\mathtt{tick}.\overline{a}\langle\rangle)$. P is typed as shown in Figure 9. As a and b have exactly the same behaviour, let us focus on the typing of $a().\mathtt{tick}.\overline{b}\langle\rangle$. The derivation for the subprocess $\mathtt{tick}.\overline{b}\langle\rangle$ should be clear. By assigning the usage to $\mathtt{In}_{[\infty,\infty]}^{[0,0]}$, the cost for $a().\mathtt{tick}.\overline{b}\langle\rangle$ is calculated by: $[\infty,\infty];K = [0,0]$. Thus, we can correctly infer that the complexity of the deadlocked process is 0. ⌟

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\cdot;\cdot;a:\mathtt{ch}()/0, b:\mathtt{ch}()/\mathtt{Out}_0^{[0,0]} \vdash \overline{b}\langle\rangle \triangleleft [0,0]}}{\cdot;\cdot;a:\mathtt{ch}()/0, b:\mathtt{ch}()/\mathtt{Out}_0^{[1,1]} \vdash \mathtt{tick}.\overline{b}\langle\rangle \triangleleft [1,1]}}{\cdot;\cdot;a:\mathtt{ch}()/\mathtt{In}_{[\infty,\infty]}^{[0,0]}, b:\mathtt{ch}()/\mathtt{Out}_0^{[\infty,\infty]} \vdash a().\mathtt{tick}.\overline{b}\langle\rangle \triangleleft [0,0]}\quad \text{symmetry for the other branch}}{\cdot;\cdot;a:\mathtt{ch}()/(\mathtt{In}_{[\infty,\infty]}^{[0,0]} \mid \mathtt{Out}_0^{[\infty,\infty]}), b:\mathtt{ch}()/(\mathtt{Out}_0^{[\infty,\infty]} \mid \mathtt{In}_{[0,0]}^{[0,0]}) \vdash a().\mathtt{tick}.\overline{b}\langle\rangle \mid b().\mathtt{tick}.\overline{a}\langle\rangle \triangleleft [0,0]}}{\cdot;\cdot;\cdot \vdash (\nu a)(\nu b)(a().\mathtt{tick}.\overline{b}\langle\rangle \mid b().\mathtt{tick}.\overline{a}\langle\rangle) \triangleleft [0,0]}$$

Figure 9 Typing of Example 22.

▶ **Example 23.** We describe informally an example for which our system can give a complexity, but fails to catch a precise bound. Let us consider the process:

$$P := \mathtt{tick}.!a(n).\mathtt{match}\ n\ \{\mathtt{case}\ \mathtt{0} \mapsto 0; \mathtt{case}\ \mathtt{s}(m) \mapsto \overline{a}\langle m\rangle\} \mid \overline{a}\langle 10\rangle \mid \mathtt{tick}.\mathtt{tick}.!a(n).0$$

This process has complexity 2. However, if we want to give a usage to the server a, we must have a usage:

$$!\mathtt{In}_0^{[1,1]}.\mathtt{Out}_1^{[0,0]} \mid \mathtt{Out}_{[1,2]}^{[0,0]} \mid !\mathtt{In}_0^{[2,2]}$$

We took as obligations the number of ticks before the action, and as capacity the minimal number for which we have reliability. So in particular, because of the capacity 1 in the usage $\mathtt{Out}_1^{[0,0]}$, typing the recursive call $\overline{a}\langle m\rangle$ increases the complexity by one, and so typing n recursive calls generates a complexity of n in the type system. So, in our setting, the complexity of this process can only be bounded by 10. Overall, this type system may not behave well when there are more than one replicated input process on each server channel, since an imprecision on a capacity for a recursive call leads to an overall imprecision depending on the number of recursive calls. This issue is the only source of imprecision we found with respect to the type system of [4]: see the conjecture in Section 5. ⌟

5 Related Work

Some contributions to the complexity analysis of parallel functional programs by types appear in [16, 20] but the languages studied do not express concurrency. Alternatively [1, 2, 15] address the problem of analysing the time complexity of distributed or concurrent systems. They provide interesting analyses on some instances of systems but do not handle dynamic creation of processes and channel name passing as in the π-calculus. Moreover, the flow graph or rely-guarantee reasoning techniques employed in [1, 2] do not seem to offer the same compositionality as type systems.

There have recently been several studies on type-based cost analysis for binary or multiparty session calculi [5, 9, 10]. It is not clear whether and how those methods can be extended to deal with more general concurrent processes that can be written in the π-calculus, where there may be more than one sender/receiver process for each channel. Among those studies, Das et al.'s work [9] seems technically closest to ours. Both their cost models and ours are parametric, as they rely on a similar `tick` operation. Moreover, their temporal operators seem to have a strong correspondence with usages and operations on them. More specifically, the next operator $\bigcirc$ [9] is similar to the usage operator $\uparrow^{[1,1]}$ in our type system, and $\Box$ and $\Diamond$ roughly correspond to input and output usages with capacity and obligations $[0, \infty]$. For more precise comparison, we need to extend our type system with variant and recursive types, to encode their session calculus into the π-calculus, following [8, 24]. It is left for future work.

Kobayashi et al. [24–26] also used the notion of usages to reason about deadlocks, livelocks, and information flow, but he used a single number for each obligation and capacity (the latter is called a "capability" in his work). In particular, a usage type system for time boundedness, related with parallel complexity, was given in [23]. However, the definition of parallel complexity and thus the definition of usages and reliability in this work is quite different from ours, as its reduction does not take into account some non-deterministic paths. Moreover, as explained in Example 20, the use of a single number and not intervals induces a loss of precision even on simple examples; we have generalized the number to an interval to improve the precision of our analysis. More recently, the first two authors proposed in [4] a type system with the same goal of analysing parallel complexity in π-calculus. This type system builds on sized types and input/output types instead of usages. Because of that, they cannot manage successive uses of the same channel as in Example 2, as their names can essentially be used at only one specific time. In most cases, the time annotation used for channels in their setting corresponds to the sum of the lower bound for obligation and the upper bound for capacity in our setting. We conjecture the following result:

▶ **Conjecture 24** (Comparison with [4]). *Suppose given a typing $\varphi; \Phi; \Gamma_{i/o} \vdash_{i/o} P \triangleleft J$ in the input/output sized type system of [4], such that this process P has a linear use of channels. Then, there exists a reliable context Γ such that $\varphi; \Phi; \Gamma \vdash P \triangleleft [0, J]$.*

More details and some intuitions are given in the Appendix A. So, on a simple use of names our system is strictly more precise if this conjecture is true. However, on other cases, like in Example 23, their system is more precise as the loss of precision because of usage does not happen in their setting. On the contrary, our setting has fairly more precision for processes with a non-trivial use of channels, as in Example 2.

There have also been studies on implicit computational complexity for process calculi, albeit for less expressive calculi than the pi-calculus [7, 14, 28]. Unlike our work, they consider the work rather than the span, and characterize complexity classes, rather than estimating the precise execution time of a given process. The paper [12] by contrast considers the π-calculus and causal (parallel) complexity, but the goal here is also to delineate a characterization of polynomial complexity.

6 Conclusion

We presented a type system built on sized types and usages such that a type derivation for a process gives an upper bound on the parallel complexity of this process. The type system relies on intervals in order to give an approximation of the sizes of integers in the process, and an approximation of the time an input or an output needs to synchronize. In comparison to [4], we showed with examples that our type system can type some concurrent behaviour that was not captured in their type system, and on a certain subset of processes, we conjecture that our new type system is strictly more precise.

Building on previous work by the third author on type inference for usages [25, 27], we plan to investigate type inference, with the use of constraint solving procedures for indices.

References

1 Elvira Albert, Jesús Correas, Einar Broch Johnsen, and Guillermo Román-Díez. Parallel cost analysis of distributed systems. In *Static Analysis - 22nd International Symposium, SAS 2015, Saint-Malo, France, September 9-11, 2015, Proceedings*, volume 9291 of *Lecture Notes in Computer Science*, pages 275–292. Springer, 2015.

2 Elvira Albert, Antonio Flores-Montoya, Samir Genaim, and Enrique Martin-Martin. Rely-guarantee termination and cost analyses of loops with concurrent interleavings. *Journal of Automated Reasoning*, 59(1):47–85, 2017.

3 Martin Avanzini and Ugo Dal Lago. Automating sized-type inference for complexity analysis. *Proceedings of the ACM on Programming Languages*, 1(ICFP):43, 2017.

4 Patrick Baillot and Alexis Ghyselen. Types for complexity of parallel computation in pi-calculus. In *Programming Languages and Systems - 30th European Symposium on Programming, ESOP 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021*, volume 12648 of *Lecture Notes in Computer Science*, pages 59–86. Springer, 2021.

5 David Castro-Perez and Nobuko Yoshida. CAMP: cost-aware multiparty session protocols. *Proc. ACM Program. Lang.*, 4(OOPSLA):155:1–155:30, 2020.

6 Ugo Dal Lago and Marco Gaboardi. Linear dependent types and relative completeness. In *Logic in Computer Science (LICS), 2011 26th Annual IEEE Symposium on*, pages 133–142. IEEE, 2011.

7 Ugo Dal Lago, Simone Martini, and Davide Sangiorgi. Light logics and higher-order processes. *Mathematical Structures in Computer Science*, 26(6):969–992, 2016.

8 Ornela Dardha, Elena Giachino, and Davide Sangiorgi. Session types revisited. *Information and Computation*, 256:253–286, 2017.

9 Ankush Das, Jan Hoffmann, and Frank Pfenning. Parallel complexity analysis with temporal session types. *Proc. ACM Program. Lang.*, 2(ICFP):91:1–91:30, 2018.

10 Ankush Das, Jan Hoffmann, and Frank Pfenning. Work analysis with resource-aware session types. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, pages 305–314. ACM, 2018.

11 Romain Demangeon, Daniel Hirschkoff, Naoki Kobayashi, and Davide Sangiorgi. On the complexity of termination inference for processes. In Gilles Barthe and Cédric Fournet, editors, *Trustworthy Global Computing, Third Symposium, TGC 2007, Sophia-Antipolis, France, November 5-6, 2007, Revised Selected Papers*, volume 4912 of *Lecture Notes in Computer Science*, pages 140–155. Springer, 2007. doi:10.1007/978-3-540-78663-4_11.

12 Romain Demangeon and Nobuko Yoshida. Causal computational complexity of distributed processes. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '18, pages 344–353. ACM, 2018.

13 Yuxin Deng and Davide Sangiorgi. Ensuring termination by typability. *Information and Computation*, 204(7):1045–1082, 2006.

14 Paolo Di Giamberardino and Ugo Dal Lago. On session types and polynomial time. *Mathematical Structures in Computer Science*, -1, 2015.

15 Elena Giachino, Einar Broch Johnsen, Cosimo Laneve, and Ka I Pun. Time complexity of concurrent programs - - A technique based on behavioural types -. In *Formal Aspects of Component Software - 12th International Conference, FACS 2015, Niterói, Brazil, October 14-16, 2015, Revised Selected Papers*, volume 9539 of *Lecture Notes in Computer Science*, pages 199–216. Springer, 2016.

16 Stéphane Gimenez and Georg Moser. The complexity of interaction. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 243–255, 2016.

17 Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. Multivariate amortized resource analysis. *ACM Trans. Program. Lang. Syst.*, 34(3):14:1–14:62, 2012.

18 Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. Resource aware ML. In *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, volume 7358 of *Lecture Notes in Computer Science*, pages 781–786. Springer, 2012.

19 Jan Hoffmann and Martin Hofmann. Amortized resource analysis with polynomial potential. In *Programming Languages and Systems, 19th European Symposium on Programming, ESOP 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, volume 6012 of *Lecture Notes in Computer Science*, pages 287–306. Springer, 2010.

20 Jan Hoffmann and Zhong Shao. Automatic static cost analysis for parallel programs. In Jan Vitek, editor, *Programming Languages and Systems*, pages 132–157, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

21 Martin Hofmann and Steffen Jost. Static prediction of heap space usage for first-order functional programs. In *Conference Record of POPL 2003: The 30th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, New Orleans, Louisisana, USA, January 15-17, 2003*, pages 185–197. ACM, 2003.

22 Naoki Kobayashi. A partially deadlock-free typed process calculus. In *Proceedings, 12th Annual IEEE Symposium on Logic in Computer Science, Warsaw, Poland, June 29 - July 2, 1997*, pages 128–139. IEEE Computer Society, 1997. doi:10.1109/LICS.1997.614941.

23 Naoki Kobayashi. A type system for lock-free processes. *Information and Computation*, 177(2):122–159, 2002.

24 Naoki Kobayashi. Type systems for concurrent programs. In *Formal Methods at the Crossroads. From Panacea to Foundational Support*, pages 439–453. Springer, 2003.

25 Naoki Kobayashi. Type-based information flow analysis for the π-calculus. *Acta Informatica*, 42(4-5):291–347, 2005.

26 Naoki Kobayashi. A new type system for deadlock-free processes. In *International Conference on Concurrency Theory*, pages 233–247. Springer, 2006.

27 Naoki Kobayashi, Shin Saito, and Eijiro Sumii. An implicitly-typed deadlock-free process calculus. In Catuscia Palamidessi, editor, *CONCUR 2000 — Concurrency Theory*, pages 489–504. Springer Berlin Heidelberg, 2000.

28 Antoine Madet and Roberto M. Amadio. An elementary affine λ-calculus with multithreading and side effects. In *Typed Lambda Calculi and Applications - 10th International Conference, TLCA 2011, Novi Sad, Serbia, June 1-3, 2011. Proceedings*, volume 6690 of *Lecture Notes in Computer Science*, pages 138–152. Springer, 2011.

29 Robin Milner. *Communication and concurrency*. PHI Series in computer science. Prentice Hall, 1989.

30 Eijiro Sumii and Naoki Kobayashi. A generalized deadlock-free process calculus. In *Proc. of Workshop on High-Level Concurrent Language (HLCL'98)*, volume 16(3) of *ENTCS*, pages 55–77, 1998.

A Comparison with [4]

In this section, we give intuitively a description of how to simulate types on [4] in a linear setting with usage. We say that a process has a linear use of channels if it use channel names at most one time for input and at most one time for output. For servers, we suppose that the replicated input is once and for all defined at the beginning of a process, and as free variables it can only use others servers. In their type system, a channel is given a type $\mathtt{Ch}_I(\widetilde{T})$ where I is an upper bound on the time this channel communicates. It can also be a variant of this type with only input or only output capability. Such a channel would be represented in out type system by a type $\mathtt{ch}(\widetilde{T})/(\mathtt{In}_{J_c^1}^{[I_1,I_1]} \mid \mathtt{Out}_{J_c^2}^{I_2,I_2})$ where either J_c^1 is 0 and then $I_1 \leq I$, either $J_c^1 = [J_1, J_1]$ and then $I_1 + J_1 \leq I$. We have the same thing for J_c^2 and I_2. To be more precise, the typing in our setting should be a non-deterministic choice (using $+$) over such usages, and the capacity should adapt to the obligation of the dual action in order to be reliable. So, for example if $I_1 \leq I_2$, then we would take: $\mathtt{ch}(\widetilde{T})/(\mathtt{In}_{[I_2-I_1,I_2-I_1]}^{[I_1,I_1]} \mid \mathtt{Out}_0^{I_2,I_2})$. Note that this shape of type adapts well to the way time is delayed in their setting. For example, the `tick` constructor in their setting make the time advance by 1, and in our setting, then we would obtain the usage $(\mathtt{In}_{[I_2-I_1,I_2-I_1]}^{[I_1+1,I_1+1]} \mid \mathtt{Out}_0^{I_2+1,I_2+1})$ and we still have $I_2 - I_1 = (I_2 + 1) - (I_1 + 1)$.

In the same way, in their setting when doing an output (or input), the time is delayed by I. Here, with usages, it would be delayed by J_c which is, by definition, a delay of the shape $\uparrow^{[J,J]}$ with $J \leq I$. So, we would keep the invariant that our time annotation have the shape of singleton interval with a smaller value than the time annotation in their setting.

For servers, in the linear setting, their types have the shape: ${}_I\forall\widetilde{i}.\mathtt{srv}^J(\widetilde{T})$ where $I = 0$ is again a time annotation giving an upper bound on the time the input action of this server is defined, and J is a complexity as in our setting. So, in our setting it would be:

$$\forall\widetilde{i}.\mathtt{srv}^{[0,J]}(\widetilde{\underline{T}})/!\mathtt{In}_{\infty}^{[0,0]}.!\mathtt{Out}_{0}^{[0,\infty]} \mid !\mathtt{Out}_{0}^{[0,\infty]}$$

Note that this usage is reliable. The main point here is this infinite capacity for input. Please note that because of our input rule for servers, it does not generates an infinite complexity. However, it imposes a delaying $\uparrow^{[0,\infty]}!\Gamma$ in the context. Because of the shape we gave to types, it means that the context can only have outputs for other servers as free variables, but this was the condition imposed by linearity. Note that in [4], they have a restriction on the free variables of servers that is in fact the same restriction so it does not harm the comparison to take this restriction on free variables. As an example, the bitonic sort described in [4] could be typed similarly in our setting with this kind of type.

Finally, choice in usages $U_1 + U_2$ is used to put together the different usages we obtain in the two branches of a pattern matching.

B Proofs

In this section, we prove Theorem 18, after giving various lemmas.

B.1 Properties of Subusage

The subusage relation satisfies some properties that are essential for the soundness theorem. First, we have the usual properties of subusage:

▶ **Lemma 25.** *If* $\varphi; \Phi \vDash B_o \subseteq A_o$ *then* $\varphi; \Phi \vdash (\uparrow^{A_o} U) \sqsubseteq (\uparrow^{B_o} U)$

▶ **Lemma 26** (Properties of Subusage)**.** *For a set of index variables* φ *and a set of constraints* Φ *on* φ *we have:*

1. *If* $\varphi; \Phi \vdash U \sqsubseteq V$ *then for any interval* A_o*, we have* $\varphi; \Phi \vdash \uparrow^{A_o} U \sqsubseteq \uparrow^{A_o} V$*.*
2. *If* $\varphi; \Phi \vdash U \sqsubseteq V$ *and* $\varphi; \Phi \vdash V \longrightarrow V'$*, then there exists* U' *such that* $\varphi; \Phi \vdash U \longrightarrow^* U'$ *and* $\varphi; \Phi \vdash U' \sqsubseteq V'$ *(with* $\mathtt{err} \sqsubseteq U$ *for any usage* U*)*
3. *If* $\varphi; \Phi \vdash U \sqsubseteq V$ *and* U *is reliable under* $\varphi; \Phi$ *then* V *is reliable under* $\varphi; \Phi$*.*

The proof of the first lemma is rather easy, but for the second lemma it is a bit cumbersome. Intuitively all the base rules for subusage satisfy this and the main difficulty is to show that the congruence rule and context rule conserve those properties.

More specific to this type system, we also have the following lemma, stating that delaying does not modify the behaviour of a type.

▶ **Lemma 27** (Invariance by Delaying). *For any interval A_o:*
1. *If $\varphi;\Phi \vdash \uparrow^{A_o} U \longrightarrow V'$ then, there exists V such that $\uparrow^{A_o} V = V'$ and $\varphi;\Phi \vdash U \longrightarrow V$. (with $\texttt{err} = \uparrow^{A_o}\texttt{err}$)*
2. *If $\varphi;\Phi \vdash U \longrightarrow V$ then $\varphi;\Phi \vdash \uparrow^{A_o} U \longrightarrow \uparrow^{A_o} V$.*
3. *U is reliable under $\varphi;\Phi$ if and only if $\uparrow^{A_o} U$ is reliable under $\varphi;\Phi$.*

In our setting, this lemma shows among other things that the `tick` constructor, or more generally the annotation $n : P$, does not break reliability. Those properties can easily be proved with simple inductions.

B.2 Intermediate Lemmas

We give some intermediate lemmas for the soundness theorem

▶ **Lemma 28** (Weakening). *Let φ, φ' be disjoint set 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 $\varphi;\Phi;\Gamma \vdash e : T$ then $(\varphi, \varphi');(\Phi, \Phi');\Gamma, \Gamma' \vdash e : T$.*
2. *If $\varphi;\Phi;\Gamma \vdash P \triangleleft K$ then $(\varphi, \varphi');(\Phi, \Phi');\Gamma, \Gamma' \vdash P \triangleleft K$.*

We also show that we can remove some useless hypothesis.

▶ **Lemma 29** (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. *If $\varphi;(\Phi, C);\Gamma, \Gamma' \vdash e : T$ and the variables in Γ' are not free in e, then $\varphi;\Phi;\Gamma \vdash e : T$.*
2. *If $\varphi;(\Phi, C);\Gamma, \Gamma' \vdash P \triangleleft K$ and the variables in Γ' are not free in P, then $\varphi;\Phi;\Gamma \vdash P \triangleleft K$.*

Those two lemmas are proved easily by successive induction on the definitions in this paper. Then, we also have a lemma expressing that index variables can indeed be replaced by any index.

▶ **Lemma 30** (Index Substitution). *Let φ be a set of index variables and $i \notin \varphi$. Let $J_{\mathbb{N}}$ be an index with free variables in φ. Then,*
1. *If $(\varphi, i);\Phi;\Gamma \vdash e : T$ then $\varphi;\Phi\{J_{\mathbb{N}}/i\};\Gamma\{J_{\mathbb{N}}/i\} \vdash e : T\{J_{\mathbb{N}}/i\}$.*
2. *If $(\varphi, i);\Phi;\Gamma \vdash P \triangleleft K$ then $\varphi;\Phi\{J_{\mathbb{N}}/i\};\Gamma\{J_{\mathbb{N}}/i\} \vdash P \triangleleft K\{J_{\mathbb{N}}/i\}$.*

B.3 Substitution Lemma

We now present the variable substitution lemmas. In the setting of usages, this lemma is a bit more complex than usual. Indeed, we have a separation of contexts with the parallel composition, and we have to rely on subusage, especially the rule $\varphi;\Phi \vdash (\alpha_J^{A_o}.U) \mid (\uparrow^{A_o+J_c} V) \sqsubseteq \alpha_{J_c}^{A_o}.(U \mid V)$ as expressed in the Example 11 above. We put some emphasis on the following notation: when we write $\Gamma, v : T$ as a context in typing, it means that v does not appear in Γ.

▶ **Lemma 31** (Substitution). *Let Γ and Δ be contexts such that $\Gamma \mid \Delta$ is defined. Then we have:*
1. *If $\varphi;\Phi;\Gamma, v : T \vdash e' : T'$ and $\Delta \vdash e : T$ then $\varphi;\Phi;\Gamma \mid \Delta \vdash e'[v := e] : T'$*
2. *If $\varphi;\Phi;\Gamma, v : T \vdash P \triangleleft K$ and $\Delta \vdash e : T$ then $\varphi;\Phi;\Gamma \mid \Delta \vdash P[v := e] \triangleleft K$*

The first point is straightforward. It uses the fact that we have the relation $\varphi;\Phi \vdash U \sqsubseteq 0$ for any usage U, and so we can use $\varphi;\Phi \vdash \Gamma \mid \Delta \sqsubseteq \Gamma$ in order to weaken Δ (similarly for Γ) if needed. The second point is more interesting. The easy case is when T is $\mathsf{Nat}[I, J]$ for some $[I, J]$. Then, we take a Δ that only uses the zero usage, and so $\Gamma \mid \Delta = \Gamma$ and everything becomes simpler. The more interesting cases are:

▶ **Lemma 32** (Difficult Cases of Substitution). *We have:*

- *If* $\varphi;\Phi;\Gamma, b{:}\,\mathtt{ch}(\widetilde{S})/W_0, c{:}\,\mathtt{ch}(\widetilde{S})/W_1 \vdash P \triangleleft K$ *then* $\varphi;\Phi;\Gamma, b{:}\,\mathtt{ch}(\widetilde{S})/(W_0 \,|\, W_1) \vdash P[c := b] \triangleleft K$
- *If* $\varphi;\Phi;\Gamma, b : \forall\widetilde{i}.\mathtt{srv}^K(\widetilde{S})/W_0, c : \forall\widetilde{i}.\mathtt{srv}^K(\widetilde{S})/W_1 \vdash P \triangleleft K$ *then* $\varphi;\Phi;\Gamma, b : \forall\widetilde{i}.\mathtt{srv}^K(\widetilde{S})/(W_0 \mid W_1) \vdash P[c := b] \triangleleft K$

With a careful induction, we can indeed prove this lemma, relying on the subusage relation.

B.4 Congruence Equivalence

In order to prove the soundness theorem, we need first a lemma saying that the congruence relation behaves well with typing.

▶ **Lemma 33** (Congruence and Typing). *Let P and Q be annotated processes such that $P \equiv Q$. Then, $\varphi;\Phi;\Gamma \vdash P \triangleleft K$ if and only if $\varphi;\Phi;\Gamma \vdash Q \triangleleft K$.*

The proof is an induction on $P \equiv Q$, relying on all the previous lemma, and especially the lemmas on delaying for congruence rules specific to annotated processes.

B.5 Proof of Theorem 18 (Subject Reduction)

We now detail some cases for the proof of Theorem 18.

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, notice that in those kinds of proof, the top-level rule of subtyping can be ignored. Indeed, we can always simulate exactly the same subtyping rule for both P and Q We now proceed by doing the case analysis on the rules of Figure 1. In order to simplify the proof, we will also consider that types and indexes invariant by subtyping (like the complexity in a server) are not renamed with subtyping. Note that this only add cumbersome notations but it does not change the core of the proof. We detail only the case for synchronization:

Case $(n : a(\widetilde{v}).P) \mid (m : \overline{a}\langle\widetilde{e}\rangle.Q) \Rightarrow (\max(m,n) : (P[\widetilde{v} := \widetilde{e}] \mid Q))$. Consider the typing $\varphi;\Phi;\Gamma_0 \mid \Delta_0, a : \mathtt{ch}(\widetilde{T})/(U_0 \mid V_0) \vdash (n : a(\widetilde{v}).P) \mid (m : \overline{a}\langle\widetilde{e}\rangle.Q) \triangleleft K_0 \sqcup K'_0$. The first rule is the rule for parallel composition, then the proof is split into the two following subtree:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\pi_P}{\varphi;\Phi;\Gamma_2, a : \mathtt{ch}(\widetilde{T})/U_2, \widetilde{v} : \widetilde{T} \vdash P \triangleleft K_2}}{\varphi;\Phi;\uparrow^{J_c}\Gamma_2, a : \mathtt{ch}(\widetilde{T})/\mathtt{In}^{[0,0]}_{J_c}.U_2 \vdash a(\widetilde{v}).P \triangleleft J_c; K_2}\ (3)}}{\varphi;\Phi;\Gamma_1, a : \mathtt{ch}(\widetilde{T})/U_1 \vdash a(\widetilde{v}).P \triangleleft K_1}}{\varphi;\Phi;\uparrow^{[n,n]}\Gamma_1, a : \mathtt{ch}(\widetilde{T})/\uparrow^{[n,n]}U_1 \vdash n : a(\widetilde{v}).P \triangleleft K_1 + [n,n]}\ (4)}}{\varphi;\Phi;\Gamma_0, a : \mathtt{ch}(\widetilde{T})/U_0 \vdash n : a(\widetilde{v}).P \triangleleft K_0}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\pi_e}{\varphi;\Phi;\Delta_2, a : \mathtt{ch}(\widetilde{T})/V_2 \vdash \widetilde{e} : \widetilde{T}} \quad \dfrac{\pi_Q}{\varphi;\Phi;\Delta'_2, a : \mathtt{ch}(\widetilde{T})/V'_2 \vdash Q \triangleleft K'_2}}{\varphi;\Phi;\uparrow^{J'_c}(\Delta_2 \mid \Delta'_2), a : \mathtt{ch}(\widetilde{T})/\mathtt{Out}^{[0,0]}_{J'_c}.(V_2 \mid V'_2) \vdash \overline{a}\langle\widetilde{e}\rangle.Q \triangleleft J'_c; K'_2}\ (1)}}{\varphi;\Phi;\Delta_1, a : \mathtt{ch}(\widetilde{T})/V_1 \vdash \overline{a}\langle\widetilde{e}\rangle.Q \triangleleft K'_1}}{\varphi;\Phi;\uparrow^{[m,m]}\Delta_1, a : \mathtt{ch}(\widetilde{T})/\uparrow^{[m,m]}V_1 \vdash m : \overline{a}\langle\widetilde{e}\rangle.Q \triangleleft K'_1 + [m,m]}\ (2)}}{\varphi;\Phi;\Delta_0, a : \mathtt{ch}(\widetilde{T})/V_0 \vdash m : \overline{a}\langle\widetilde{e}\rangle.Q \triangleleft K'_0}$$

where (1) is:

$$\varphi; \Phi \vdash \Delta_1 \sqsubseteq \uparrow^{J'_c}(\Delta_2 \mid \Delta'_2) \qquad \varphi; \Phi \vdash V_1 \sqsubseteq \mathtt{Out}^{[0,0]}_{J'_c}.(V_2 \mid V'_2) \qquad \varphi; \Phi \vDash J'_c; K'_2 \subseteq K'_1$$

(2) is:

$$\varphi; \Phi \vdash \Delta_0 \sqsubseteq \uparrow^{[n,n]}\Delta_1; V_0 \sqsubseteq \uparrow^{[m,m]}V_1; K'_1 + [m,m] \subseteq K'_0$$

(3) is:

$$\varphi; \Phi \vdash \Gamma_1 \sqsubseteq \uparrow^{J_c}\Gamma_2; U_1 \sqsubseteq \mathtt{In}^{[0,0]}_{J_c}.U_2; J_c; K_2 \subseteq K_1$$

(4) is:

$$\varphi; \Phi \vdash \Gamma_0 \sqsubseteq \uparrow^{[n,n]}\Gamma_1; U_0 \sqsubseteq \uparrow^{[n,n]}U_1; K_1 + [n,n] \subseteq K_0$$

First, we know that $\Gamma_0 \mid \Delta_0$ is defined. Moreover, we have

$$\varphi; \Phi \vdash \Gamma_0 \sqsubseteq \uparrow^{[n,n]}\Gamma_1 \qquad \varphi; \Phi \vdash \Gamma_1 \sqsubseteq \uparrow^{J_c}\Gamma_2 \qquad \varphi; \Phi \vdash \Delta_0 \sqsubseteq \uparrow^{[m,m]}\Delta_1 \qquad \Delta_1 \sqsubseteq \uparrow^{J'_c}(\Delta_2 \mid \Delta'_2)$$

So, for the channel and server types, in those seven contexts, the shape of the type does not change (only the usage can change). We also have:

$$\Gamma_0^{\mathsf{Nat}} = \Delta_0^{\mathsf{Nat}} \qquad \varphi; \Phi \vdash \Gamma_0^{\mathsf{Nat}} \sqsubseteq \Gamma_1^{\mathsf{Nat}} \sqsubseteq \Gamma_2^{\mathsf{Nat}} \qquad \varphi; \Phi \vdash \Delta_0^{\mathsf{Nat}} \sqsubseteq \Delta_1^{\mathsf{Nat}} \sqsubseteq \Delta_2^{\mathsf{Nat}} \qquad \Delta_2^{\mathsf{Nat}} = \Delta_2'^{\mathsf{Nat}}$$

So, from π_e and π_P we obtain by subtyping:

$$\varphi; \Phi; \Gamma_0^{\mathsf{Nat}}, \Gamma_2^{\nu}, a : \mathtt{ch}(\widetilde{T})/U_2, \widetilde{v} : \widetilde{T} \vdash P \triangleleft K_2 \qquad \varphi; \Phi; \Gamma_0^{\mathsf{Nat}}, \Delta_2^{\nu}, a : \mathtt{ch}(\widetilde{T})/V_2 \vdash \widetilde{e} : \widetilde{T}$$

So, we use the substitution lemma (Lemma 31) and we obtain:

$$\varphi; \Phi; \Gamma_0^{\mathsf{Nat}}, (\Gamma_2^{\nu} \mid \Delta_2^{\nu}), a : \mathtt{ch}(\widetilde{T})/(U_2 \mid V_2) \vdash P[\widetilde{v} := \widetilde{e}] \triangleleft K_2$$

As previously, by subtyping from π_Q, we have:

$$\varphi; \Phi; \Gamma_0^{\mathsf{Nat}}, \Delta_2'^{\nu}, a : \mathtt{ch}(\widetilde{T})/V'_2 \vdash Q \triangleleft K'_2$$

Thus, with the parallel composition rule (as parallel composition of context is defined) and subtyping we have:

$$\varphi; \Phi; \Gamma_0^{\mathsf{Nat}}, (\Gamma_2^{\nu} \mid \Delta_2^{\nu} \mid \Delta_2'^{\nu}), a : \mathtt{ch}(\widetilde{T})/(U_2 \mid V_2 \mid V'_2) \vdash (P[\widetilde{v} := \widetilde{e}] \mid Q) \triangleleft K_2 \sqcup K'_2$$

Let us denote $M = \max(m,n)$. Thus, we derive the proof:

$$\frac{\varphi; \Phi; \Gamma_0^{\mathsf{Nat}}, (\Gamma_2^{\nu} \mid \Delta_2^{\nu} \mid \Delta_2'^{\nu}), a : \mathtt{ch}(\widetilde{T})/(U_2|V_2V'_2) \vdash (P[\widetilde{v} := \widetilde{e}] \mid Q) \triangleleft K_2 \sqcup K'_2}{\varphi; \Phi; \Gamma_0^{\mathsf{Nat}}, \uparrow^{[M,M]}(\Gamma_2^{\nu}|\Delta_2^{\nu}|\Delta_2'^{\nu}), a : \mathtt{ch}(\widetilde{T})/\uparrow^{[M,M]}(U_2|V_2|V'_2) \vdash M : (P[\widetilde{v} := \widetilde{e}] \mid Q) \triangleleft (K_2 \sqcup K'_2) + [M,M]}$$

Now, recall that by hypothesis, $U_0 \mid V_0$ is reliable. We have:

$$\varphi; \Phi \vdash U_0 \sqsubseteq \uparrow^{[n,n]}U_1 \qquad \varphi; \Phi \vdash U_1 \sqsubseteq \mathtt{In}^{[0,0]}_{J_c}.U_2 \qquad \varphi; \Phi \vdash V_0 \sqsubseteq \uparrow^{[m,m]}V_1 \qquad \varphi; \Phi \vdash V_1 \sqsubseteq \mathtt{Out}^{[0,0]}_{J'_c}(V_2 \mid V'_2)$$

So, by Point 1 of Lemma 26, with transitivity and parallel composition of subusage, we have:

$$\varphi; \Phi \vdash U_0 \mid V_0 \sqsubseteq (\uparrow^{[n,n]}U_1) \mid (\uparrow^{[m,m]}V_1) \sqsubseteq \mathtt{In}^{[n,n]}_{J_c}.U_2 \mid \mathtt{Out}^{[m,m]}_{J'_c}(V_2 \mid V'_2)$$

By Point 3 of Lemma 26, we have $\mathtt{In}_{J_c}^{[n,n]}.U_2 \mid \mathtt{Out}_{J'_c}^{[m,m]}(V_2 \mid V'_2)$ reliable. So, in particular, we have:

$$\varphi; \Phi \vdash \mathtt{In}_{J_c}^{[n,n]}.U_2 \mid \mathtt{Out}_{J'_c}^{[m,m]}(V_2 \mid V'_2) \longrightarrow \uparrow^{[M,M]}(U_2 \mid V_2 \mid V'_2)$$

$$\varphi; \Phi \vDash [n,n] \subseteq [m,m] \oplus J'_c \qquad \varphi; \Phi \vDash [m,m] \subseteq [n,n] \oplus J_c$$

Thus, we deduce that

$$\varphi; \Phi \vDash [M,M] \subseteq [n,n] + J_c \qquad \varphi; \Phi \vDash [M,M] \subseteq [m,m] + J'_c$$

So, we have in particular, with Lemma 25 and Point 1 of Lemma 26 and parallel composition:

$$\varphi; \Phi \vdash \Gamma_0 \mid \Delta_0 \sqsubseteq (\uparrow^{[n,n]}\Gamma_1) \mid \uparrow^{[m,m]}\Delta_1 \sqsubseteq (\uparrow^{[n,n]+J_c}\Gamma_2) \mid (\uparrow^{[m,m]+J'_c}(\Delta_2 \mid \Delta'_2)) \sqsubseteq \uparrow^{[M,M]}(\Gamma_2 \mid \Delta_2 \mid \Delta'_2)$$

We also have

$$\varphi; \Phi \vDash K_2 + [M,M] \subseteq J_c; K_2 + [n,n] \subseteq K_0 \qquad \varphi; \Phi \vDash K'_2 + [M,M] \subseteq J'_c; K'_2 + [m,m] \subseteq K'_0$$

So, we obtain directly $\varphi; \Phi \vDash (K_2 \sqcup K'_2) + [M,M] \subseteq K_0 \sqcup K'_0$

Thus, we can simplify a bit the derivation given above, and we have:

$$\dfrac{\dfrac{\varphi; \Phi; \Gamma_0^{\mathsf{Nat}}, (\Gamma_2^{\nu} \mid \Delta_2^{\nu} \mid \Delta_2'^{\nu}), a : \mathtt{ch}(\widetilde{T})/(U_2 \mid V_2 \mid V'_2) \vdash (P[\widetilde{v} := \widetilde{e}] \mid Q) \triangleleft K_2 \sqcup K'_2}{\varphi; \Phi; \Gamma_0^{\mathsf{Nat}}, \uparrow^{[M,M]}(\Gamma_2^{\nu} | \Delta_2^{\nu} | \Delta_2'^{\nu}), a : \mathtt{ch}(\widetilde{T})/\uparrow^{[M,M]}(U_2 | V_2 | V'_2) \vdash M : (P[\widetilde{v} := \widetilde{e}] \mid Q) \triangleleft (K_2 \sqcup K'_2) + [M,M]}}{\varphi; \Phi; (\Gamma_0 \mid \Delta_0), a : \mathtt{ch}(\widetilde{T})/\uparrow^{[M,M]}(U_2 \mid V_2 \mid V'_2) \vdash M : (P[\widetilde{v} := \widetilde{e}] \mid Q) \triangleleft K_0 \sqcup K'_0}$$

By Point 2 of Lemma 26, there exists W such that:

$$\varphi; \Phi \vdash U_0 \mid V_0 \longrightarrow^* W \qquad \varphi; \Phi \vdash W \sqsubseteq \uparrow^{[M,M]}(U_2 \mid V_2 \mid V'_2)$$

So, by subtyping we have a proof:

$$\varphi; \Phi; \Gamma_0 \mid \Delta_0, a : \mathtt{ch}(\widetilde{T})/W \vdash M : (P[\widetilde{v} := \widetilde{e}] \mid Q) \triangleleft K_0 \sqcup K'_0$$

This concludes this case.