Implementations of Sized Types for Parallel Complexity of Message-passing Processes

Thomas Herrmann, Hans Hüttel, Naoki Kobayashi, and Mikkel Korup Lauridsen

Somewhere

Abstract. We provide a sound framework for analyzing the parallel complexity of π-calculus processes in the form of a type checker and a type inference algorithm. The type checker can verify complexity analyses of some polynomial- and linear time primitive recursive functions, encoded as replicated channel inputs (servers), by using integer programming to bound channel synchronizations. The type inference algorithm involves over-approximating the constraint satisfaction problems; we provide a Haskell implementation using the Z3 SMT solver.

1 Introduction

Static analysis of computational complexity has long been a central part of algorithm design and computer science as a whole. One way of performing such a static analysis on a program is by means of type-based techniques. Traditionally, soundness properties have pertained to the absence of certain run-time errors, i.e. *well-typed programs do not go wrong* [17] but with the advent of behavioral type disciplines, soundness properties that for instance ensure bounds on the resource use of programs have been proved.

Research on type systems for computational complexity has originally focused on sequential programs, using a notion of types that can express sizes of terms in a program, referred to as sized types, which have been both formalized and implemented [9,7,6,14,1]. However, there is particularly interest in static complexity analysis of parallel and concurrent computation, following the trend for programs to increase in size, as distributed systems scale better. Moreover, parallel and especially concurrent computation is significantly more difficult to analyze, and so the work on type systems for static complexity analysis has been extended to parallel computation [8], and more recently to the more intricate domain of message-passing processes, using behavioral type disciplines to bound message-passing [2,3].

Baillot and Ghyselen [2] introduce a type system for parallel computational complexity of π-calculus processes, extended with naturals and pattern matching as a computational model, combining sized types and input/output types to bound synchronizations on channels, and thereby bound the parallel time complexity of a process. Baillot et al. [3] generalize the type system using the more expressive behavioral type discipline usage types [11,13]. However, these type systems are quite abstract and build on a notion of sized types that has yet to be implemented in the context of message-passing, and so neither type checking nor type inference has until now been realized for either type system.

In this thesis, we explore the challenges of implementing type checking and type inference for the type system by Baillot and Ghyselen [2]. An important part of this type system is the concept of indices. That is, arithmetic expressions that may contain index variables that represent unknown sizes, thereby enabling a notion of size polymorphism. Indices appear in sized types, to for instance express the timesteps at which a channel must synchronize, which may depend on the size of a value received on a replicated input. To compute an upper bound on the parallel complexity, a partial order on channel synchronizations is required, represented as index comparisons that we refer to as constraint judgements and read as: *Provided a set of constraints on valuations of index variables, is one index always less than or equal to another?* Many of the challenges that arise for both type checking and type inference are related to either verification or satisfaction of such judgements.

We implement a type checker for the type system by Baillot and Ghyselen. This effort is two-fold: We define algorithmic type rules and show how constraint judgements on linear indices can be verified using integer programming or alternatively be over-approximated as linear programs. The type system makes heavy use of subtyping, which we partially account for using *combined complexities* that are effectively sets of indices with a number of associated functions that enable us to discard indices we can guarantee to be bounded by other indices. Combined complexities have the advantage that we can defer finding a single index representing a least upper bound until a later time, and we can in fact show that the combined complexity of a closed process can always be reduced to a singleton, i.e. a singular complexity bound.

We prove that our type checker is sound with regards to time complexity, and to this effort we prove a subject reduction property. Our soundness results guarantee that the bounds assigned to well-typed processes by our type checker are indeed upper bounds on the parallel complexity. To increase the expressiveness of the type checker, we also show how constraints on monotonic univariate polynomial indices can be reduced to linear constraints.

We also define a type inference algorithm for the type system by Baillot and Ghyselen. We take a constraint based approach akin to that of [9,7,6,13,12,15], where unknown indices are represented by *templates*: linear functions over a set of known index variables with unknown coefficients represented by coefficient variables. Inspired by Kobayashi et al. [13], we first infer simple types, which are then used to infer a constraint satisfaction problem on use-capabilities and subtyping which we then reduce to constraints of the form $\exists \alpha_1, \ldots, \alpha_n . \forall i_1, \ldots, i_m . C_1 \land \cdots \land C_k \implies I \leq J$ where $\alpha_1, \ldots, \alpha_n$ are coefficient variables, $i_1, \ldots, i_m$ are index variables, $C_1, \ldots, C_k$ are inequality constraints on indices and I and J are indices.

We provide a Haskell implementation of our type inference algorithm using the Z3 SMT solver [16]. We naively eliminate universal quantifiers by over-approximating our constraints using coefficient-wise inequality constraints. We account for antecedents $C_1, \dots, C_k$ by substitution. For instance, if we can deduce that coefficient c is positive in the constraint $\exists \alpha_1, \dots, \alpha_n . \forall i_1, \dots, i_j, \dots, i_m . K \leq L + c i_j \wedge C_1 \wedge \dots \wedge C_k \implies I \leq J$, then we can simulate the antecedent by substituting $\frac{K-L}{c} + i_j$ for i_j, i.e. $\exists \alpha_1, \dots, \alpha_n . \forall i_1, \dots, i_j, \dots, i_m . C_1\{\frac{K-L}{c} + i_j / i_j\} \wedge \dots \wedge C_k\{\frac{K-L}{c} + i_j / i_j\} \implies I\{\frac{K-L}{c} + i_j / i_j\} \leq J\{\frac{K-L}{c} + i_j / i_j\}$. Using these over-approximations, our implementation is able to infer precise bounds on several processes containing replicated inputs with linear time complexity.

1.1 Related work

Hughes et al. [10] introduce the concept of sized types in the domain of reactive systems, to prove the absence of deadlocks and non-termination in embedded programs. This notion of sized types statically bounds the maximum size that variables may take during run-time using linear functions over size variables (which they refer to as *size indices*). Since the introduction of sized types, several pieces of work regarding computational complexity have been made based on these. Hofmann and Jost [9] extend the use of sized types to analyzing the space complexity of sequential programs. To infer the space complexity of a program, they set up so-called templates for unknown sizes that are sums of terms for universally quantified size variables with unknown coefficients represented as existentially quantified variables. These templates must satisfy certain constraints imposed by the typing of a program. By restricting themselves to linear bounds, they then find a solution to said constraints using linear programming, thus inferring size bounds.

Hoffmann and Hofmann [7] introduce a type system that utilizes a potential-based amortized analysis to infer resource bounds on sequential programs. In addition to using amortized analysis to infer tighter bounds, they are also able to type programs with certain polynomial bounds by deriving linear constraints on the coefficients of the polynomials. They use a unique representation of polynomials using binomial coefficients, which have a number of advantages, increasing the expressiveness of the type system. One of their limitations regarding polynomials, is that they can only infer sums of univariate polynomials, which means that a polynomial bound such as nm must be overestimated by $n^2 + m^2$. As they derive linear constraints, they are able to infer types in much the same way as Hofmann and Jost using linear programming. In addition to implementing type inference, they also prove that their analysis is sound. Hoffmann et al. [6] show how this can be extended to multivariate polynomials, and prove that the system is sound.

Dal Lago and Gaboardi [4] present a family of type systems based on dependent types and linear logic. As the previously mentioned type systems, they also use a notion of sized types based on indices. They rely on an underlying logic of indices, that has partly unspecified function symbols. However, they require that the function symbols of indices include the addition and monus operators. This is not unlike the type systems by both Baillot and Ghyselen [2] as well as Baillot et al. [3]. Dal Lago and Gaboardi prove soundness of the family of type systems as well as *relative completeness.* They only prove relative completeness as the actual completeness of any type system in the family depends on the completeness of the underlying logic of indices of that type system. As such, they also provide no implementation of type checking nor type inference. Instead, they discuss the undecidability of type checking in the general sense, which is the case due to semantic assumptions of indices as well as subtyping judgements.

With the continuous rise in popularity of distributed systems and parallel programs, work has also been done on complexity analysis of parallel programs. Of such is the work by Hoffmann and Shao [8], who introduce the first type system for automatic analysis for deriving complexity bounds on parallel first-order functional programs. They analyze both the work and span of a program using two different methods. For analyzing the work, they simply use the method described by Hoffmann et al., however, for analyzing the span they use a novel method. The main challenge in this work is to extend the potential method to parallel programs, while ensuring the type system is composable and sound. As for some of the previously mentioned type systems, they reduce the problem of type inference to a linear programming problem that they solve to obtain types for the program. However, this type system does not allow parallel subprograms to pass messages to each other, and so they cannot communicate.

A type system that differs widely from the previously mentioned ones is introduced by Das et al. [5]. They introduce a type system for parallel complexity of the π-calculus based on session types by extending them with the *temporal modalities next*, *always*, and *eventually.* By using session types, they naturally consider the sessions between processes and not the processes directly. This has both advantages and disadvantages in that they may describe useful information between communications of processes, but less information about the program as a whole. However, with the rise in distributed systems, it may be beneficial to focus on the protocols between processes. However, this type system does not use sized types, and so only constant time complexity bounds can be expressed.

A central notion in the type system by Baillot et al. is that of *usages.* The idea of usages has roots in work by Kobayashi [11], where usages are used to partially ensure deadlock-freedom in the π-calculus. Sumii and Kobayashi [19] refine the definition of usages and present a type system for deadlock-freedom. When considering deadlock freedom analysis and complexity analysis, we are interested in similar program properties as both consider the run-time of programs. Usages describe the behavior of channels by indicating how they are used for input and output in parallel and sequentially. A key concept of usages is that of obligations and capabilities which describe when a channel is ready to communicate and when communication may succeed, respectively. Kobayashi et al. [13] introduce a type inference algorithm for the type system. To do this, they introduce a generalization of usages as well as a subusage relation. They take a constraint based approach to type inference, inferring constraint satisfaction problems that have solutions that match each possible typing of a process.

A recurring theme for many of the aforementioned type systems based on sized types is that of their choice of underlying logic for indices. Some of the type systems specify a specific logic (e.g. [10,9,7,6]), while others abstain from specifying one (e.g. [2,3,4]). For the latter, completeness is relative to the choice of logic.

The rest of our paper is structured as follows: In Section 2, we present the variant of the π-calculus and the notion of parallel complexity used in the type systems by Baillot and Ghyselen [2] and Baillot et al. [3]. In Section **??**, we discuss sized types and provide an overview of the type system by Baillot and Ghyselen, aiming to establish familiarity with these topics. In Section **??**, we define a type checker for this type system, first presenting algorithmic type rules and then showing how premises can be verified using integer programming. We prove that our algorithmic type rules are sound with respect to parallel complexity. In Section **??**, we define a type inference algorithm for the type system, automating parallel complexity analysis of message-passing processes. We also provide a Haskell implementation of our algorithm. Finally, we conclude and discuss future work in Section **??**.

2 The π-calculus

We first introduce a extended π-calculus and a cost model for time that uses process annotations to represent incurred reduction costs. Using this, we can then define the notion of parallel complexity.

2.1 Syntax and semantics

We consider an asynchronous polyadic π-calculus extended with naturals as algebraic terms and a pattern matching construct that enables deconstruction of such terms. The languages of processes and expressions are defined by the syntax

$$\begin{aligned} P, Q \text{ (processes) } &::= \mathbf{0} \mid (P \mid Q) \mid a(\widetilde{v}).P \mid \overline{a}\langle\widetilde{e}\rangle \mid !a(\widetilde{v}).P \mid (\nu a)P \mid \texttt{tick}.P \mid \\ &\quad\ \texttt{match}\ e\ \{0 \mapsto P;\ s(x) \mapsto Q\} \\ e \text{ (expressions) } &::= 0 \mid s(e) \mid v \end{aligned}$$

where we assume a countably infinite set of names **Var**, such that $a, b, c \in \mathbf{Var}$ represent channels, $x, y, z \in \mathbf{Var}$ are bound to algebraic terms and the meta-variable $v \in \mathbf{Var}$ may be bound to any expression. As usual, we have inaction $\mathbf{0}$ representing a terminated process, the parallel composition of two processes $P \mid Q$ and restrictions $(\nu a)P$. For polyadic inputs and outputs $a(\widetilde{v}).P$ and $\overline{a}\langle\widetilde{e}\rangle$, we use the notation $\widetilde{v}$ and $\widetilde{e}$, respectively, to denote sequences of names $v_1, \ldots, v_n$ and expressions $e_1, \ldots, e_n$, and we write $P[\widetilde{v} \mapsto \widetilde{e}]$ to denote the substitution $P[v_1 \mapsto e_1, \ldots, v_n \mapsto e_n]$ for names in process P. Similarly, for nested restrictions $(\nu a_1) \cdots (\nu a_n)P$ we may write $(\nu\widetilde{a})P$. For technical convenience, we only enable replication on inputs, i.e. $!a(\widetilde{v}).P$, such that we can more easily determine which channels induce recursive behavior. Note that any replicated process $!P$

can be simulated using a replicated input $!a().(P \mid \overline{a}\langle\rangle) \mid \overline{a}\langle\rangle)$.

Naturals are represented by a zero constructor 0, representing the smallest natural number, and a successor constructor $s(e)$ that represents the successor of the natural number e. Thus, algebraic terms have a clearly distinguishable base case. We use this for the pattern matching constructor that deconstructs such terms, by branching based on the shapes of the expressions. This will be useful later, as this enables us to analyze termination conditions for some processes with recursive behavior. The `tick` constructor represents a cost in time complexity of one. We provide a detailed account of this constructor in Subsection 2.2.

In Definition 1, we introduce the usual structural congruence relation [**?**]. We omit rules for replication, as these will be covered by the reduction relation. The congruence relation essentially introduces associative and commutative properties to parallel compositions and enables widening or narrowing of the scopes of restrictions, thereby simplifying the semantics.

Definition 1 (Structural congruence). *We define structural congruence $\equiv$ as the least congruence relation that satisfies the rules*

$$(\text{SC-}\textsc{nil})\ \ P \mid \mathbf{0} \equiv P \qquad (\text{SC-}\textsc{commu})\ \ P \mid Q \equiv Q \mid P$$

$$(\text{SC-}\textsc{assoc})\ \ P \mid (Q \mid R) \equiv (P \mid Q) \mid R$$

$$(\text{SC-}\textsc{scope})\ \ (\nu a)\,(P \mid Q) \equiv (\nu a)P \mid Q \quad \textit{if } a \textit{ is not free in } Q$$

$$(\text{SC-}\textsc{par})\ \ \frac{P \equiv P'}{P \mid Q \equiv P' \mid Q} \qquad\qquad (\text{SC-}\textsc{res})\ \ \frac{P \equiv Q}{(\nu a)P \equiv (\nu a)Q}$$

We extend the usual definition of structural congruence with rules that enable us to group restrictions. This allows us to introduce a normal form for processes that simplifies the definition of parallel complexity or span, and consequently makes it easier to prove various properties for typed π-calculi. We formalize the normal form in Definition 2, referring to it as the canonical form. Essentially, we can view a process in canonical form as a sequence of bound names and a multiset of guarded processes. In Lemma 1, we prove that an arbitrary process is structurally congruent to a process in this form.

Definition 2 (Canonical form). *We say that a process P is in canonical form if is has the shape $(\nu\widetilde{a})\,(G_1 \mid G_2 \mid \ldots \mid G_n)$ where $G_1, G_2, \ldots, G_n$ are referred to as guarded processes which may be of any of the forms*

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

If $n = 0$ then the canonical form is $(\nu\widetilde{a})\mathbf{0}$.

Lemma 1 (Existence of a canonical form). *Let P be a process. Then there exists a process Q in canonical form such that $P \equiv Q$.*

Proof. Suppose by α-renaming that all names are unique, then by using rule (SC-RES) *from left to right and* (SC-SCOPE) *from right to left, we can widen the scope of all unguarded restrictions, such that all unguarded restrictions are outmost. Then using rule* (SC-RES) *and* (SC-ZERO) *from left to right, we remove all unguarded inactions. If the whole process is unguarded, one inaction will remain and we have the canonical form* $(\nu\widetilde{a} : \widetilde{T})\mathbf{0}$.

In Table 1, we define the reduction relation $\longrightarrow$ for π-calculus processes, such that $P \longrightarrow Q$ denotes that process P reduces to Q in one reduction step [**?**]. We have the usual rules for the asynchronous polyadic π-calculus, enriched with rules for replicated inputs, pattern matching and temporal reductions. The former is covered by rule (R-REP) that synchronizes a replicated input with an output, preserving the replicated input for subsequent synchronizations. Rule (R-ZERO) considers the base case for pattern matching on naturals. For pattern matches on successors, we substitute the *deconstructed* terms for variables bound in the patterns of the pattern match constructors. Finally, rule (R-TICK) reduces a tick prefix, representing a cost of one in time complexity.

$$\text{(R-REP)}\ \frac{}{!a(\widetilde{v}).P \mid \overline{a}\langle\widetilde{e}\rangle \longrightarrow\ !a(\widetilde{v}).P \mid P[\widetilde{v} \mapsto \widetilde{e}]} \qquad \text{(R-COMM)}\ \frac{}{a(\widetilde{v}).P \mid \overline{a}\langle\widetilde{e}\rangle \longrightarrow P[\widetilde{v} \mapsto \widetilde{e}]}$$

$$\text{(R-ZERO)}\ \frac{}{\texttt{match}\ 0\ \{0 \mapsto P;\ s(x) \mapsto Q\} \longrightarrow P} \qquad \text{(R-PAR)}\ \frac{P \longrightarrow Q}{P \mid R \longrightarrow Q \mid R}$$

$$\text{(R-SUCC)}\ \frac{}{\texttt{match}\ s(e)\ \{0 \mapsto P;\ s(x) \mapsto Q\} \longrightarrow Q[x \mapsto e]} \qquad \text{(R-TICK)}\ \frac{}{\texttt{tick}.P \longrightarrow P}$$

$$\text{(R-RES)}\ \frac{P \longrightarrow Q}{(\nu a)P \longrightarrow (\nu a)Q} \qquad \text{(R-STRUCT)}\ \frac{P \equiv P' \quad P' \longrightarrow Q' \quad Q' \equiv Q}{P \longrightarrow Q}$$

Table 1. The reduction rules defining $\longrightarrow$.

2.2 Parallel complexity

There are several tried approaches to modeling time complexity for the π-calculus. One way is to incur a cost for every axiomatic reduction, in our case whenever a channel synchronizes or a pattern match branches. A widely used alternative, including the one chosen by Baillot and Ghyselen, is to introduce an explicit process prefix constructor $\texttt{tick}.P$ denoting that the continuation P is preceded by a cost in time complexity [2,3,5]. This approach is more flexible, as we can simulate many different cost models, based on placement patterns of tick prefixes. We now present some of the properties of the tick constructor with respect to parallel complexity. We are interested in maximizing the parallelism, and to do so we must reduce ticks in parallel. Consider the process

$$\overbrace{\mathtt{tick}.\mathbf{0} \mid \mathtt{tick}.\mathbf{0} \mid \cdots \mid \mathtt{tick}.\mathbf{0}}^{n}$$

The sequential complexity or work is n, whereas the parallel complexity or span is 1, as we can reduce the n ticks in parallel. To keep track of the span during reduction, we introduce integer annotations to processes, such that process P can be represented as $m : P$ where m represents the time already incurred. We refer to such processes as *annotated processes*, and we enrich the definition of structural congruence with four additional rules that include time annotations in Definition 3. Intuitively, this means that process annotations can be moved outward and may be summed. Any process is also structurally congruent to one with an extra annotation of 0.

Definition 3. *We enrich the definition of structural congruence with the four rules*

$$(\text{SC-}\textsc{dis})\ \ m : (P \mid Q) \equiv (m : P) \mid (m : Q) \qquad\qquad (\text{SC-}\textsc{ares})\ \ m : (\nu a)P \equiv (\nu a)(m : P)$$
$$(\text{SC-}\textsc{sum})\ \ m : (n : P) \equiv (m + n) : P \qquad\qquad (\text{SC-}\textsc{zero})\ \ 0 : P \equiv P$$

Annotations on processes introduce another notion of canonical process that can be seen in Definition 4. That is, an annotated process in canonical form is a sequence of nested restrictions on a parallel composition of annotated guarded processes.

Definition 4 (Annotated canonical form). *An annotated process is in canonical form if it is of the form*

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

where $G_1, \ldots, G_m$ are guarded annotated processes. If $m = 0$, its canonical form is $(\nu\widetilde{a})(0 : \mathbf{0})$.

Lemma 2 (Existence of an annotated canonical form). *Let P be an annotated process. Then there exists an annotated process Q in annotated canonical form such that $P \equiv Q$.*

Proof. Suppose by α-renaming that all names are unique, then by using rule (SC-RES), (SC-ARES), (SC-DIS) *and* (SC-SUM) *from left to right, and* (SC-SCOPE) *from right to left, we can widen the scope of all unguarded restrictions, such that all unguarded restrictions are outmost, and all non-guarded annotations are prefixes to guarded processes. Then using rule* (SC-RES) *and* (SC-ZERO) *from left to right, we remove all unguarded inactions. For all remaining guarded processes that are not prefixed with an annotation, we use* (SC-RES), (SC-PAR) *and* (SC-ZERO) *to introduce prefixing* 0*-annotations. If the whole process is unguarded, one inaction will remain and we have the canonical form* $(\nu\widetilde{a} : \widetilde{T})\mathbf{0}$.

With annotated processes, we can define the parallel reduction relation $\Longrightarrow$ in Table 2. Most notably, we can see that the tick constructor reduces to an annotation of 1, and during communication we choose the maximum annotation amongst the two endpoints.

$$(\text{PR-REP})\ \frac{}{(n : !a(\widetilde{v}).P) \mid (m : \overline{a}\langle\widetilde{e}\rangle) \Longrightarrow (n : !a(\widetilde{v}).P) \mid (\max(n, m) : P[\widetilde{v} \mapsto \widetilde{e}])}$$

$$(\text{PR-COMM})\ \frac{}{(n : a(\widetilde{v}).P) \mid (m : \overline{a}\langle\widetilde{e}\rangle) \Longrightarrow \max(n, m) : P[\widetilde{v} \mapsto \widetilde{e}]}$$

$$(\text{PR-TICK})\ \frac{}{\texttt{tick}.P \Longrightarrow 1 : P} \qquad (\text{PR-ZERO})\ \frac{}{\texttt{match}\ 0\ \{0 \mapsto P;\ s(x) \mapsto Q\} \Longrightarrow P}$$

$$(\text{PR-SUCC})\ \frac{}{\texttt{match}\ s(e)\ \{0 \mapsto P;\ s(x) \mapsto Q\} \Longrightarrow Q[x \mapsto e]}$$

$$(\text{PR-PAR})\ \frac{P \Longrightarrow Q}{P \mid R \Longrightarrow Q \mid R} \qquad (\text{PR-RES})\ \frac{P \Longrightarrow Q}{(\nu a)P \Longrightarrow (\nu a)Q}$$

$$(\text{PR-ANNOT})\ \frac{P \Longrightarrow Q}{n : P \Longrightarrow n : Q} \qquad (\text{PR-STRUCT})\ \frac{P \equiv P' \quad P' \Longrightarrow Q' \quad Q' \equiv Q}{P \Longrightarrow Q}$$

Table 2. The reduction rules defining $\Rightarrow$.

In Definition 5, we define the local parallel complexity $C_\ell(P)$ of a process P. Intuitively, the local complexity is the maximal integer annotation in the canonical form of P.

Definition 5 (Local complexity). *We define the local parallel complexity $\mathcal{C}_\ell(P)$ of a process P by the following rules*

$$\mathcal{C}_\ell(n : P) = n + \mathcal{C}_\ell(P) \qquad \mathcal{C}_\ell(P \mid Q) = \mathit{max}(\mathcal{C}_\ell(P), \mathcal{C}_\ell(Q))$$
$$\mathcal{C}_\ell((\nu a)P) = \mathcal{C}_\ell(P) \qquad \mathcal{C}_\ell(G) = 0 \textit{ if } G \textit{ is a guarded process}$$

We now formalize the parallel complexity or span of a process in Definition 6. To account for the non-determinism of the π-calculus, the parallel complexity of a process is defined as the maximal integer annotation in any reduction sequence. To see why this is necessary, consider the process

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

We have two possible reduction sequences with different integer annotations. That is, if we were to reduce the left-most input first, we have a single time reduction, as the second tick will be guarded. If we instead reduce the second input first, then we can synchronize of channel a again after one time reduction, thus yielding two time reductions.

Definition 6 (Parallel complexity). *We define the parallel complexity (or span) $\mathcal{C}_{\mathcal{P}}(P)$ of process P as the maximal local complexity of any reduction sequence from P.*

$$\mathcal{C}_{\mathcal{P}}(P) = max\{n \mid P \Longrightarrow^* Q \wedge \mathcal{C}_\ell(Q) = n\}$$

where $\Longrightarrow^*$ is the reflexive and transitive closure of $\Longrightarrow$.

References

1. Martin Avanzini and Ugo Dal Lago. Automating sized-type inference for complexity analysis. *Proc. ACM Program. Lang.*, 1(ICFP), August 2017.
2. Patrick Baillot and Alexis Ghyselen. Types for complexity of parallel computation in pi-calculus. In Nobuko Yoshida, editor, *Programming Languages and Systems*, pages 59–86, Cham, 2021. Springer International Publishing.
3. Patrick Baillot, Alexis Ghyselen, and Naoki Kobayashi. Sized Types with Usages for Parallel Complexity of Pi-Calculus Processes. In Serge Haddad and Daniele Varacca, editors, *32nd International Conference on Concurrency Theory (CONCUR 2021)*, volume 203 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 34:1–34:22, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.
4. Ugo Dal Lago and Marco Gaboardi. Linear dependent types and relative completeness. In *Proceedings of the 2011 IEEE 26th Annual Symposium on Logic in Computer Science*, LICS '11, page 133–142, USA, 2011. IEEE Computer Society.
5. Ankush Das, Jan Hoffmann, and Frank Pfenning. Parallel complexity analysis with temporal session types. *Proc. ACM Program. Lang.*, 2(ICFP), July 2018.
6. Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. Multivariate amortized resource analysis. *ACM Trans. Program. Lang. Syst.*, 34(3), nov 2012.
7. Jan Hoffmann and Martin Hofmann. Amortized resource analysis with polynomial potential. In Andrew D. Gordon, editor, *Programming Languages and Systems*, pages 287–306, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.
8. Jan Hoffmann and Zhong Shao. Automatic static cost analysis for parallel programs. In *Proceedings of the 24th European Symposium on Programming on Programming Languages and Systems - Volume 9032*, page 132–157, Berlin, Heidelberg, 2015. Springer-Verlag.
9. Martin Hofmann and Steffen Jost. Static prediction of heap space usage for first-order functional programs. *SIGPLAN Not.*, 38(1):185–197, jan 2003.
10. John Hughes, Lars Pareto, and Amr Sabry. Proving the correctness of reactive systems using sized types. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, page 410–423, New York, NY, USA, 1996. Association for Computing Machinery.
11. Naoki Kobayashi. A partially deadlock-free typed process calculus. *ACM Trans. Program. Lang. Syst.*, 20(2):436–482, mar 1998.
12. Naoki Kobayashi. Type-based information flow analysis for the π-calculus. *Acta Inf.*, 42(4):291–347, dec 2005.

13. 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, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.
14. Ugo Dal Lago and Marco Gaboardi. Linear Dependent Types and Relative Completeness. *Logical Methods in Computer Science*, 8(4), 2012.
15. Cédric Lhoussaine. Type inference for a distributed π-calculus. *Sci. Comput. Program.*, 50(1–3):225–251, mar 2004.
16. Microsoft Research. Z3. `https://github.com/Z3Prover/z3`. Accessed: June 11th 2022.
17. Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.
18. Ryan Stansifer. Presburger's article on integer airthmetic: Remarks and translation. Technical Report TR84-639, Cornell University, Computer Science Department, September 1984.
19. Eijiro Sumii and Naoki Kobayashi. A generalized deadlock-free process calculus. *Electronic Notes in Theoretical Computer Science*, 16(3):225–247, 1998. HLCL '98, 3rd International Workshop on High-Level Concurrent Languages (Satellite Workshop of CONCUR '98).