

# 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  $\pi$ -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* [21] but with the advent of behavioral type disciplines, soundness properties that for instance ensure bounds on the resource use of programs have been proved.

Citations here

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 [12,10,9,18,1]. 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 [11], and more recently to the more intricate domain of message-passing processes, using behavioral type disciplines to bound message-passing [3,4].

Baillot and Ghyselen [3] introduce a type system for parallel computational complexity of  $\pi$ -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. [4] generalize the type system using the more expressive behavioral type discipline usage types [15,17]. 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.

paper

In this thesis, we explore the challenges of implementing type checking and type inference for the type system by Baillot and Ghyselen [3]. ~~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.

Omit ?

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 [12,10,9,17,16,19], 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. [17], 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  $\exists \alpha_1, \dots, \alpha_n. \forall i_1, \dots, i_m. C_1 \wedge \dots \wedge C_k \implies I \leq J$  where  $\alpha_1, \dots, \alpha_n$  are coefficient variables,  $i_1, \dots, i_m$  are index variables,  $C_1, \dots, C_k$  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 [20]. ~~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  $e$  is positive in the constraint  $\exists \alpha_1, \dots, \alpha_n. \forall i_1, \dots, i_j, \dots, i_m. K \leq L + e i_j \wedge C_1 \wedge \dots \wedge C_k \implies I \leq J$ , then we can simulate the antecedent by substituting  $\frac{K-L}{e} + i_j$  for  $i_j$ , i.e.  $\exists \alpha_1, \dots, \alpha_n. \forall i_1, \dots, i_j, \dots, i_m. C_1 \{\frac{K-L}{e} + i_j/i_j\} \wedge \dots \wedge C_k \{\frac{K-L}{e} + i_j/i_j\} \implies I \{\frac{K-L}{e} + i_j/i_j\} \leq J \{\frac{K-L}{e} + i_j/i_j\}$ .~~ 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 Condense and move up

Hughes et al. [13] 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 [12] 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 [10] 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. [9] show how this can be extended to multivariate polynomials, and prove that the system is sound.

Dal Lago and Gaboardi [5] 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 [3] as well as Baillot et al. [4]. 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 [11], 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. [6]. They introduce a type system for parallel complexity of the  $\pi$ -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 [15], where usages are used to partially ensure deadlock-freedom in the  $\pi$ -calculus. Sumii and Kobayashi [24] 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. [17] 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.

Move upwards

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. [13,12,10,9]), while others abstain from specifying one (e.g. [3,4,5]). For the latter, completeness is relative to the choice of logic.

Omit??

The rest of our paper is structured as follows: In Section 2, we present the variant of the  $\pi$ -calculus and the notion of parallel complexity used in the type systems by Baillot and Ghyselen [3] and Baillot et al. [4]. 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 4, 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 5, 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 7.

## 2 The $\pi$ -calculus

We first introduce an extended  $\pi$ -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  $\pi$ -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

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

where we assume a countably infinite set of names  $\mathbf{Var}$ , 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  $a(\tilde{v}).P$  and  $\bar{a}(\tilde{e})$ , we use the notation  $\tilde{v}$  and  $\tilde{e}$ , respectively, to denote sequences of names  $v_1, \dots, v_n$  and expressions  $e_1, \dots, e_n$ , and we write  $P[\tilde{v} \mapsto \tilde{e}]$  to denote the substitution  $P[v_1 \mapsto e_1, \dots, v_n \mapsto e_n]$  for names in process  $P$ . Similarly, for nested restrictions  $(\nu a_1) \dots (\nu a_n)P$  we may write  $(\nu \tilde{a})P$ . For technical convenience, we only enable replication on inputs, i.e.  $!a(\tilde{v}).P$ , 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  $!a().(P \mid \bar{a}()) \mid \bar{a}()$ .

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 [22]. 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*

$$\begin{array}{ll}
(\text{SC-NIL}) \quad P \mid \mathbf{0} \equiv P & (\text{SC-COMMU}) \quad P \mid Q \equiv Q \mid P \\
(\text{SC-ASSOC}) \quad P \mid (Q \mid R) \equiv (P \mid Q) \mid R & \\
(\text{SC-SCOPE}) \quad (\nu a)(P \mid Q) \equiv (\nu a)P \mid Q \quad \text{if } a \text{ is not free in } Q & \\
(\text{SC-PAR}) \quad \frac{P \equiv P'}{P \mid Q \equiv P' \mid Q} & (\text{SC-RES}) \quad \frac{P \equiv Q}{(\nu a)P \equiv (\nu a)Q}
\end{array}$$

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  $\pi$ -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 it has the shape  $(\nu \tilde{a})(G_1 \mid G_2 \mid \dots \mid G_n)$  where  $G_1, G_2, \dots, G_n$  are referred to as guarded processes which may be of any of the forms*

$$G ::= !a(\tilde{y}).P \mid a(\tilde{v}).P \mid \bar{a}(\tilde{e}) \mid \text{match } e \{0 \mapsto P; s(x) \mapsto Q\} \mid \text{tick}.P$$

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

**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 a 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  $(\nu \tilde{a} \cdot \tilde{T})\mathbf{0}$ .~~

In Table 1, we define the reduction relation  $\longrightarrow$  for  $\pi$ -calculus processes, such that  $P \longrightarrow Q$  denotes that process  $P$  reduces to  $Q$  in one reduction step [22]. We have the usual rules for the asynchronous polyadic  $\pi$ -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.

(R-REP) $\frac{!a(\tilde{v}).P \mid \bar{a}(\tilde{e})}{!a(\tilde{v}).P \mid P[\tilde{v} \mapsto \tilde{e}]}$	(R-COMM) $\frac{a(\tilde{v}).P \mid \bar{a}(\tilde{e})}{P[\tilde{v} \mapsto \tilde{e}]}$
(R-ZERO) $\frac{}{\text{match } 0 \{0 \mapsto P; s(x) \mapsto Q\} \longrightarrow P}$	(R-PAR) $\frac{P \longrightarrow Q}{P \mid R \longrightarrow Q \mid R}$
(R-SUCC) $\frac{}{\text{match } s(e) \{0 \mapsto P; s(x) \mapsto Q\} \longrightarrow Q[x \mapsto e]}$	(R-TICK) $\frac{}{\text{tick}.P \longrightarrow P}$
(R-RES) $\frac{P \longrightarrow Q}{(\nu a)P \longrightarrow (\nu a)Q}$	(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  $\pi$ -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  $\text{tick}.P$  denoting that the continuation  $P$  is preceded by a cost in time complexity [3,4,6]. 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

allows us to

respect to parallel complexity. We are interested in maximizing the parallelism, and to do so we must reduce ticks in parallel. Consider the process

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

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*

$$\begin{aligned} (\text{SC-DIS}) \quad m : (P \mid Q) &\equiv (m : P) \mid (m : Q) & (\text{SC-ARES}) \quad m : (\nu a)P &\equiv (\nu a)(m : P) \\ (\text{SC-SUM}) \quad m : (n : P) &\equiv (m + n) : P & (\text{SC-ZERO}) \quad 0 : P &\equiv P \end{aligned}$$

~~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*

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

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

**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  $\alpha$ -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  $(\nu \tilde{a} : \tilde{T})\mathbf{0}$ .



## Maybe all we need is the annotated semantics

With annotated processes, we can define the parallel reduction relation  $\Rightarrow$  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.

(PR-REP) $\frac{}{(n : !a(\tilde{v}).P) \mid (m : \bar{a}(\tilde{e})) \Rightarrow (n : !a(\tilde{v}).P) \mid (\max(n, m) : P[\tilde{v} \mapsto \tilde{e}])}$	
(PR-COMM) $\frac{}{(n : a(\tilde{v}).P) \mid (m : \bar{a}(\tilde{e})) \Rightarrow \max(n, m) : P[\tilde{v} \mapsto \tilde{e}]}$	
(PR-TICK) $\frac{}{\text{tick}.P \Rightarrow 1 : P}$	(PR-ZERO) $\frac{}{\text{match } 0 \{0 \mapsto P; s(x) \mapsto Q\} \Rightarrow P}$
(PR-SUCC) $\frac{}{\text{match } s(e) \{0 \mapsto P; s(x) \mapsto Q\} \Rightarrow Q[x \mapsto e]}$	
(PR-PAR) $\frac{P \Rightarrow Q}{P \mid R \Rightarrow Q \mid R}$	(PR-RES) $\frac{P \Rightarrow Q}{(\nu a)P \Rightarrow (\nu a)Q}$
(PR-ANNOT) $\frac{P \Rightarrow Q}{n : P \Rightarrow n : Q}$	(PR-STRUCT) $\frac{P \equiv P' \quad P' \Rightarrow Q' \quad Q' \equiv Q}{P \Rightarrow Q}$

**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  $C_\ell(P)$  of a process  $P$  by the following rules*

$$\begin{aligned}
 C_\ell(n : P) &= n + C_\ell(P) & C_\ell(P \mid Q) &= \max(C_\ell(P), C_\ell(Q)) \\
 C_\ell((\nu a)P) &= C_\ell(P) & C_\ell(G) &= 0 \text{ if } G \text{ is a guarded process}
 \end{aligned}$$

We now formalize the parallel complexity or span of a process in Definition 6. To account for the non-determinism of the  $\pi$ -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

$$a().\text{tick}. \mid a().\text{tick}.\bar{a}() \mid \bar{a}()$$

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  $\Longrightarrow$ .

### 3 Sized types for parallel complexity

In this section, we introduce the type system for parallel complexity of message-passing processes introduced in Baillot and Ghyselen [3]. ~~This type system builds on the foundations of indices and constraint judgements and formalizes parallel complexity analysis of  $\pi$  calculus processes. Due to extensive use of subtyping and the challenges involved in verifying and satisfying constraint judgements, substantial modifications must be made to enable type checking and type inference of processes.~~

The type system ~~for parallel complexity of message-passing processes introduced by Baillot and Ghyselen~~ uses sized types to express parametric complexity of replicated input invocation, and thereby achieves precise bounds on primitively recursive processes: A class of processes behaving as primitively recursive functions. This requires a notion of polymorphism in the message types of replicated inputs. Baillot and Ghyselen introduce size polymorphism by bounding sizes of algebraic terms and synchronizations on channels with indices that may contain index variables representing unknown sizes. We may interpret an index with an index valuation that maps its index variables to naturals, such that the index may be evaluated to a natural number.

We first formally define indices and constraints on the valuations of indices. We give both a predicate logic and a model-theoretic interpretation of judgements on such constraints, referring to these as *constraint judgements*. We then define sized types, the subtyping relation and introduce non-algorithmic type rules.

#### 3.1 Indices and constraint judgements

In the type system by Baillot and Ghyselen, indices are used to keep track of sizes of inputs received on replicated inputs. As these sizes may be parametric, in that they may be dependent on the sizes of values received on replicated inputs, we view indices as algebraic expressions consisting of index variables  $i, j, k \in \mathcal{V}$  ranging over a countable set, and function symbols, using meta-variable  $f$ , that

may represent natural number constants as nullary functions as well as algebraic operators

$$I, J ::= i \mid f(I_1, I_2, \dots, I_n)$$

Each function symbol  $f$  has an arity  $\text{ar}(f)$  and an interpretation  $\llbracket f \rrbracket : \mathbb{N}^{\text{ar}(f)} \rightarrow \mathbb{N}$ . For the interpretation of binary difference, we assume that  $\llbracket - \rrbracket(n, m) = 0$  when  $m \geq n$ , which we refer to as the *monus* operator. As indices may contain index variables, we assume some index valuation  $\rho : \mathcal{V} \rightarrow \mathbb{N}$ , and extend the definition of interpretations to indices, such that  $\llbracket I \rrbracket_\rho$  is a natural number instance of index  $I$ , according to index valuation  $\rho$ , where for all  $i$  in  $I$ ,  $\rho(i)$  substitutes for  $i$  denoted  $I\{\rho(i)/i\}$ . Index substitution is defined in Definition 7. Based on the structure of the process that indices are used in the typing of, we may be able to establish relationships between the instances of these indices. For instance, a replicated input may receive values of sizes defined by an interval of two indices  $[I, J]$ . Then, we are only interested in index valuations  $\rho$  that satisfy  $\llbracket I \rrbracket_\rho \leq \llbracket J \rrbracket_\rho$ . To express such relationships, we define binary constraints on indices in Definition 8.

**Definition 7.** We define index substitution by the following rules

$$\begin{aligned} i\{I/j\} &= j \text{ if } i = j \\ i\{I/j\} &= i \text{ if } i \neq j \\ f(I_1, I_2, \dots, I_n)\{J/i\} &= f(I_1\{J/i\}, I_2\{J/i\}, \dots, I_n\{J/i\}) \end{aligned}$$

**Definition 8 (Index constraints).** Given a finite set of index variables  $\varphi \subset \mathcal{V}$ , we define a constraint  $C$  on  $\varphi$  to be an expression of the form  $I \bowtie J$ , where  $I$  and  $J$  are indices with all free index variables in  $\varphi$  and  $\bowtie \in \{\leq, =, \geq\}$  is a binary relation on  $\mathbb{N}$ . A finite set of constraints is represented by meta-variable  $\Phi$ .

A constraint  $I \bowtie J$  on  $\varphi$  is satisfied given an index valuation  $\rho : \varphi \rightarrow \mathbb{N}$  when  $\llbracket I \rrbracket_\rho \bowtie \llbracket J \rrbracket_\rho$  is satisfied, denoted  $\rho \models I \bowtie J$ . For a finite set of constraints  $\Phi$ , we write  $\rho \models \Phi$  when  $\rho \models C$  holds for all  $C \in \Phi$ . Finally,  $\varphi; \Phi \models C$  holds when for all index valuations  $\rho$  such that  $\rho \models \Phi$  holds, we also have  $\rho \models C$ . That is,  $\varphi; \Phi \models C$  holds exactly when  $C$  does not impose further restrictions on index valuations on  $\varphi$ . Such judgements are fundamental to the type system by Baillot and Ghyselen, especially ones of the form  $\varphi; \Phi \models I \leq J$ , as they impose a partial order on indices wrt. how indices may be interpreted. This enables a notion of subtyping for parametric complexities, such that only indices that are greater or equal may substitute, thus preserving upper bounds on the global parallel complexity, as we shall see in the following sections.

Keep or omit?

### 3.2 Alternative formulations of constraint judgements

There are several equivalent formulations of the problem of verifying the judgement  $\varphi; \Phi \models C_0$ . One such formulation is that the judgement holds, when the

conjunction of constraints in  $\Phi$  implies  $C_0$ , i.e. assuming that  $n \bowtie m$  evaluates to a truth value based on membership in the relation  $\bowtie$ , the predicate formula  $C_1 \wedge \dots \wedge C_n \implies C_0$ , where  $\Phi = \{C_1, \dots, C_n\}$ , must be satisfied for all valuations  $\rho$  over  $\varphi$ . That is, let  $C_i = I_i \bowtie_i J_i$ , then for any valuation  $\rho : \varphi \rightarrow \mathbb{N}$ , the formula  $(\llbracket I_1 \rrbracket_\rho \bowtie_1 \llbracket J_1 \rrbracket_\rho) \wedge \dots \wedge (\llbracket I_n \rrbracket_\rho \bowtie_n \llbracket J_n \rrbracket_\rho) \implies \llbracket I_0 \rrbracket_\rho \bowtie_0 \llbracket J_0 \rrbracket_\rho$  must be satisfied. Another interpretation of the problem is that the intersection of the feasible regions of all (inequality) constraints in  $\Phi$  must be contained in the feasible region of  $C_0$ , or equivalently, the set of all valuations over  $\varphi$  that satisfy all the constraints in  $\Phi$ , referred to as the model space of  $\Phi$  wrt.  $\varphi$ ,  $\mathcal{M}_\varphi(\Phi)$  must be a subset of the model space of  $C_0$  wrt.  $\varphi$

$$\mathcal{M}_\varphi(\Phi) \subseteq \mathcal{M}_\varphi(\{C_0\}) \quad \text{where} \quad \mathcal{M}_\varphi(\Phi) = \{\rho : \varphi \rightarrow \mathbb{N} \mid \rho \models C \text{ for } C \in \Phi\}$$

or equivalently

$$\forall \rho \in \mathcal{M}_\varphi(\Phi) (\rho \in \mathcal{M}_\varphi(\{C_0\}))$$

Finally, given the fact that the current statement of the problem is expressed using a universal quantifier, we can negate the problem, obtaining a problem that can instead be expressed using an existential quantifier by the fact that  $\neg \forall x P(x)$  is equivalent to  $\exists x \neg P(x)$ . This means the problem can also be expressed as

$$\neg(\exists \rho \in \mathcal{M}_\varphi(\Phi) (\rho \notin \mathcal{M}_\varphi(\{C_0\})))$$

or equivalently

$$\mathcal{M}_\varphi(\Phi) \cap \mathcal{M}'_\varphi(\{C_0\}) = \emptyset \quad \text{where} \quad \begin{aligned} \mathcal{M}_\varphi(\Phi) &= \{\rho : \varphi \rightarrow \mathbb{N} \mid \rho \models C \text{ for all } C \in \Phi\} \\ \mathcal{M}'_\varphi(\Phi) &= \{\rho : \varphi \rightarrow \mathbb{N} \mid \rho \not\models C \text{ for some } C \in \Phi\} \end{aligned}$$

Notice that  $\mathcal{M}'_\varphi(\{C\})$  is equivalent to  $\mathcal{M}_\varphi(\{C'\})$  where  $C'$  is the inverse constraint of constraint  $C$ , and so  $\mathcal{M}_\varphi(\Phi) \cap \mathcal{M}'_\varphi(\{C_0\}) = \mathcal{M}_\varphi(\Phi \cup \{C'_0\})$  given some method to invert constraints. Thus, the problem can also be expressed simply as

$$\mathcal{M}_\varphi(\Phi \cup \{C'_0\}) = \emptyset \quad \text{where } C'_0 = \text{inverse of } C_0$$

In Example 1, we show how a judgement can be verified manually using the predicate logic and model-theoretic interpretations of judgements provided above.

*Example 1.* Given index variables  $\varphi = \{i, j, k\}$  and constraints  $\Phi = \{C_1, C_2, C_3, C_4\}$  where

$$\begin{aligned} C_1 &= i \geq 4 \\ C_2 &= j \geq 2 \\ C_3 &= -k + 3 < 0 \\ C_4 &= i + j + k \leq 11 \end{aligned}$$

we want to check if  $\varphi; \Phi \models 2i + j^2 + 3k \geq 20$  always holds. Namely, we are interested in verifying whether the constraint  $2i + j^2 + 3k \geq 20$  imposes any additional

constraints to the index variables  $i$ ,  $j$  and  $k$  given the existing constraints  $C_1$ ,  $C_2$ ,  $C_3$  and  $C_4$ . In this case, we can notice that the minimum values of  $i$ ,  $j$  and  $k$  are 4, 2 and 4 respectively. As such, given these constraints, the minimum value  $2i + j^2 + 3k$  may evaluate to is  $2 \cdot 4 + 2^2 + 3 \cdot 4 = 24$ . As such, we can conclude that  $\varphi; \Phi \models 2i + j^2 + 3k \geq 20$  always holds.

We can also consider the predicate logic interpretation of the example. It suffices to only consider the index valuations that satisfy the conjunction of constraints, of which there are four. Here, we represent a valuation  $\rho$  as a set of pairs of the form  $\{(i, \rho(i)) \mid i \in \varphi\}$ , and so we have  $\{(i, 4), (j, 2), (k, 4)\}$ ,  $\{(i, 5), (j, 2), (k, 4)\}$ ,  $\{(i, 4), (j, 3), (k, 4)\}$  and  $\{(i, 4), (j, 2), (k, 5)\}$ . We can then verify the corresponding implications to show that the judgement holds

$$\begin{aligned} (4 \geq 4) \wedge (2 \geq 2) \wedge (-4 + 3 < 0) &\implies 4 + 2 + 4 \leq 11 \\ (5 \geq 4) \wedge (2 \geq 2) \wedge (-4 + 3 < 0) &\implies 5 + 2 + 4 \leq 11 \\ (4 \geq 4) \wedge (3 \geq 2) \wedge (-4 + 3 < 0) &\implies 4 + 3 + 4 \leq 11 \\ (4 \geq 4) \wedge (2 \geq 2) \wedge (-5 + 3 < 0) &\implies 4 + 2 + 5 \leq 11 \end{aligned}$$

Or correspondingly in model-theoretic notation

$$\begin{aligned} \mathcal{M}_\varphi(\Phi) &= \{\{(i, 4), (j, 2), (k, 4)\}, \{(i, 5), (j, 2), (k, 4)\}, \{(i, 4), (j, 3), (k, 4)\}, \{(i, 4), (j, 2), (k, 5)\}\} \\ \mathcal{M}_\varphi(\{2i + j^2 + 3k \geq 20\}) &= \{\{(i, n_1), (j, n_2), (k, n_3)\} \mid n_1, n_2, n_3 \in \mathbb{N}, 2n_1 + n_2^2 + 3n_3 \geq 20\} \\ \mathcal{M}_\varphi(\Phi) &\subseteq \mathcal{M}_\varphi(\{2i + j^2 + 3k \geq 20\}) \end{aligned}$$

We can also solve the inverse of the model-theoretic interpretation of the problem. Then we want to show that  $\mathcal{M}_\varphi(\Phi) \cap \mathcal{M}'_\varphi(\{2i + j^2 + 3k \geq 20\}) = \emptyset$  or equivalently  $\mathcal{M}_\varphi(\Phi \cup \{2i + j^2 + 3k < 20\}) = \emptyset$ .

$$\begin{aligned} \mathcal{M}_\varphi(\{2i + j^2 + 3k < 20\}) &= \{\{(i, n_1), (j, n_2), (k, n_3)\} \mid n_1, n_2, n_3 \in \mathbb{N}, 2n_1 + n_2^2 + 3n_3 < 20\} \\ \mathcal{M}_\varphi(\Phi) \cap \mathcal{M}'_\varphi(\{2i + j^2 + 3k \geq 20\}) &= \mathcal{M}_\varphi(\Phi \cup \{2i + j^2 + 3k < 20\}) = \emptyset \end{aligned}$$

### 3.3 Types and subtyping

~~We now introduce~~ the types from the type system of Baillot and Ghyselen. ~~The types~~ include a base type describing naturals as algebraic terms with sizes bounded by an interval consisting of two indices. This enables us to statically reason about how sizes of data structures change throughout reduction of processes, providing us termination guarantees for some forms of recursion. The type system of Baillot and Ghyselen contains lists as an additional base type, however for conciseness of the type system, we only consider naturals.

$$T, S ::= \text{Nat}[I, J] \mid \text{ch}_I^\sigma(\tilde{T}) \mid \forall_{I\tilde{I}}. \text{serv}_K^\sigma(\tilde{T})$$

We use input/output types for channels, and we further distinguish between channels that have replicated inputs, i.e. channels that have recursive behavior,

and those that do not. We refer to the former as *servers*, and we more specifically require all inputs on such channels to be replicated for technical convenience. Both servers and normal channels are annotated with an index  $I$  that for a normal channel represents the number of time steps remaining before the channel synchronizes, and for a server the remaining time before it becomes available. Note that this imposes a temporal linearity constraint onto normal channels, as such channels can synchronize at exactly one time step. For servers we have an additional index  $K$  that represents the parametric complexity of invoking the continuation of a replicated input on the server. Finally, the set  $\sigma \subseteq \{\text{in}, \text{out}\}$  is a subset of use-capabilities. Since types consist partly of indices, we define index substitution on types in Definition 9.

**Definition 9.** *We define index substitution on types by the following rules*

$$\begin{aligned} \text{Nat}[I, J] \{K/i\} &= \text{Nat}[I\{K/i\}, J\{K/i\}] \\ \text{ch}_I^\sigma(\tilde{T})\{J/i\} &= \text{ch}_{I\{J/i\}}^\sigma(\tilde{T}\{J/i\}) \\ \forall \tilde{i}. \text{serv}_K^\sigma(\tilde{T})\{J/j\} &= \forall_{I\{J/j\}} \tilde{i}. \text{serv}_K^\sigma(\tilde{T}) \text{ if } j \in \tilde{i} \\ \forall \tilde{i}. \text{serv}_K^\sigma(\tilde{T})\{J/j\} &= \forall_{I\{J/j\}} \tilde{i}. \text{serv}_{K\{J/j\}}^\sigma(\tilde{T}\{J/j\}) \text{ if } j \notin \tilde{i} \end{aligned}$$

Subtyping for base types and types is the least reflexive relation  $\sqsubseteq$  that satisfies the subtyping rules in Table 3. As the type system should provide upper bounds on the parallel complexity of processes, it is safe to weaken the bounds on the sizes of natural types. That is, we may decrease the lower bound and increase the upper bound on the sizes of such terms. For server and channel types, we may relax use-capabilities and use the subtyping relation on parameter types as well as modify the complexity bounds on servers, depending on the use-capabilities. Servers and channels of input/output capability are invariant, those of input capability are covariant and those of output capability are contravariant. That is, if a server or channel that inputs a value of type  $T$  is required, then we can safely use a server or channel that inputs a subtype of  $T$ , respectively. Conversely, when a server or channel of output capability is required, we can safely use a channel or server that outputs a supertype of the required parameter type [23]. This becomes apparent when we assume types **Integer** and **Real** such that **Integer**  $\sqsubseteq$  **Real**, as any process that receives reals can also safely receive integers, and any process that output reals can also safely output integers. Unlike Baillot and Ghyselen [3], we do not discard associations from our type contexts, rather we discard use-capabilities from channels and servers. Thus, to ensure the type checker is sound, we introduce rules (BGS-EMPTY) and (BGS-EMPTY) such that channel and server types are super types of ones with no use-capabilities.

### 3.4 Non-algorithmic type rules

We first consider the type rules for expressions, which are shown in Table 4. The zero term 0 intuitively receives the type  $\text{Nat}[0, 0]$  and a successor to a natural

(BGS-NWEAK) $\frac{\varphi; \Phi \models I' \leq I \quad \varphi; \Phi \models J \leq J'}{\varphi; \Phi \vdash \text{Nat}[I, J] \sqsubseteq \text{Nat}[I', J']}$	(BGS-CINVAR) $\frac{\varphi; \Phi \vdash \tilde{T} \sqsubseteq \tilde{S} \quad \varphi; \Phi \vdash \tilde{S} \sqsubseteq \tilde{T}}{\varphi; \Phi \vdash \text{ch}_I^{\{\text{in}, \text{out}\}}(\tilde{T}) \sqsubseteq \text{ch}_I^{\{\text{in}, \text{out}\}}(\tilde{S})}$
(BGS-CCOVAR) $\frac{\{\text{in}\} \subseteq \sigma \quad \varphi; \Phi \vdash \tilde{T} \sqsubseteq \tilde{S}}{\varphi; \Phi \vdash \text{ch}_I^\sigma(\tilde{T}) \sqsubseteq \text{ch}_I^{\{\text{in}\}}(\tilde{S})}$	(BGS-CCONTRA) $\frac{\{\text{out}\} \subseteq \sigma \quad \varphi; \Phi \vdash \tilde{S} \sqsubseteq \tilde{T}}{\varphi; \Phi \vdash \text{ch}_I^\sigma(\tilde{T}) \sqsubseteq \text{ch}_I^{\{\text{out}\}}(\tilde{S})}$
(BGS-SINVAR) $\frac{(\varphi, \tilde{i}); \Phi \vdash \tilde{T} \sqsubseteq \tilde{S} \quad (\varphi, \tilde{i}); \Phi \vdash \tilde{S} \sqsubseteq \tilde{T} \quad (\varphi, \tilde{i}); \Phi \models K = K'}{\varphi; \Phi \vdash \forall \tilde{i}. \text{serv}_K^{\{\text{in}, \text{out}\}}(\tilde{T}) \sqsubseteq \forall \tilde{i}. \text{serv}_{K'}^{\{\text{in}, \text{out}\}}(\tilde{S})}$	
(BGS-SCOVAR) $\frac{\{\text{in}\} \subseteq \sigma \quad (\varphi, \tilde{i}); \Phi \vdash \tilde{T} \sqsubseteq \tilde{S} \quad (\varphi, \tilde{i}); \Phi \models K' \leq K}{\varphi; \Phi \vdash \forall \tilde{i}. \text{serv}_K^\sigma(\tilde{T}) \sqsubseteq \forall \tilde{i}. \text{serv}_{K'}^{\{\text{in}\}}(\tilde{S})}$	
(BGS-SCONTRA) $\frac{\{\text{out}\} \subseteq \sigma \quad (\varphi, \tilde{i}); \Phi \vdash \tilde{S} \sqsubseteq \tilde{T} \quad (\varphi, \tilde{i}); \Phi \models K \leq K'}{\varphi; \Phi \vdash \forall \tilde{i}. \text{serv}_K^\sigma(\tilde{T}) \sqsubseteq \forall \tilde{i}. \text{serv}_{K'}^{\{\text{out}\}}(\tilde{S})}$	
(BGS-CEMPTY) $\frac{}{\varphi; \Phi \vdash \text{ch}_I^\sigma(\tilde{S}) \sqsubseteq \text{ch}_I^\emptyset(\tilde{T})}$	(BGS-SEMPY) $\frac{}{\varphi; \Phi \vdash \forall \tilde{i}. \text{serv}_K^\sigma(\tilde{S}) \sqsubseteq \forall \tilde{i}. \text{serv}_{K'}^\emptyset(\tilde{T})}$

**Table 3.** Rules for subtyping of base types and types.

term has the same type as its predecessor, but with 1 added to its lower and upper bounds. Finally, a variable receives a type if it is bound in the type context.

Before introducing the type rules for processes, we first introduce a function  $\downarrow_I^{\varphi; \Phi}(T)$  in Definition 10 that *advances* the time of type  $T$  by  $I$  units of time complexity. For a channel type  $\text{ch}_J^\sigma(\tilde{S})$ , we subtract  $I$  from  $J$  whenever we can guarantee that  $J \geq I$  under the constraints imposed on  $\varphi$  by  $\Phi$ . Otherwise, the advancement of  $I$  units of time complexity is undefined for type  $\text{ch}_J^\sigma(\tilde{S})$ , to ensure bounds on communication are not violated. For a server type  $\forall \tilde{i}. \text{serv}_K^\sigma(\tilde{S})$ , corresponding outputs are well-typed for any timestep  $I$  with  $I \geq J$ , and so a server simply loses input capability whenever we cannot guarantee that  $J \geq I$ . We extend advancement of time to contexts such that  $\downarrow_I^{\varphi; \Phi}(\Gamma)(v) = \downarrow_I^{\varphi; \Phi}(\Gamma(v))$ . When it is clear from context, we may omit  $\varphi$  and  $\Phi$ .

**Definition 10 (Advancement of Time).** *Let  $\varphi$  be a set of index variables,  $\Phi$  a set of constraints on indices,  $T$  a type and  $J$  an index. Then  $T$  after  $J$  units*

(BG-NZERO) $\frac{}{\varphi; \Phi; \Gamma \vdash 0 : \mathbf{Nat}[0, 0]}$	(BG-NSUCC) $\frac{\varphi; \Phi; \Gamma \vdash e : \mathbf{Nat}[I, J]}{\varphi; \Phi; \Gamma \vdash s(e) : \mathbf{Nat}[I + 1, J + 1]}$
(BG-SUB) $\frac{\varphi; \Phi; \Delta \vdash e : S \quad \varphi; \Phi \vdash \Gamma \sqsubseteq \Delta \quad \varphi; \Phi \vdash S \sqsubseteq T}{\varphi; \Phi; \Gamma \vdash e : T}$	(BG-VAR) $\frac{}{\varphi; \Phi; \Gamma, v : T \vdash v : T}$

**Table 4.** Type rules for expressions.

of time complexity,  $\downarrow_I^{\varphi; \Phi}(T)$ , is given by the rules below

$$\begin{aligned}
\downarrow_I^{\varphi; \Phi}(\mathbf{Nat}[I, J]) &= \mathbf{Nat}[I, J] \\
\downarrow_I^{\varphi; \Phi}(\mathbf{ch}_J^\sigma(\tilde{T})) &= \begin{cases} \mathbf{ch}_{J-I}^\sigma(\tilde{T}) & \text{if } \varphi; \Phi \models J \geq I \\ \mathbf{ch}_0^\emptyset(\tilde{T}) & \text{if } \varphi; \Phi \not\models J \geq I \end{cases} \\
\downarrow_I^{\varphi; \Phi}(\forall_{J-I} \tilde{i}. \mathbf{serv}_K^\sigma(\tilde{T})) &= \begin{cases} \forall_{J-I} \tilde{i}. \mathbf{serv}_K^\sigma(\tilde{T}) & \text{if } \varphi; \Phi \models J \geq I \\ \forall_{J-I} \tilde{i}. \mathbf{serv}_K^{\sigma \cap \{\mathbf{out}\}}(\tilde{T}) & \text{if } \varphi; \Phi \not\models J \geq I \end{cases}
\end{aligned}$$

**Definition 11 (Time invariance).** Let  $\Gamma$  be a type context. We say that  $\Gamma$  is time invariant if it only contains variables of either base types or server type with time 0 and use-capabilities  $\sigma$  such that  $\sigma \subseteq \{\mathbf{out}\}$ , i.e.  $\forall_0 \tilde{i}. \mathbf{serv}_K^\sigma(\tilde{T})$  for some index variables  $\tilde{i}$ , types  $\tilde{T}$  and index  $K$ .

We now present the type rules of the type system by Baillot and Ghyselen, adapted to fit our syntax. Type judgements are of the form  $\varphi; \Phi; \Gamma \vdash P \triangleleft K$ , which means that process  $P$  has complexity  $K$  given constraints  $\Phi$  with index variables in  $\varphi$  and given a type environment  $\Gamma$ . The type rules are defined in Table 5. Rule (BG-ISERV) handles replicated inputs and ensures that name  $a$  is bound to a server type with input capability in the type context. We must also make sure that in the continuation  $P$ , the type context must be time invariant as the replicated input may be invoked any number of times after  $I$  units of time have elapsed. Thus, only free naturals and servers with no input capability are safe. Rule (BS-ICH) is similar except we do not require the type context in the continuation to be time invariant as it is only used once. Rule (BG-OSERV) types output servers and most notably uses polymorphism in the index variables  $\tilde{i}$ . As such, when typing the expressions sent on the server, we must ensure that we can *instantiate* the index variables of the server using a substitution. Finally, type rule (BG-MATCH) shows how index constraints are introduced when typing processes by utilizing information gained from the two branches of the match expression.

We now show how a server calculating the  $n$ th digit of the Fibonacci sequence can be typed. Before presenting the process for the implementation of Fibonacci's



sequence, we first need to encode addition in the  $\pi$ -calculus, which we do using the *add* server as follows.

$$P_{\text{add}} \stackrel{\text{def}}{=} !\text{add}(x, y, r). \text{match } x \{0 \mapsto \bar{r}(y); s(z) \mapsto \overline{\text{add}}(z, s(y), r)\}$$

The *add* server needs three inputs  $x$ ,  $y$ , and  $r$ . The parameters  $x$  and  $y$  represent two naturals to be added, and  $r$  represents the channel intended for receiving the result. Note that no ticks are included in the server as we assume that addition can be done in constant time. The following process for calculating the  $n$ th number of the Fibonacci sequence is a naïve recursive implementation calculating  $\text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2)$ . The server takes two parameters  $n$  and  $r$  where  $n$  is the number of the Fibonacci sequence to calculate and  $r$  represents the channel intended for receiving the result.

$$\begin{aligned} P_{\text{fib}} \stackrel{\text{def}}{=} & !\text{fib}(n, r). \text{match } n \{0 \mapsto \bar{r}(0); s(n_1) \mapsto \\ & \text{match } n_1 \{0 \mapsto \bar{r}(s(0)); s(n_2) \mapsto \\ & (\nu r_1, r_2, r_3)(\overline{\text{fib}}(n_1, r_1) \mid \overline{\text{fib}}(n_2, r_2) \\ & \mid r_2(m_2).r_1(m_1).\text{tick}.\overline{\text{add}}(m_1, m_2, r_3) \mid r_3(m_3).\bar{r}(m_3))\}\} \end{aligned}$$

Finally we present a type context  $\Gamma$  under which the two servers *add* and *fib* are well-typed. Note that even though we use a naïve implementation of the Fibonacci sequence, we can still get a linear bound as the program runs in parallel.

$$\begin{aligned} \Gamma \stackrel{\text{def}}{=} & \text{add} : \forall_0 i, j, k. \text{serv}_0^{\{\text{in}, \text{out}\}}(\text{Nat}[0, i], \text{Nat}[j, k], \text{ch}_0(\text{Nat}[j, i + k])), \\ & \text{fib} : \forall_0 l. \text{serv}_l^{\{\text{in}, \text{out}\}}(\text{Nat}[0, l], \text{ch}_l(\text{Nat}[0, \text{fib}(l)])) \end{aligned}$$

### 3.5 Soundness

Baillot and Ghyselen [3] prove the soundness of their type system with respect to parallel complexity, using a subject reduction property and a safety property. The subject reduction property states that the reduction relation  $\Longrightarrow$  is type-preserving, and is proved by induction on the rules defining  $\Longrightarrow$ . The safety property states that the index assigned to a well-typed process is an upper bound on the parallel complexity of the process, which follows from subject reduction and the definition of local complexity.

**Theorem 1 (Subject reduction).** *If  $\varphi; \Phi; \Gamma \vdash P \triangleleft K$  and  $P \Rightarrow Q$  then  $\varphi; \Phi; \Gamma \vdash Q \triangleleft K$ .*

*Proof.* By induction on the reduction rules defining  $\Longrightarrow$  [3].

**Theorem 2 (Parallel complexity).** *Let  $P$  be an annotated process and  $m$  be its parallel complexity. Then, for a typing  $\varphi; \Phi; \Gamma \vdash P \triangleleft K$  we have that  $\varphi; \Phi \models K \geq m$ .*

*Proof.* Follows from subject reduction and the definition of local complexity [3].

(BG-ZERO) $\frac{}{\varphi; \Phi; \Gamma \vdash \mathbf{0} \triangleleft 0}$	(BG-SUBTYPE) $\frac{\varphi; \Phi; \Delta \vdash P \triangleleft K \quad \varphi; \Phi \vdash \Gamma \sqsubseteq \Delta \quad \varphi; \Phi \models K \leq K'}{\varphi; \Phi; \Gamma \vdash P \triangleleft K'}$
(BG-MATCH) $\frac{\varphi; \Phi; \Gamma \vdash e : \mathbf{Nat}[I, J] \quad \varphi; \Phi, I \leq 0; \Gamma \vdash P \triangleleft K \quad \varphi; \Phi, J \geq 1; \Gamma, x : \mathbf{Nat}[I-1, J-1] \vdash Q \triangleleft K}{\varphi; \Phi; \Gamma \vdash \mathbf{match} \ e \ \{0 \mapsto P; \ s(x) \mapsto Q\} \triangleleft K}$	
(BG-PAR) $\frac{\varphi; \Phi; \Gamma \vdash P \triangleleft K \quad \varphi; \Phi; \Gamma \vdash Q \triangleleft K}{\varphi; \Phi; \Gamma \vdash P \mid Q \triangleleft K}$	(BG-TICK) $\frac{\varphi; \Phi; \downarrow_1 \Gamma \vdash P \triangleleft K}{\varphi; \Phi; \Gamma \vdash \mathbf{tick}.P \triangleleft K + 1}$
(BG-ISERV) $\frac{\mathbf{in} \in \sigma \quad \varphi; \Phi \vdash \downarrow_I \Gamma, a : \forall_0 \tilde{i}. \mathbf{serv}_K^\sigma(\tilde{T}) \sqsubseteq \Gamma' \text{ and } \Gamma' \text{ time invariant} \quad \varphi, \tilde{i}; \Phi; \Gamma', \tilde{v} : \tilde{T} \vdash P \triangleleft K}{\varphi; \Phi; \Gamma, \Delta, a : \forall_I \tilde{i}. \mathbf{serv}_K^\sigma(\tilde{T}) \vdash !a(\tilde{v}).P \triangleleft I}$	
(BG-ICH) $\frac{\mathbf{in} \in \sigma \quad \varphi; \Phi; \downarrow_I \Gamma, \tilde{v} : \tilde{T}, a : \mathbf{ch}_0^\sigma(\tilde{T}) \vdash P \triangleleft K}{\varphi; \Phi; \Gamma, a : \mathbf{ch}_I^\sigma(\tilde{T}) \vdash a(\tilde{v}).P \triangleleft K + I}$	(BG-och) $\frac{\mathbf{out} \in \sigma \quad \varphi; \Phi; \downarrow_1 \Gamma \vdash \tilde{e} : \tilde{T}}{\varphi; \Phi; \Gamma, a : \mathbf{ch}_I^\sigma(\tilde{T}) \vdash \overline{a}(\tilde{e}) \triangleleft I}$
(BG-OSERV) $\frac{\mathbf{out} \in \sigma \quad \varphi; \Phi; \downarrow_I \Gamma \vdash \tilde{e} : \tilde{T}\{\tilde{J}/\tilde{i}\}}{\varphi; \Phi; \Gamma, a : \forall_I \tilde{i}. \mathbf{serv}_K^\sigma(\tilde{T}) \vdash \overline{a}(\tilde{e}) \triangleleft K\{\tilde{J}/\tilde{i}\} + I}$	(BG-NU) $\frac{\varphi; \Phi; \Gamma, a : T \vdash P \triangleleft K}{\varphi; \Phi; \Gamma \vdash (\nu a)P \triangleleft K}$

**Table 5.** Sized typing rules for parallel complexity of processes.

## 4 Type checking sized types for parallel complexity

As mentioned in Chapter ??, Baillot and Ghyselen [3] bound sizes of algebraic terms and synchronizations on channels using indices, leading to a partial order on indices. For instance, for a process of the form  $a(v).\bar{b}(v) \mid P$  (assuming synchronizations induce a cost in time complexity of one) we must enforce that the bound on  $a$  is strictly smaller than the bound on  $b$ . Thus, we must impose constraints on the interpretations of indices. Another concern in the typing of the process above is the parallel complexity. Granted separate bounds on the complexities of  $a(v).\bar{b}(v)$  and  $P$ , how do we establish a tight bound on their parallel composition? This turns out to be another major challenge, as bounds may be parametric, such that comparison of bounds is a partial order. Finally, for a process of the form  $!a(v).P \mid \overline{a}(e)$  we must *instantiate* the parametric complexity of  $!a(v).P$  based on the deducible size bounds of  $e$ , which quickly becomes difficult as indices become more complex.

The purpose of this section is to present a version of the type system by Baillot and Ghyselen that is algorithmic in the sense that its type rules can be easily implemented in a programming language, and so we must address the challenges described above. For the type checker, we assume we are given a set of constraints  $\Phi$  on index variables in  $\varphi$  and a type environment  $\Gamma$ . We first present the types of the type system as well as subtyping. Afterwards, we

present auxiliary functions and type rules. For the type rules we also present the concept of combined complexities that we use to bound parallel complexities by deferring comparisons of indices when these are not defined. We then prove the soundness of the type checker and show how it can be extended accompanied by examples. Finally, we show how we can verify the constraint judgements that show up in the type rules.

#### 4.1 Algorithmic type rules

When typing a process, we often need to find an index that is an upper bound on two other indices, for which there may be many options. To allow for the type checker to be as precise as possible, we want to find the minimum complexity that is a bound of two other complexities, which will depend on the representation of complexity, and as such, instead of representing complexity bounds using indices, we opt to use sets of indices which we refer to as *combined complexities*. Intuitively, given any point in the space spanned by some index variables, the combined complexity at that point is the maximum of the complexities making up the combined complexity at that point. This is illustrated in Figure 1 which shows a combined complexity consisting of three indices. The red dashed line represents the bound on the combined complexity. Representing complexities as sets of indices has the effect of *externalizing* the process of finding bounds of complexities by deferring this until a later time. We will later define the algorithm *basis* that removes superfluous indices of a combined complexity. In Figure 1 the index  $K$  is superfluous as it never contributes to the bound of the combined complexity.

**Fig. 1.** Combined complexities illustrated. The combined complexity consists of the three indices  $I, J, K$  of the single index variable  $i$ . The dashed red line shows the bound of the combined complexity.  $K$  is a superfluous index in the combined complexity as it never contributes to the bound of the combined complexity.

**Definition 12 (Combined complexity).** *We refer to a set  $\kappa$  of complexities as a combined complexity. We extend constraint judgements to include combined complexities such that*

1.  $\varphi; \Phi \models \kappa \leq \kappa'$  if for all  $K \in \kappa$  there exists  $K' \in \kappa'$  such that  $\varphi; \Phi \models K \leq K'$ .
2.  $\varphi; \Phi \models \kappa = \kappa'$  if  $\varphi; \Phi \models \kappa \leq \kappa'$  and  $\varphi; \Phi \models \kappa' \leq \kappa$ .
3.  $\kappa + I = \{K + I \mid K \in \kappa\}$ .
4.  $\kappa\{J/i\} = \{K\{J/i\} \mid K \in \kappa\}$ .

*In the above, we may substitute an index for a combined complexity. In such judgements, the index represents a singleton set. For instance,  $\varphi; \Phi \models \kappa \leq K$  represents  $\varphi; \Phi \models \kappa \leq \{K\}$ .*

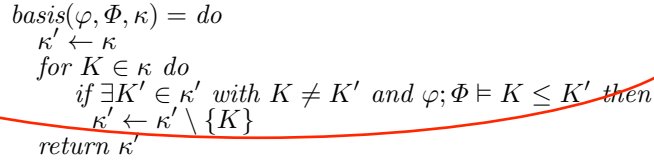
More specifically, when considering a combined complexity  $\kappa$ , we are interested in the maximal complexity given some valuation  $\rho$ , which we find by simply comparing the different values for the complexities within  $\kappa$  given  $\rho$ . Note that the complexity  $K \in \kappa$  that is maximal may be different for different valuations. In Definition 12 we extend the binary relations in  $\bowtie$  on indices to combined complexities, such that we can compare two combined complexities such as  $\varphi; \Phi \models \kappa \bowtie \kappa'$  and a combined complexity and complexity such as  $\varphi; \Phi \models \kappa \bowtie K$ . Definition 13 defines the function *basis* that discards any  $K \in \kappa$  that can never be the maximal complexity given some set of constraints  $\Phi$  (i.e. the complexities that are bounded by other complexities in the set), such that we can always keep the number of complexities in a combined complexity to a minimum.

**Definition 13.** We define the function *basis* that takes a set of index variables  $\varphi$ , a set of constraints  $\Phi$  and a combined complexity  $\kappa$ , and returns a new combined complexity without superfluous complexities (The basis of  $\kappa$ )

$$\text{basis}(\varphi, \Phi, \kappa) = \bigcap \{ \kappa' \subseteq \kappa \mid \forall K \in \kappa. \exists K' \in \kappa'. \varphi; \Phi \models K \leq K' \}$$

Moreover, the algorithm below computes the basis

Omit?



```

basis(φ, Φ, κ) = do
  κ' ← κ
  for K ∈ κ do
    if ∃K' ∈ κ' with K ≠ K' and φ; Φ ⊨ K ≤ K' then
      κ' ← κ' \ {K}
  return κ'

```

For typing expressions, we use the rules presented in Table 4, excluding the rule (BG-SUB). In Table 6 we show the type rules for processes. Type judgments are of the form  $\varphi; \Phi; \Gamma \vdash P \triangleleft \kappa$  where  $\kappa$  denotes the complexity of process  $P$ . The rule (S-TICK) types a **tick** prefix and incurs a cost of one in time complexity. We advance the time of all types in the context accordingly when typing the continuation. Rule (S-ANNOT) is similar but may incur a cost of  $n$ . Matches on naturals are typed with rule (S-NMATCH). Most notably, we extend the set of known constraints when typing the two continuations. That is, we can deduce constraints on the lower and upper bounds on the size of the expression we match on. For instance, for the zero pattern we can deduce that the lower bound  $I$  must be equal to 0 (or equivalently  $I \leq 0$ ), and for the successor pattern, we can guarantee that the upper bound  $J$  must be greater than or equal to 1. For the complexity of pattern matches and parallel composition, we take advantage of the fact that we represent complexities using combined complexities. As such, we include complexities in both  $P$  and  $Q$  in the result. To remove redundancy from the set  $\kappa \cup \kappa'$ , we use the basis function.

Rule (S-ISERV) types a replicated input on a name  $a$ , and so  $a$  must be bound to a server type with input capability. As the index  $I$  in the server type denotes the time steps remaining before the server is ready to synchronize, we

(S-NU) $\frac{\varphi; \Phi; \Gamma, a : T \vdash P \triangleleft \kappa}{\varphi; \Phi; \Gamma \vdash (\nu a : T)P \triangleleft \kappa}$	(S-PAR) $\frac{\varphi; \Phi; \Gamma \vdash P \triangleleft \kappa \quad \varphi; \Phi; \Gamma \vdash Q \triangleleft \kappa'}{\varphi; \Phi; \Gamma \vdash P \mid Q \triangleleft \text{basis}(\varphi, \Phi, \kappa \cup \kappa')}$
(S-TICK) $\frac{\varphi; \Phi; \downarrow_1 \Gamma \vdash P \triangleleft \kappa}{\varphi; \Phi; \Gamma \vdash \text{tick}.P \triangleleft \kappa + 1}$	(S-ANNOT) $\frac{\varphi; \Phi; \downarrow_n \Gamma \vdash P \triangleleft \kappa}{\varphi; \Phi; \Gamma \vdash n : P \triangleleft \kappa + n}$
(S-MATCH) $\frac{\varphi; \Phi; \Gamma \vdash e : \text{Nat}[I, J] \quad \varphi; \Phi, I \leq 0; \Gamma \vdash P \triangleleft \kappa \quad \varphi; \Phi, J \geq 1; \Gamma, x : \text{Nat}[I-1, J-1] \vdash Q \triangleleft \kappa'}{\varphi; \Phi; \Gamma \vdash \text{match } e \{0 \mapsto P; s(x) \mapsto Q\} \triangleleft \text{basis}(\varphi, \Phi, \kappa \cup \kappa')}$	
(S-ISERV) $\frac{\text{in} \in \sigma \quad \varphi; \Phi; \Gamma \vdash a : \forall_I i. \text{serv}_K^\sigma(\tilde{T}) \quad \varphi; \tilde{i}; \Phi; \text{ready}(\varphi, \Phi, \downarrow_I \Gamma), \tilde{v} : \tilde{T} \vdash P \triangleleft \kappa \quad \varphi; \tilde{i}; \Phi \models \kappa \leq K}{\varphi; \Phi; \Gamma \vdash !a(\tilde{v}).P \triangleleft \{I\}}$	(S-NIL) $\frac{}{\varphi; \Phi; \Gamma \vdash \mathbf{0} \triangleleft \{0\}}$
(S-OSERV) $\frac{\text{out} \in \sigma \quad \varphi; \Phi; \Gamma \vdash a : \forall_I i. \text{serv}_K^\sigma(\tilde{T}) \quad \varphi; \Phi; \downarrow_I \Gamma \vdash \tilde{e} : \tilde{S} \quad \text{instantiate}(\tilde{i}, \tilde{S}) = \{\tilde{J}/\tilde{i}\} \quad \varphi; \Phi \models \tilde{S} \sqsubseteq \tilde{T}}{\varphi; \Phi; \Gamma \vdash \overline{a}(\tilde{e}) \triangleleft \{K\{\tilde{J}/\tilde{i}\} + I\}}$	
(S-ICH) $\frac{\text{in} \in \sigma \quad \varphi; \Phi; \Gamma \vdash a : \text{ch}_I^\sigma(\tilde{T}) \quad \varphi; \Phi; \downarrow_I \Gamma, \tilde{v} : \tilde{T} \vdash P \triangleleft \kappa}{\varphi; \Phi; \Gamma \vdash a(\tilde{v}).P \triangleleft \kappa + I}$	(S-OCH) $\frac{\text{out} \in \sigma \quad \varphi; \Phi; \Gamma \vdash a : \text{ch}_I^\sigma(\tilde{T}) \quad \varphi; \Phi; \downarrow_I \Gamma \vdash \tilde{e} : \tilde{S} \quad \varphi; \Phi \models \tilde{S} \sqsubseteq \tilde{T}}{\varphi; \Phi; \Gamma \vdash \overline{a}(\tilde{e}) \triangleleft \{I\}}$

**Table 6.** Sized typing rules for parallel complexity of processes.

advance the time by  $I$  units of time complexity when typing the continuation  $P$ . To ensure that bounds on synchronizations in  $\downarrow_I^{\varphi; \Phi} \Gamma$  are not violated, we type  $P$  under the time invariant part of  $\downarrow_I^{\varphi; \Phi} \Gamma$ , i.e.  $\text{ready}(\varphi, \Phi, \downarrow_I \Gamma)$ . Note that the bound on the span of the replicated input is the bound on the time remaining before the server is ready to synchronize. As the replicated input may be invoked many times, the cost of invoking the server is accounted for in rule (S-OSERV) using the complexity bound  $K$  in the server type. Therefore, we enforce that  $K$  is in fact an upper bound on the span of the continuation  $P$ .

The rule (S-OSERV) types outputs on names bound to server types. Here, as stated above, we must account for the cost of invoking a server, and as a replicated input on a server is parametric, we must *instantiate* it based on the types of the expressions we are to output. Recall that in the type rule for outputs on servers from Chapter ??, this is to be done by finding a substitution that satisfies the premise  $\tilde{T} \sqsubseteq \tilde{S}\{\tilde{J}/\tilde{i}\}$ . However, this turns out to be a difficult problem, and we can in fact prove it NP-complete for types of polynomial indices even if we

disregard subtyping. However, note that it might not be necessary to use the full expressive power of polynomial indices, and so this may not necessarily affect type checking. Nevertheless, we over-approximate finding such a substitution, by using the function *instantiate*. That is, we *zip* together the index variables  $\tilde{i}$  with indices in types  $\tilde{T}$ . Remark that Baillot and Ghyselen [3] propose types for inference in their technical report, where the problem is simplified substantially, by forcing naturals to have lower bounds of 0 and upper bounds with exactly one index variable and a constant. Our approach admits more expressive lower bounds and multiplications, while imposing no direct restrictions on the number of index variables in an index, and is thus more suitable for a type-checker.

We now prove the NP-completeness of the smaller problem of checking whether there exists a substitution  $\{\tilde{J}/\tilde{i}\}$  that satisfies  $T = S\{\tilde{J}/\tilde{i}\}$  where  $T$  and  $S$  are types with polynomial indices. The main idea is a reduction proof from the NP-complete 3-SAT problem, i.e. the satisfiability problem of a boolean formula in conjunctive normal form with exactly three literals in each clause [14]. We first define a translation from a 3-SAT formula to a polynomial index in Definition 14. This is a polynomial time computable reduction, as we simply replace each logical-and with a multiplication, each logical-or with an addition and each negation with a subtraction from 1. In Lemma 3, we prove that the reduction is faithful with respect to satisfiability of a boolean formula. Finally, in Lemma 4, we prove that it is an NP-complete decision problem to verify the existence of a substitution that satisfies  $T = S\{\tilde{J}/\tilde{i}\}$  for types  $T$  and  $S$ .

**Definition 14 (3-SAT reduction).** We assume a one-to-one mapping  $f$  from unknowns to index variables. Let  $\phi$  be a 3-SAT formula

$$\phi = \bigwedge_{i=1}^n (\ell_{i1} \vee \ell_{i2} \vee \ell_{i3})$$

where  $\ell_{i1}$ ,  $\ell_{i2}$  and  $\ell_{i3}$  are of the forms  $x$  or  $\neg x$  for some variable  $x$ . We define a translation of  $\phi$  to a polynomial index

$$\llbracket \phi \rrbracket_{3\text{-SAT}} = \prod_{i=1}^n (\llbracket \ell_{i1} \rrbracket_{3\text{-SAT}} + \llbracket \ell_{i2} \rrbracket_{3\text{-SAT}} + \llbracket \ell_{i3} \rrbracket_{3\text{-SAT}})$$

where  $\llbracket x \rrbracket_{3\text{-SAT}} = f(x)$  and  $\llbracket \neg x \rrbracket_{3\text{-SAT}} = (1 - f(x))$ .

**Lemma 3.** Let  $\phi$  be a 3-SAT formula. Then  $\phi$  is satisfiable if and only if there exists a substitution  $\{\tilde{n}/\tilde{i}\}$  such that  $1 \leq \llbracket \phi \rrbracket_{3\text{-SAT}}\{\tilde{n}/\tilde{i}\}$ .

*Proof.* We consider the implications separately

1. Assume that  $\phi$  is satisfiable. Then there exists a truth assignment  $\tau$  such that each clause of  $\phi$  is true. Correspondingly, as  $\llbracket \phi \rrbracket_{3\text{-SAT}}$  is a product of non-negative factors, we for some substitution  $\{\tilde{n}/\tilde{i}\}$  have that  $1 \leq \llbracket \phi \rrbracket_{3\text{-SAT}}\{\tilde{n}/\tilde{i}\}$  if and only if each factor in the product is positive. We compare the conditions for a clause to be true in  $\phi$  to those for a corresponding factor in

Omit?

- $\llbracket \phi \rrbracket_{3\text{-SAT}}$  to be positive, and show that a substitution  $\{\tilde{n}/\tilde{i}\}$  exists such that  $1 \leq \llbracket \phi \rrbracket_{3\text{-SAT}}\{\tilde{n}/\tilde{i}\}$ . A clause in  $\phi$  is a disjunction of three literals of either the form  $x$  or  $\neg x$  for some unknown  $x$ . Thus, for a clause to be true, we must have at least one literal  $\tau(x) = tt$  or  $\neg\tau(x) = tt$  with  $\tau(x) = ff$ . The corresponding factor in  $\llbracket \phi \rrbracket_{3\text{-SAT}}$  is a sum of three terms of the forms  $f(x)$  or  $(1 - f(x))$  for some unknown  $x$ , where  $f$  is a one-to-one mapping from unknowns to index variables. Here, we utilize that in the type system by Baillot and Ghyselen [3], we have  $(1 - i\{\tilde{n}/\tilde{i}\}) = 0$  when  $i\{\tilde{n}/\tilde{i}\} \geq 1$  and  $(1 - i\{\tilde{n}/\tilde{i}\}) = 1$  when  $i\{\tilde{n}/\tilde{i}\} = 0$ . Thus, for a factor to be positive, it suffices that one term is positive, and so we can construct a substitution that guarantees this from the interpretation of  $\phi$ . That is, if  $\tau(x) = tt$ , we substitute 1 for  $f(x)$ , and if  $\tau(x) = ff$ , we substitute 0 for  $f(x)$ . Then, whenever a literal is true in  $\phi$ , the corresponding term in  $\llbracket \phi \rrbracket_{3\text{-SAT}}$  is positive, and so if  $\phi$  is satisfiable then there exists a substitution  $\{\tilde{n}/\tilde{i}\}$  such that  $1 \leq \llbracket \phi \rrbracket_{3\text{-SAT}}\{\tilde{n}/\tilde{i}\}$ .
2. Assume that there exists a substitution  $\{\tilde{n}/\tilde{i}\}$  such that  $1 \leq \llbracket \phi \rrbracket_{3\text{-SAT}}\{\tilde{n}/\tilde{i}\}$ . Then, as  $\llbracket \phi \rrbracket_{3\text{-SAT}}$  is a product of non-negative factors, each factor must be positive. Correspondingly, if  $\Phi$  is satisfiable, then there exists a truth assignment such that each clause of  $\phi$  is true. We compare the conditions for a factor in  $\llbracket \phi \rrbracket_{3\text{-SAT}}\{\tilde{n}/\tilde{i}\}$  to be positive to those for a corresponding clause in  $\phi$  to be true, and show that  $\phi$  is satisfiable. A factor in  $\llbracket \phi \rrbracket_{3\text{-SAT}}$  is a sum of at most three terms of the forms  $f(x)\{\tilde{n}/\tilde{i}\}$  or  $(1 - f(x)\{\tilde{n}/\tilde{i}\})$ . Here we again utilize that in the type system by Baillot and Ghyselen [3], we have  $(1 - f(x)\{\tilde{n}/\tilde{i}\}) = 0$  when  $f(x)\{\tilde{n}/\tilde{i}\} \geq 1$  and  $(1 - f(x)\{\tilde{n}/\tilde{i}\}) = 1$  when  $f(x)\{\tilde{n}/\tilde{i}\} = 0$ , and so it must be that in the factor, we have at least one term  $f(x)\{\tilde{n}/\tilde{i}\} \geq 1$  or  $(1 - f(x)\{\tilde{n}/\tilde{i}\}) \geq 1$ . Correspondingly, for the clause in  $\phi$  to be true, at least one literal must be true. We show that there exists a truth assignment  $\tau$  such that if a term in  $\llbracket \phi \rrbracket_{3\text{-SAT}}$  is positive, then the corresponding literal in  $\phi$  is true. If  $f(x)\{\tilde{n}/\tilde{i}\} \geq 1$  then we set  $\tau(x) = tt$ , and if  $f(x)\{\tilde{n}/\tilde{i}\} = 0$  we set  $\tau(x) = ff$ , as  $\llbracket x \rrbracket_{3\text{-SAT}}\{\tilde{n}/\tilde{i}\} \geq 1$  when  $f(x)\{\tilde{n}/\tilde{i}\} \geq 1$  and  $\llbracket \neg x \rrbracket_{3\text{-SAT}} \geq 1$  when  $f(x)\{\tilde{n}/\tilde{i}\} = 0$ . Then, whenever a term is positive in  $\llbracket \phi \rrbracket_{3\text{-SAT}}\{\tilde{n}/\tilde{i}\}$ , the corresponding literal in  $\phi$  is true, and so if there exists a substitution  $\{\tilde{n}/\tilde{i}\}$  such that  $1 \leq \llbracket \phi \rrbracket_{3\text{-SAT}}\{\tilde{n}/\tilde{i}\}$ , then  $\phi$  is satisfiable.

**Lemma 4.** Let  $T$  and  $S$  be types with polynomial indices. Then checking whether there exists a substitution  $\{\tilde{J}/\tilde{i}\}$  such that  $T = S\{\tilde{J}/\tilde{i}\}$  is an NP-complete problem.

*Proof.* By reduction from the 3-SAT problem. Assume that we have some algorithm that can verify the existence of a substitution  $\{\tilde{J}/\tilde{i}\}$  such that  $T = S\{\tilde{J}/\tilde{i}\}$ , and let  $\phi$  be a 3-SAT formula. Then using the algorithm, we can check whether  $\phi$  is satisfiable by verifying whether there exists  $\{\tilde{J}/\tilde{i}\}$  such that the following holds

$$\text{Nat}[0, 1] = \text{Nat}[0, (1 - (1 - \llbracket \phi \rrbracket_{3\text{-SAT}}))\{\tilde{J}/\tilde{i}\}]$$

That is,  $1 = (1 - (1 - \llbracket \phi \rrbracket_{3\text{-SAT}}\{\tilde{J}/\tilde{i}\}))$  implies  $1 \leq \llbracket \phi \rrbracket_{3\text{-SAT}}\{\tilde{J}/\tilde{i}\}$ , as  $(1 - \llbracket \phi \rrbracket_{3\text{-SAT}}\{\tilde{J}/\tilde{i}\}) = 0$  when  $\llbracket \phi \rrbracket_{3\text{-SAT}}\{\tilde{J}/\tilde{i}\} \geq 1$  and  $(1 - \llbracket \phi \rrbracket_{3\text{-SAT}}\{\tilde{J}/\tilde{i}\}) = 1$  when

$\llbracket \phi \rrbracket_{3\text{-SAT}}\{\tilde{J}/\tilde{i}\} = 0$ . Furthermore, for  $1 \leq \llbracket \phi \rrbracket_{3\text{-SAT}}\{\tilde{J}/\tilde{i}\}$  to hold, the indices in the sequence  $\tilde{J}$  cannot contain index variables, and so there must exist an equivalent substitution of naturals for index variables  $\{\tilde{n}/\tilde{i}\}$ . Then, by Lemma 3 we have that  $\phi$  is satisfiable if and only if there exists a substitution  $\{\tilde{n}/\tilde{i}\}$  such that  $1 \leq \llbracket \phi \rrbracket_{3\text{-SAT}}\{\tilde{n}/\tilde{i}\}$ . Thus, as 3-SAT is an NP-complete problem, the reduction from 3-SAT is computable in polynomial time and as polynomial reduction is a transitive relation, i.e. any NP-problem is polynomial time reducible to verifying the existence of a substitution  $\{\tilde{J}/\tilde{i}\}$  that satisfies the equation  $T = S\{\tilde{J}/\tilde{i}\}$ , it follows that the problem is NP-hard. To show that it is an NP-complete problem, we show that a certificate can be verified in polynomial time. That is, given some substitution  $\{\tilde{J}/\tilde{i}\}$ , we can in linear time check whether  $T = S\{\tilde{J}/\tilde{i}\}$  by substituting indices  $\tilde{J}$  for indices  $\tilde{i}$  in type  $S$  and by then comparing the two types.

In Example 2, we show how a process implementing addition of naturals can be typed using our type rules, yielding a precise bound on the parallel complexity.

*Example 2.* As an example of a process that is typable using our type rules, we show how the addition operator for naturals can be written as a process and subsequently be typed. We use a server to encode the addition operator

$$!\text{add}(x, y, r).\text{match } x \{0 \mapsto \overline{r}(y); s(z) \mapsto \text{tick}.\overline{\text{add}}(z, s(y), r)\}$$

such that channel  $r$  is used to output the addition of naturals  $x$  and  $y$ . To type the process, we use the following contexts and set of index variables

$$\Gamma \stackrel{\text{def}}{=} \text{add} : \forall_0 i, j, k, l, m, n, o. \text{serv}_j^{\{\text{in}, \text{out}\}}(\text{Nat}[0, j], \text{Nat}[0, l], \text{ch}_j^{\{\text{out}\}}(\text{Nat}[0, j + l]))$$

$$\Delta \stackrel{\text{def}}{=} \text{ready}(\cdot, \cdot, \Gamma), x : \text{Nat}[0, j], y : \text{Nat}[0, l], r : \text{ch}_j^{\{\text{out}\}}(\text{Nat}[0, j + l])$$

$$\varphi \stackrel{\text{def}}{=} \{i, j, k, l, m, n, o\}$$

We now derive a type for the encoding of the addition operator, yielding a precise bound of  $j$ , corresponding to an upper bound on the size of  $x$ , as we pattern match at most  $j$  times on natural  $x$ . Notably we have that  $\text{instantiate}((i, j, k, l, m, n, o), \text{Nat}[0, j-1], \text{Nat}[1, l+1], \text{ch}_j^{\{\text{out}\}}(\text{Nat}[0, j + l])) = \{0/i, j-1/j, 0/k, l+1/l, j/m, 0/n, j+l/o\}$ .

## 4.2 Soundness

In this section, we prove the soundness of the type system presented in Section 4.1. That is, we prove a subject reduction result and that a typing of a process provides a bound on the parallel complexity of the process. We first augment the definition of annotated processes and extend the type rules.

## 4.3 Augmented language and type system

In the type system from Chapter ??, subtyping rules for processes and expressions are used extensively to bound parallel compositions and to prove a subject reduction property. However, such use of subtyping is notoriously difficult



to implement, as there may be an infinite number of subtypes. Consider for instance a parallel composition of two processes  $P$  and  $Q$  that are typed as  $\varphi; \Phi; \Gamma \vdash P \triangleleft 10 - i$  and  $\varphi; \Phi; \Gamma \vdash Q \triangleleft i^2 - 100i$ , respectively. These complexities are both functions of  $i$  and clearly intersect. It is not trivial to find a common subtype of these indices in practice, and for more complex indices, it quickly becomes intractable.

In the case of subject reduction, where we have a reduction  $P \Longrightarrow Q$ , Baillot and Ghyselen [3] use a subtyping rule for expressions extensively. In particular, upon synchronization of an input and an output on some channel, we require that the expressions to be transmitted can be typed as super types of the message types of the channel. Thus, upon reduction we use the subtyping rule to assign the message types to the expressions. In this case, we use information from the typing of  $P$  to *guide* subtyping in the typing of  $Q$ , and so by augmenting the reduction relation with annotations, we can implement subtyping. An alternative is to permit the subtyping rule for expressions to prove the soundness of the type system, and then omit it from the implementation, i.e. we simply reject processes that cannot be typed without using the subtyping rule. This is a valid compromise, as it avoids needless cluttering of the reduction relation with annotations, and we are usually not interested in using the subject reduction property in practice. That is, we need only type a process prior to any reduction, presuming the type system is sound. Therefore, we take this approach, introducing the subtyping rule for expressions below.

$$(S\text{-SUBSUMPTION}) \frac{\varphi; \Phi; \Gamma \vdash e : S \quad \varphi; \Phi \vdash S \sqsubseteq T}{\varphi; \Phi; \Gamma \vdash e : T}$$

As reduction may reduce ticks, subsequently modifying the typing of a process, we need a *delaying* property to prove a subject reduction result. That is, it must be safe to increase the bound in a typing, so long as all bounds on synchronizations in the type context are correspondingly increased (We write this as  $\uparrow^I \Gamma$  when we increase bounds by  $I$ , as defined in Definition 15). Such a property is proved by induction on the type rules.

**Definition 15 (Delaying).** *We define the delaying  $\uparrow^I T$  of a type  $T$  by an index  $I$  by the rules*

$$\begin{aligned} \uparrow^L \text{Nat}[I, J] &= \text{Nat}[I, J] \\ \uparrow^L \text{ch}_I^\sigma(\tilde{T}) &= \text{ch}_{I+L}^\sigma(\tilde{T}) \\ \uparrow^L \forall_{\tilde{I}} \tilde{i}. \text{serv}_K^\sigma(\tilde{T}) &= \forall_{\tilde{I}+L} \tilde{i}. \text{serv}_K^\sigma(\tilde{T}) \end{aligned}$$

*We extend delaying to contexts such that for all  $v \in \text{dom}(\Gamma)$  such that  $\Gamma(v) = T$  we have that  $(\uparrow^I \Gamma)(v) = \uparrow^I T$ .*

However, this turns out to be problematic due to type annotations on restrictions. That is, in the clause for restriction, we can assume that  $\varphi; \Phi; \Gamma \vdash (\nu a :$

$T)P \triangleleft \kappa$  from which we also have  $\varphi; \Phi; \Gamma, a : T \vdash P \triangleleft \kappa$ , and by induction we obtain  $\varphi; \Phi; \uparrow^I \Gamma, a : \uparrow^I T \vdash P \triangleleft \kappa + I$ , which unfortunately does not comply with the type annotation in  $(\nu a : T)P$ . Without a delaying property, we are (to our knowledge) unable to preserve a typing upon reduction, i.e. either the needed type context or assignable combined complexity (possibly both) changes. Unfortunately, this is an insufficient condition to prove that a bound on an assigned combined complexity  $\kappa$  is in fact a bound on the span, as constant changes to expressions  $I$  or  $J$  in a judgement  $\varphi; \Phi; \Gamma \vdash I \bowtie J$  is enough to sometimes (even for linear functions) fail a judgement. An example of this is depicted in Figure 2, showcasing that a strong subject reduction property is crucial to prove soundness.

**Fig. 2.** The image on the left depicts two linear indices  $I = 5i + 6$  and  $J = 3i + 5$  as functions of  $i$  that do not intersect in the non-negative valuation space of  $i$ . On the right, we subtract 3 from  $I$ , shifting the intersection of  $I$  and  $J$  into the non-negative valuation space. Thus,  $\{i\}; \cdot \models J \leq I$  but  $\{i\}; \cdot \not\models J \leq I - 3$ .

As we are not interested in actually using a subject reduction property in practice, we can introduce two versions of the implementation: One with type annotations on restrictions, and one without. That is, one without a subject reduction property, and one with a subject reduction property. Intuitively, if  $\varphi; \Phi; \Gamma \vdash P \triangleleft \kappa$  by the type system with annotations, then also  $\varphi; \Phi; \Gamma \vdash P \triangleleft \kappa$  by the type system without annotations, and so it suffices to prove soundness for the type system without type annotations, to show that any bound on  $\kappa$  is an upper bound on the span. Thus, in the remainder of this section, we assume that restrictions do not have type annotations.

#### ~~4.4 Intermediary lemmas~~

~~We are now ready to present the soundness results. We first prove some intermediary lemmas that we use for the main results. We first prove some useful properties with respect to delaying in Lemma 5. We use the first point in the clause of (SC-ARES) in the proof of subject congruence. We use the second and third points for the proof of subject reduction, namely for synchronizations on channels, where we preserve the maximal local complexity amongst an input and output, and so we may need to *delay* the type context.~~

#### ~~Lemma 5 (Delaying).~~

- ~~1.  $\downarrow_I^{\varphi; \Phi}(\uparrow^I T) = T$ .~~
- ~~2. If  $\varphi; \Phi; \Gamma \vdash e : T$  then  $\varphi; \Phi; \uparrow^I \Gamma \vdash e : \uparrow^I T$ .~~
- ~~3. If  $\varphi; \Phi; \Gamma \vdash P \triangleleft \kappa$  then  $\varphi; \Phi; \uparrow^I \Gamma \vdash P \triangleleft \kappa + I$ .~~

~~Proof. Point 1 is straightforward. Point 2 and 3 are proved by induction on the type rules of expressions and processes, respectively.~~

~~We next prove that advancement of time is additive. This result is integral to subject congruence, namely for (SC SUM) that allows us to sum two annotations, and correspondingly split one annotation into two.~~

~~**Lemma 6 (Additive advancement of time).** Let  $\Phi$  be a set of constraints with unknowns in  $\varphi$  and let  $T$  be a type then  $\downarrow_I^{\varphi;\Phi}(\downarrow_J^{\varphi;\Phi}(T)) = \downarrow_{I+J}^{\varphi;\Phi}(T)$ .~~

~~*Proof.* On the structure of  $T$ . The proof is shown in Appendix ??.~~

~~We now prove the usual weakening and strengthening lemmas in Lemma 7 and Lemma 8, respectively. In this work, we can weaken and strengthen sets of constraints and type contexts. That is, we can safely introduce new constraints or type associations, and discard constraints that are covered by the remaining constraints, as well as type associations for variables that are not free in a corresponding expression or process. As a consequence of type rule (S SUBSUMPTION), it is also safe to use subtyping on contexts.~~

~~**Lemma 7 (Weakening).** Let  $F$  and  $F'$  be disjoint contexts. Then~~

- ~~1. If  $\varphi, \Phi, F \vdash e : T$  then  $\varphi, \varphi', \Phi, \Phi', F, F' \vdash e : T$ .~~
- ~~2. If  $\varphi, \Phi, F \vdash P \triangleleft \kappa$  then  $\varphi, \varphi', \Phi, \Phi', F, F' \vdash P \triangleleft \kappa$ .~~
- ~~3. If  $\varphi, \Phi, F \vdash e : T$  and  $\varphi, \Phi \vdash \Delta \sqsubseteq F$  then  $\varphi, \Phi, \Delta \vdash e : T$ .~~
- ~~4. If  $\varphi, \Phi, F \vdash P \triangleleft \kappa$  and  $\varphi, \Phi \vdash \Delta \sqsubseteq F$  then  $\varphi, \Phi, \Delta \vdash P \triangleleft \kappa$ .~~

~~*Proof.* Point 1 and point 2 are proved by induction on the type rules of expressions and processes, respectively. Point 3 is proved by induction on the type rules of expressions, and point 4 follows from point 3.~~

~~**Lemma 8 (Strengthening).** Let  $\Phi$  and  $\Phi'$  be sets of constraints on  $\varphi$  such that  $\varphi, \Phi \vdash C$  for all  $C \in \Phi'$ . Then~~

- ~~1. If  $\varphi, (\Phi, \Phi') \vdash C'$  then also  $\varphi, \Phi \vdash C'$ .~~
- ~~2. If  $\varphi, (\Phi, \Phi') \vdash S \sqsubseteq T$  then also  $\varphi, \Phi \vdash S \sqsubseteq T$ .~~
- ~~3. If  $\downarrow_I^{\varphi;(\Phi, \Phi')}(T)$  then also  $\downarrow_I^{\varphi;\Phi}(T)$ .~~
- ~~4. If  $\varphi, \Phi, \Phi', F, F' \vdash e : T$  and the names in  $F'$  are not free in  $e$  then  $\varphi, \Phi, F \vdash e : T$ .~~
- ~~5. If  $\varphi, \Phi, \Phi', F, F' \vdash P \triangleleft \kappa$  and the names in  $F'$  are not free in  $P$  then  $\varphi, \Phi, F \vdash P \triangleleft \kappa$ .~~

~~*Proof.* Point 1 follows directly from  $\varphi, \Phi \vdash C$  for all  $C \in \Phi'$ , i.e.  $\Phi'$  imposes no further constraints on  $\varphi$  given  $\Phi$ . Point 2 is proved by induction on the subtyping rules. Point 4 is proved by induction on the type rules of expressions using point 2. Point 5 is proved by induction on the type rules of processes using point 3 and point 4.~~

~~In the case of synchronization of a replicated input and an output on a server, we *instantiate* the continuation of the input, by substituting expressions of the output for variables bound in the input. Similarly, the message types of the server are subject to index substitution, as per type rule (S OSERV). Thus, to prove~~

a subject reduction property, we need to show that a well typed process is also well typed subject to index substitution. We prove this in Lemma 9. Notably, the inverse of point 2 does not hold, as a judgement that does not hold prior to index substitution may be satisfied after. Consider for instance the judgement  $\{i, j\}; \not\vdash i \leq j$  that clearly fails, as we do not constraint valuations of  $i$  and  $j$ . However, the judgement  $\{i\}; \vdash i \leq j\{j/i\}$  is satisfied by the reflexive property of  $\leq$ . This explains the need for a weaker result with respect to point 6. That is, we may use type rule (S PAR) or (S MATCH) in the typing of a process  $P$ , such that the combined complexity is derived using the basis function, where complexity  $K$  is discarded if a judgement of the form  $\varphi; \Phi \vdash K \leq K'$  holds. Thus, after index substitution, we may be able to discard further complexity bounds.

**Lemma 9 (Index substitution).** *Let  $\varphi$  be a set of index variables with  $i \notin \varphi$ , and let  $J$  be an index with unknowns in  $\varphi$ . Then*

1.  $\llbracket I\{J/i\} \rrbracket_\rho = \llbracket I \rrbracket_{\rho[i \mapsto \llbracket J \rrbracket_\rho]}$ .
2. If  $(\varphi, i); \Phi \vdash C$  then  $\varphi; \Phi\{J/i\} \vdash C\{J/i\}$ .
3. If  $(\varphi, i); \Phi \vdash T \sqsubseteq S$  then  $\varphi; \Phi\{J/i\} \vdash T\{J/i\} \sqsubseteq S\{J/i\}$ .
4. If  $(\varphi, i); \Phi; F \vdash e : T$  then  $\varphi; \Phi\{J/i\}; F\{J/i\} \vdash e : T\{J/i\}$ .
5.  $\varphi; \Phi\{J/i\} \vdash \downarrow_{I\{J/i\}}^{\varphi; \Phi\{J/i\}}(T\{J/i\}) \sqsubseteq \downarrow_I^{(\varphi, i); \Phi}(T)\{J/i\}$ .
6. If  $(\varphi, i); \Phi; F \vdash P \triangleleft_\kappa$  then  $\varphi; \Phi\{J/i\}; F\{J/i\} \vdash P \triangleleft_{\kappa'\{J/i\}}$  with  $\varphi; \Phi\{J/i\} \vdash \kappa\{J/i\} = \kappa'\{J/i\}$ .

*Proof.* Point 1 is proved by induction on  $I$  using the definition of interpretations of indices. Point 2 is a direct consequence of point 1. Point 3 and 4 are proved by induction on the subtyping rules and type rules for expressions, respectively, using point 2. We use point 3 to prove point 4. Point 5 is useful for point 6 and follows from point 2. Point 6 is proved by induction on the type rules of processes, utilizing point 3, 4 and 5.

We next prove some properties of the basis function in Lemma 10. Point 1 is used in the proof of subject congruence, where associativity and commutativity of parallel compositions can affect how the basis function is applied. Similarly, point 2 is useful for the distributive property of annotations on parallel compositions, as this affects when in the typing the basis function is applied. Point 1 follows from the transitive property of  $\leq$ , and point 2 follows directly from  $\varphi; \Phi \vdash K + I \leq K' + I$  if and only if  $\varphi; \Phi \vdash K \leq K'$ . Point 3 is a soundness result for the basis function. We essentially prove that the function does not increase or decrease the complexity bound imposed by a combined complexity.

**Lemma 10.** *Let  $\kappa$  and  $\kappa'$  be combined complexities with unknowns in  $\varphi$ , and let  $\Phi$  be a set of constraints on  $\varphi$ . Then*

1.  $\varphi; \Phi \vdash \text{basis}(\varphi, \Phi, \kappa \cup \text{basis}(\varphi, \Phi, \kappa')) = \text{basis}(\varphi, \Phi, \kappa \cup \kappa')$ .
2. If  $I$  has all unknowns in  $\varphi$  then  $\text{basis}(\varphi, \Phi, \kappa) + I = \text{basis}(\varphi, \Phi, \kappa + I)$ .
3.  $\varphi; \Phi \vdash \text{basis}(\varphi, \Phi, \kappa) = \kappa$ .

~~Proof. Point 1 and 2 are straightforward, we use them for subject reduction. Point 3 follows from the fact that  $K \in \kappa$  and  $K \notin \text{basis}(\varphi, \Phi, \kappa)$  imply  $K' \in \text{basis}(\varphi, \Phi, \kappa)$  with  $\varphi, \Phi \vdash K \leq K'$ .~~

~~Finally, in Lemma 11, we prove some properties of typings of expressions that are useful for the usual substitution lemma. In particular, we show that if an expression is well typed, then we can safely advance the time or apply the ready function to the context the expression is typed under.~~

**Lemma 11.**

- ~~1. If  $\varphi; \Phi; \Gamma \vdash e : T$  then  $\varphi; \Phi; \downarrow_I \Gamma \vdash e : \downarrow_k^{\varphi; \Phi}(T)$ .~~
- ~~2. If  $\varphi; \Phi; \Gamma \vdash e : T$  then  $\varphi; \Phi; \text{ready}(\varphi, \Phi, \Gamma) \vdash e : \text{ready}(\varphi, \Phi, T)$ .~~

~~Proof. By induction on the type rules of expressions. The proof is straightforward.~~

#### 4.5 Subject reduction

We are now ready to prove a subject reduction property. We first state and prove the usual substitution (Lemma 12) and subject congruence (Lemma 13) properties. Our subject congruence result appears slightly weaker than usual, i.e. we are not guaranteed the exact same typing before and after application of structural congruence. This is because the congruence rules dictate when the basis function is applied, and so provided two equivalent yet different complexities, the congruence rules may affect which of the two we discard. However, this does not affect the complexity bounds imposed onto a process by a typing.

**Lemma 12 (Substitution).**

1. If  $\varphi; \Phi; \Gamma, v : T \vdash e' : S$  and  $\varphi; \Phi; \Gamma \vdash e : T$  then  $\varphi; \Phi; \Gamma \vdash e'[v \mapsto e] : S$ .
2. If  $\varphi; \Phi; \Gamma, v : T \vdash P \triangleleft \kappa$  and  $\varphi; \Phi; \Gamma \vdash e : T$  then  $\varphi; \Phi; \Gamma \vdash P[v \mapsto e] \triangleleft \kappa$ .

*Proof. The first point is proved by induction on the type rules of expressions, and the second by induction on the type rules for processes. ~~The proof is shown in Appendix ??.~~*

**Lemma 13 (Subject congruence).** Let  $P$  and  $Q$  be processes such that  $P \equiv Q$  then  $\varphi; \Phi; \Gamma \vdash P \triangleleft \kappa$  if and only if  $\varphi; \Phi; \Gamma \vdash Q \triangleleft \kappa'$  with  $\varphi; \Phi \models \kappa = \kappa'$ .

*Proof. By induction on the rules defining  $\equiv$ . ~~The proof is shown in Appendix ??.~~*

As we do not have a subtyping rule for processes (As such a rule is not algorithmic), we cannot guarantee that typing is invariant to reduction, without cluttering the semantics with annotations. A weaker yet sufficient property is that the complexity bound of a process is decreasing subject to reduction. We prove this result in Theorem 3 by induction on the parallel reduction relation  $\Rightarrow$ .

**Theorem 3 (Subject reduction).** If  $\varphi; \Phi; \Gamma \vdash P \triangleleft \kappa$  and  $P \Longrightarrow Q$  then  $\varphi; \Phi; \Gamma \vdash Q \triangleleft \kappa'$  with  $\varphi; \Phi \models \kappa' \leq \kappa$ .

*Proof.* By induction on the rules defining  $\Longrightarrow$ .

**(PR-rep)** We have the reduction  $n \cdot !a(\tilde{v}) P \mid m \cdot \bar{a}(\tilde{e}) \Longrightarrow n \cdot !a(\tilde{v}) P \mid \text{mar}(n, m) \cdot P[\tilde{v} \mapsto \tilde{e}]$ . We type the process with (SA-PAR) with the two following type derivations for the two subprocesses. Using (SA-PAR), the whole process receives the bound basis  $(\varphi, \Phi, [I] \cup \{K\{\tilde{J}/\tilde{i}\} + (I - m) + m\})$ . We can reuse the first derivation tree, and so it suffices to show that  $\varphi; \Phi; \Gamma \vdash \text{mar}(n, m) \cdot P[\tilde{v} \mapsto \tilde{e}] \triangleleft \kappa'$  with  $\varphi; \Phi \models \kappa' \leq K\{\tilde{J}/\tilde{i}\} + I$ . Whereas we can deduce for the first derivation tree that  $\varphi; \Phi \vdash n \leq I$ , from the fact that channel  $a$  must have input capability upon typing the replicated input, we do not have the same guarantees for outputs on servers. That is, a server cannot lose its output capability, and so for the second derivation tree we have two cases

1.  $\varphi; \Phi \models m \leq I$  and so  $\varphi; \Phi \models (I - m) + m = I$ , such that  $\varphi; \Phi; \downarrow_I \Gamma \vdash \tilde{e} : \tilde{S}$ . By (S-SUBSUMPTION) we then obtain  $\varphi; \Phi; \downarrow_I \Gamma \vdash \tilde{e} : \tilde{T}$  from  $\varphi; \Phi \models \tilde{S} \sqsubseteq \tilde{T}$ , and by weakening (Lemma 7) we have that  $\varphi; \Phi; \downarrow_I \Gamma \vdash \tilde{e} : \tilde{T}$ . As  $\text{ready}(\varphi, \Phi, \downarrow_I \Gamma)$  only discards use capabilities from  $\downarrow_I \Gamma$ , we derive  $(\varphi, \tilde{i}); \Phi; \downarrow_I \Gamma, \tilde{v} : \tilde{T} \vdash P \triangleleft \kappa$  by weakening (Lemma 7). By substitution (Lemma 12), we thus obtain  $(\varphi, \tilde{i}); \Phi; \downarrow_I \Gamma \vdash P[\tilde{v} \mapsto \tilde{e}] \triangleleft \kappa$ . Then, using the fact that  $\tilde{i}$  are not free in  $\Gamma$ ,  $\forall \tilde{i}. \text{serv}_K^G$  and  $\Phi$ , i.e.  $\Gamma\{\tilde{J}/\tilde{i}\} = \Gamma$  and  $\Phi\{\tilde{J}/\tilde{i}\} = \Phi$ , we derive from index substitution (Lemma 9) that  $\varphi; \Phi; \downarrow_I \Gamma \vdash P[\tilde{v} \mapsto \tilde{e}] \triangleleft \kappa'\{\tilde{J}/\tilde{i}\}$  with  $\varphi; \Phi \models \kappa'\{\tilde{J}/\tilde{i}\} = \kappa'\{\tilde{J}/\tilde{i}\}$ . Using the fact that  $\varphi; \Phi \models \text{mar}(n, m) \leq I$ , we obtain  $\varphi; \Phi; \downarrow_{\text{mar}(n, m)} \Gamma \vdash P[\tilde{v} \mapsto \tilde{e}] \triangleleft \kappa'\{\tilde{J}/\tilde{i}\} + (I - \text{mar}(n, m))$  by delaying (Lemma 5). Finally, by (S-ANNOT), we obtain  $\varphi; \Phi; \Gamma \vdash \text{mar}(n, m) \cdot P[\tilde{v} \mapsto \tilde{e}] \triangleleft \kappa'\{\tilde{J}/\tilde{i}\} + I$ . By index substitution (Lemma 9), it follows from  $\varphi; \Phi \vdash \kappa \leq K$  that  $\varphi; \Phi \vdash \kappa\{\tilde{J}/\tilde{i}\} \leq K\{\tilde{J}/\tilde{i}\}$ , and as  $\varphi; \Phi \vdash \kappa\{\tilde{J}/\tilde{i}\} = \kappa'\{\tilde{J}/\tilde{i}\}$ , we derive  $\varphi; \Phi \vdash \kappa'\{\tilde{J}/\tilde{i}\} + I \leq K\{\tilde{J}/\tilde{i}\} + I$ .
2.  $\varphi; \Phi \not\models m \leq I$  and so  $\text{mar}(n, m) = m$  and  $\varphi; \Phi \vdash (I - m) + m \neq I$ . We first verify that  $\varphi; \Phi \vdash I \leq (I - m) + m$ . If for some valuation  $\rho$  we have  $\llbracket I \rrbracket_\rho \leq m$  then  $\llbracket I - m \rrbracket_\rho = 0$  and so  $\llbracket m + (I - m) \rrbracket_\rho \geq \llbracket I \rrbracket_\rho$ . If instead  $\llbracket I \rrbracket_\rho > m$ , then  $\llbracket m + (I - m) \rrbracket_\rho = \llbracket I \rrbracket_\rho$ . We now show that  $\varphi; \Phi; \downarrow_{m+(I-m)} \Gamma \vdash P[\tilde{v} \mapsto \tilde{e}] \triangleleft \kappa\{\tilde{J}/\tilde{i}\}$ . Using the fact that  $\downarrow_{\tilde{J}, \Phi}^{\varphi, \Phi}(\text{ready}(\varphi, \Phi, \downarrow_I \Gamma)) = \text{ready}(\varphi, \Phi, \downarrow_I \Gamma)$  for some  $J'$  as the context is time invariant, we have that  $\text{ready}(\varphi, \Phi, \downarrow_I \Gamma), \Gamma'$  is contained in  $\downarrow_{m+(I-m)}^{\varphi, \Phi}(\Gamma)$ . Thus, by weakening we obtain  $\varphi; \Phi; \downarrow_{m+(I-m)} \Gamma, \tilde{v} : \tilde{T} \vdash P \triangleleft \kappa$ . Then, using the fact that  $\tilde{i}$  are not free in  $\Gamma$ ,  $\forall \tilde{i}. \text{serv}_K^G(\tilde{T})$ ,  $I$  and  $\Phi$ , i.e.  $\Gamma\{\tilde{J}/\tilde{i}\} = \Gamma$  and  $\Phi\{\tilde{J}/\tilde{i}\} = \Phi$ , we derive from index substitution (Lemma 9) that  $\varphi; \Phi; \downarrow_{m+(I-m)} \Gamma, \tilde{v} : \tilde{T} \vdash P \triangleleft \kappa'\{\tilde{J}/\tilde{i}\}$  with  $\varphi; \Phi \models \kappa'\{\tilde{J}/\tilde{i}\} = \kappa'\{\tilde{J}/\tilde{i}\}$ . By application of (S-SUBSUMPTION) and by weakening (Lemma 7) we obtain  $\varphi; \Phi; \downarrow_{m+(I-m)} \Gamma, \tilde{v} : \tilde{T} \vdash \tilde{e} : \tilde{T}$  from  $\varphi; \Phi \vdash \tilde{S} \sqsubseteq \tilde{T}$ , and so by substitution (Lemma 12), we obtain  $\varphi; \Phi; \downarrow_{m+(I-m)} \Gamma \vdash P[\tilde{v} \mapsto$

~~$\tilde{e}] \triangleleft \kappa' \{ \tilde{I}/\tilde{i} \}$ . By delaying (Lemma 5) we have that  $\varphi; \Phi; \downarrow_m \Gamma \vdash P[\tilde{v} \mapsto \tilde{e}] \triangleleft \kappa' \{ \tilde{I}/\tilde{i} \} + (I - m)$ , and so by (S-ANNOT), we obtain  $\varphi; \Phi; \Gamma \vdash P[\tilde{v} \mapsto \tilde{e}] \triangleleft \kappa' \{ \tilde{I}/\tilde{i} \} + m + (I - m)$ . By index substitution (Lemma 9), it follows from  $\varphi; \Phi \vdash \kappa \leq K$  that  $\varphi; \Phi \vdash \kappa \{ \tilde{J}/\tilde{i} \} \leq K \{ \tilde{J}/\tilde{i} \}$ , and as  $\varphi; \Phi \vdash \kappa \{ \tilde{J}/\tilde{i} \} = \kappa' \{ \tilde{J}/\tilde{i} \}$ , we derive  $\varphi; \Phi \vdash \kappa' \{ \tilde{J}/\tilde{i} \} + m + (I - m) \leq K \{ \tilde{J}/\tilde{i} \} + m + (I - m)$ .~~

~~(PR-comm) We have the reduction  $n : a(\tilde{v}). P \mid m : \bar{a}(\tilde{e}) \Longrightarrow \max(n, m) : P[\tilde{v} \mapsto \tilde{e}]$ . Prior to reduction we type the process with (SA-PAR), and for the two subprocesses we have the derivations. Thus, the whole process receives the bound basis  $(\varphi; \Phi, (\kappa \mid I) \cup \{I\})$ . By application of (S-SUBSUMPTION) we have from  $\varphi; \Phi \vdash \tilde{S} \sqsubseteq \tilde{T}$  that  $\varphi; \Phi; \downarrow_I \Gamma \vdash \tilde{e} \cdot \tilde{T}$ . By substitution (Lemma 12) we thus obtain  $\varphi; \Phi; \downarrow_I \Gamma \vdash P[\tilde{v} \mapsto \tilde{e}] \triangleleft \kappa$ . As channel  $a$  retains input and output capability in both derivation trees, we derive from the definition of advancement of time that  $\varphi; \Phi \vdash \max(n, m) \leq I$ , and so by delaying (Lemma 5) we obtain  $\varphi; \Phi; \downarrow_{\max(n, m)} \Gamma \vdash P[\tilde{v} \mapsto \tilde{e}] \triangleleft \kappa + (I - \max(n, m))$ . Finally, by application of (S-ANNOT), we obtain  $\varphi; \Phi; \Gamma \vdash \max(n, m) : P[\tilde{v} \mapsto \tilde{e}] \triangleleft \kappa + I$ . By the reflexive property of  $\leq$ , we have that  $\varphi; \Phi \vdash \kappa \mid I \leq (\kappa \mid I) \cup \{I\}$ , and by Lemma 10 we obtain  $\varphi; \Phi \vdash (\kappa \mid I) \cup \{I\} \leq \text{basis}(\varphi, \Phi, (\kappa \mid I) \cup \{I\})$ . Thus, by the transitive property of  $\leq$  we obtain  $\varphi; \Phi \vdash \kappa \mid I \leq \text{basis}(\varphi, \Phi, (\kappa \mid I) \cup \{I\})$ .~~

~~(PR-par) We have the reduction  $P \mid R \Longrightarrow Q \mid R$  such that  $P \Longrightarrow Q$ . We type the parallel composition with (S-PAR) and so we have the judgements  $\varphi; \Phi; \Gamma \vdash P \triangleleft \kappa$ ,  $\varphi; \Phi; \Gamma \vdash R \triangleleft \kappa'$  and  $\varphi; \Phi; \Gamma \vdash P \mid R \triangleleft \text{basis}(\varphi, \Phi, \kappa \cup \kappa')$ . By induction we obtain  $\varphi; \Phi; \Delta \vdash Q \triangleleft \kappa''$  with  $\varphi; \Phi \vdash \kappa'' \leq \kappa$ , and by application of (S-PAR) we have that  $\varphi; \Phi; \Gamma \vdash Q \mid R \triangleleft \text{basis}(\varphi, \Phi, \kappa'' \cup \kappa')$ . It follows that  $\varphi; \Phi \vdash \kappa'' \cup \kappa' \leq \kappa \cup \kappa'$ . By Lemma 10 we then derive  $\varphi; \Phi \vdash \text{basis}(\varphi, \Phi, \kappa'' \cup \kappa') \leq \text{basis}(\varphi, \Phi, \kappa \cup \kappa')$  from  $\varphi; \Phi \vdash \text{basis}(\varphi, \Phi, \kappa'' \cup \kappa') \leq \kappa'' \cup \kappa'$  and  $\varphi; \Phi \vdash \kappa \cup \kappa' \leq \text{basis}(\varphi, \Phi, \kappa \cup \kappa')$ .~~

~~(PR-zero) We have the reduction  $\text{match } 0 \{ 0 \mapsto P; s(x) \mapsto Q \} \Longrightarrow P$ . The process must be typed with (S-MATCH), and so we have that  $\varphi; \Phi; \Gamma \vdash 0 : \text{Nat}[I, J]$ ,  $\varphi; \Phi; I \leq 0; \Gamma \vdash P \triangleleft \kappa$ ,  $\varphi; \Phi; J \geq 1, F, x : \text{Nat}[I - 1, J - 1] \vdash Q \triangleleft \kappa'$  and  $\varphi; \Phi; \Gamma \vdash \text{match } 0 \{ 0 \mapsto P; s(x) \mapsto Q \} \triangleleft \text{basis}(\varphi, \Phi, \kappa \cup \kappa')$ . For the type derivation of  $\varphi; \Phi; \Gamma \vdash 0 : \text{Nat}[I, J]$  it must be that  $\varphi; \Phi \vdash \text{Nat}[0, 0] \sqsubseteq \text{Nat}[I, J]$ . By (SS-NWEAK) this implies  $\varphi; \Phi \vdash I \leq 0$ , and so by strengthening (Lemma 8) we obtain  $\varphi; \Phi; \Gamma \vdash P \triangleleft \kappa$ . By the reflexive property of  $\leq$  we have that  $\varphi; \Phi \vdash \kappa \leq \kappa$ , and by Lemma 10 we then derive  $\varphi; \Phi \vdash \kappa \leq \text{basis}(\varphi, \Phi, \kappa \cup \kappa')$  from  $\varphi; \Phi \vdash \kappa \cup \kappa' \leq \text{basis}(\varphi, \Phi, \kappa \cup \kappa')$ .~~

~~(PR-succ) We have the reduction  $\text{match } s(e) \{ 0 \mapsto P; s(x) \mapsto Q \} \Longrightarrow Q[x \mapsto e]$ , and so we must use type rule (S-MATCH), such that  $\varphi; \Phi; \Gamma \vdash s(e) : \text{Nat}[I, J]$ ,  $\varphi; \Phi; I \leq 0; \Gamma \vdash P \triangleleft \kappa$ ,  $\varphi; \Phi; J \geq 1, F, x : \text{Nat}[I - 1, J - 1] \vdash Q \triangleleft \kappa'$  and  $\varphi; \Phi; \Gamma \vdash \text{match } s(e) \{ 0 \mapsto P; s(x) \mapsto Q \} \triangleleft \text{basis}(\varphi, \Phi, \kappa \cup \kappa')$ . The type derivation of  $\varphi; \Phi; \Gamma \vdash s(e) : \text{Nat}[I, J]$  must be of the form As  $J'$  is non-negative, we derive from (SS-NWEAK) that  $\varphi; \Phi \vdash J \geq 1$ , and so by strengthening (Lemma 8) we obtain  $\varphi; \Phi; \Gamma, x : \text{Nat}[I - 1, J - 1] \vdash Q \triangleleft \kappa'$ . By substitution (Lemma 12) we then obtain  $\varphi; \Phi; \Gamma \vdash Q[x \mapsto e] \triangleleft \kappa'$ . By the reflexive property of  $\leq$  we have that  $\varphi; \Phi \vdash \kappa' \leq \kappa'$ , and by Lemma 10 we then derive  $\varphi; \Phi \vdash \kappa' \leq \text{basis}(\varphi, \Phi, \kappa \cup \kappa')$  from  $\varphi; \Phi \vdash \kappa \cup \kappa' \leq \text{basis}(\varphi, \Phi, \kappa \cup \kappa')$ .~~

~~(PR-annot) We have the reduction  $n : P \Rightarrow n : Q$  such that  $P \Rightarrow Q$ , and so we must use type rule (S-ANNOT), such that  $\varphi; \Phi; \downarrow_n \Gamma \vdash P \triangleleft \kappa$  and  $\varphi; \Phi; \Gamma \vdash \text{tick}.P \triangleleft \kappa + n$ . By induction we obtain  $\varphi; \Phi; \downarrow_n \Gamma \vdash P \triangleleft \kappa'$  with  $\varphi; \Phi \vdash \kappa' \leq \kappa$ . By application of (S-ANNOT) we then obtain  $\varphi; \Phi; \Gamma \vdash n : Q \triangleleft \kappa + n$ . It follows from  $\varphi; \Phi \vdash \kappa' \leq \kappa$  that  $\varphi; \Phi \vdash \kappa' + n \leq \kappa + n$ .~~

~~(PR-res) We have the reduction  $(\nu a : T)P \Rightarrow (\nu a : T)Q$  where  $P \Rightarrow Q$ . We must use type rule (S-NU) and so we have  $\varphi; \Phi; \Gamma \vdash (\nu a : T)P \triangleleft \kappa$  and  $\varphi; \Phi; \Gamma, a : T \vdash P \triangleleft \kappa$ . By induction we obtain  $\varphi; \Phi; \Gamma, a : T \vdash Q \triangleleft \kappa'$  such that  $\varphi; \Phi \vdash \kappa' \leq \kappa$ . By application of (S-NU) we derive  $\varphi; \Phi; \Gamma \vdash (\nu a : T)Q \triangleleft \kappa'$ .~~

~~(PR-struct) We have the reduction  $P \Rightarrow Q$  where  $P \equiv P'$ ,  $P' \Rightarrow Q'$  and  $Q' \equiv Q$ . We have some typing  $\varphi; \Phi; \Gamma \vdash P \triangleleft \kappa$ , and so by Lemma 13 we obtain  $\varphi; \Phi; \Gamma \vdash P' \triangleleft \kappa_1$  with  $\varphi; \Phi \vdash \kappa_1 \leq \kappa$ . By induction we then have that  $\varphi; \Phi; \Gamma \vdash Q' \triangleleft \kappa_2$  with  $\varphi; \Phi \vdash \kappa_2 \leq \kappa_1$ . Finally, by application of Lemma 13 again we obtain  $\varphi; \Phi; \Gamma \vdash Q \triangleleft \kappa_3$  with  $\varphi; \Phi \vdash \kappa_3 \leq \kappa_2$ . By the transitive property of  $\leq$  we then have that  $\varphi; \Phi \vdash \kappa_3 \leq \kappa$ .~~

~~(PR-tick) We have the reduction  $\text{tick}.P \Rightarrow 1 : P$ . This result is obtained directly from the fact that the type rules for  $\text{tick}.P$  and  $1 : P$  are equivalent.~~

#### 4.6 Upper bound on parallel complexity

We now use the subject reduction result to prove that a typing of a process provides an upper bound on the parallel complexity of the process. The idea is to prove that a typing of a process bounds the local complexity of the process. It then follows from subject reduction, i.e. that well-typed processes reduce to well-typed processes with decreasing complexity bounds, that a complexity bound from a typing of a process is a bound on the local complexity of any process in a reduction sequence from the process. The result that a typing of a process bounds its local complexity is proved in Lemma 14. The proof is quite straightforward, using the fact that any annotated process is structurally congruent to one in annotated canonical form with an equivalent typing. Then it follows from Lemma 10 and type rule (S-ANNOT) that we bound the local complexity. The main result is shown in Corollary 1.

**Lemma 14 (Local complexity).** *If  $\varphi; \Phi; \Gamma \vdash P \triangleleft \kappa$  and  $\varphi; \Phi \vdash \kappa \leq K$  then  $\varphi; \Phi \vdash \mathcal{C}_\ell(P) \leq K$ .*

*Proof.* By Lemma 2, any annotated process is structurally congruent to one in annotated canonical form. By Lemma 13,  $\varphi; \Phi; \Gamma \vdash P \triangleleft \kappa$  and  $P \equiv Q$  implies  $\varphi; \Phi; \Gamma \vdash Q \triangleleft \kappa'$  with  $\varphi; \Phi \vdash \kappa = \kappa'$ , and so it suffices to consider processes in annotated canonical form. Then we derive the typing where  $\varphi; \Phi \vdash \kappa' = \text{basis}(\varphi, \Phi, \kappa_1 \cup \dots \cup \kappa_n)$  and  $\varphi; \Phi \vdash \kappa_1 \cup \dots \cup \kappa_n \leq \text{basis}(\varphi, \Phi, \kappa_1 \cup \dots \cup \kappa_n)$  by Lemma 10. Thus, it suffices to show that for  $\varphi; \Phi; \Gamma, \tilde{a} : \tilde{T} \vdash m_i : G_i \triangleleft \kappa_i$  we have  $\varphi; \Phi \vdash \mathcal{C}_\ell(m_i : G_i) \leq \kappa_i$ . We obtain this directly from type rule (S-ANNOT), and so by the transitive property of  $\leq$ , any index  $K$  such that  $\varphi; \Phi \vdash \text{basis}(\varphi, \Phi, \kappa_1 \cup \dots \cup \kappa_n) \leq K$  is also a bound on  $\mathcal{C}_\ell(m_1 : G_1)$  through  $\mathcal{C}_\ell(m_n : G_n)$ .

**Corollary 1 (Parallel complexity).** *If  $\varphi; \Phi; \Gamma \vdash P \triangleleft \kappa$  and  $\varphi; \Phi \vdash \kappa \leq K$  then  $\varphi; \Phi \vdash \mathcal{C}_\mathcal{P}(P) \leq K$ .*



~~Proof. This result is an immediate consequence of Lemma 14 and subject reduction (Theorem 2).~~

~~In Lemma 15, we finally present a convenient result with regards to closed processes. That is, any typable closed process can be assigned a singleton combined complexity, thus immediately providing a numeral bound on its span.~~

**Lemma 15 (Closed processes).** *If  $P$  is a well-typed closed process such that  $\cdot; \Phi; \Gamma \vdash P \triangleleft \kappa$  then  $|\kappa| = 1$ .*

~~Proof. By case analysis on the type rules.~~

#### 4.7 Verification of constraint judgements

Until now we have not considered how we can verify constraint judgements in the type rules. The expressiveness of implementations of the type system by Baillot and Ghyselen [3] depends on both the expressiveness of indices and whether judgements on the corresponding constraints are decidable. Naturally, we are interested in both of these properties, and so in this section, we show how judgements on linear constraints can be verified using algorithms. Later, we show how this can be extended to certain groups of polynomial constraints. We first make some needed changes to how the type checker uses subtraction.

#### 4.8 Subtraction of naturals

The constraint judgements rely on a special minus operator ( $\dot{-}$ ) for indices such that  $n \dot{-} m = 0$  when  $m \geq n$ , which we refer to as the *monus* operator. This is apparent in the pattern match constructor type rule from Chapter ???. Without this behavior, we may encounter problems when checking subtype premises in match processes. This has the consequence that equations such as  $2 \dot{-} 3 + 3 = 3$  hold, such that indices form a semiring rather than a ring, as we are no longer guaranteed an additive inverse. In general, semirings lack many properties of rings that are desirable. For example, given two seemingly equivalent constraints  $i \leq 5$  and  $i \dot{-} 5 \leq 0$ , we see that by adding any constant to their left-hand sides, the constraints are no longer equivalent. Adding the constant 2 to their left-hand sides, we obtain  $i + 2 \leq 5$  and  $i \dot{-} 5 + 2 \leq 0$ , however, we see that the first constraint is satisfied given the valuation  $i = 3$  but the second is not. In general the associative property of  $+$  is lost.

Unfortunately, this is not an easy problem to solve implementation-wise, as indices are not actually evaluated but rather represent whole feasible regions. Thus, instead of trying to implement this operator exactly, we limit the number of processes typable by the type system. Removing the operator entirely is not an option as it is used by the type rules themselves. Instead, we ensure that one cannot *exploit* the special behavior of monus by introducing additional conditions to the type rules of the type system. More precisely, any time the type system uses the monus operator such as  $I \dot{-} J$ , we require the premise  $\varphi; \Phi \models I \geq J$ , in

which case the monus operator is safe to treat as a regular minus. This, however, puts severe restrictions on the number of processes typable, and so we relax the restriction a bit by also checking the judgement  $\varphi; \Phi \models I \leq J$ , in which case we can conclude that the result is definitely 0. If neither  $\varphi; \Phi \models I \geq J$  nor  $\varphi; \Phi \models I \leq 0$  hold, which is possible as  $\leq$  and  $\geq$  do not form a total order on indices, the result is undefined. We refer to this variant of monus as the *partial* monus operator, as formalized in Definition 16. Note that this definition of monus allows us to obtain identical behavior to minus on a constraint  $I \bowtie J$  by moving terms between the LHS and RHS, i.e.  $I - K \bowtie J \Rightarrow I \bowtie J + K$ , and so we can assume we have a standard minus operator when verifying judgements on constraints. For the remainder of this section, we assume this definition is used in the type rules instead of the usual monus. We may omit  $\varphi; \Phi$  if it is clear from the context.

**Definition 16 (Partial monus).** Let  $\Phi$  be a set of constraints in index variables  $\varphi$ . The partial monus operator is defined for two indices  $I$  and  $J$  as

$$I \dot{-}_{\varphi; \Phi} J = \begin{cases} I - J & \text{if } \varphi; \Phi \models J \leq I \\ 0 & \text{if } \varphi; \Phi \models I \leq J \\ \text{undefined} & \text{otherwise} \end{cases}$$

To ensure soundness of the algorithmic type rules after switching to the partial monus operator, we must make some changes to advancement of time. Consider the typing

$$(\cdot, i); (\cdot, i \leq 3); \Gamma \vdash !a().\mathbf{0} \mid 5 : \bar{a}\rangle \triangleleft \{5\}$$

where  $\Gamma = \cdot, a : \forall_{3-i \in \text{serv}_0^{\{\text{in}, \text{out}\}}}()$ . Upon typing the time annotation, we advance the time of the server type by 5 yielding the type  $\forall_{3-i-5 \in \text{serv}_0^{\{\text{out}\}}}()$  as  $(\cdot, i); (\cdot, i \leq 3) \not\models 3-i \geq 5$ , which is defined as  $(\cdot, i); (\cdot, i \leq 3) \models 3-i \leq 5$ . However, if we apply the congruence rule (SC-SUM) from right to left we obtain

$$!a().\mathbf{0} \mid 2 : 3 : \bar{a}\rangle \equiv !a().\mathbf{0} \mid 5 : \bar{a}\rangle$$

Then, we get a problem upon typing the first annotation. That is, as  $(\cdot, i); (\cdot, i \leq 3) \not\models 3-i \leq 2$  (i.e. when for some valuation  $\rho$  we have  $\rho(i) = 0$ ) the operation  $(3-i) \dot{-}_{(\cdot, i); (\cdot, i \leq 3)} 2$  is undefined. Thus, the type system loses its subject congruence property, and subsequently its subject reduction property. There are, however, several ways to address this. One option is to modify the type rules to perform a single advancement of time for a sequence of annotations. A more contained option is to remove monus from the definition of advancement of time, by enriching the formation rules of types with the constructor  $\forall_I \tilde{i}. \text{serv}_K^{\sigma}(\tilde{T})^{-J}$

and by augmenting the definition of advancement as so

$$\begin{aligned} \downarrow_I^{\varphi; \Phi}(\forall_J \tilde{i}. \mathbf{serv}_K^\sigma(\tilde{T})) &= \begin{cases} \forall_{J-I} \tilde{i}. \mathbf{serv}_K^\sigma(\tilde{T}) & \text{if } \varphi; \Phi \models I \leq J \\ \forall_0 \tilde{i}. \mathbf{serv}_K^{\sigma \cap \{\text{out}\}}(\tilde{T}) & \text{if } \varphi; \Phi \models J \leq I \\ \forall_J \tilde{i}. \mathbf{serv}_K^{\sigma \cap \{\text{out}\}}(\tilde{T})^{-I} & \text{if } \varphi; \Phi \not\models I \leq J \text{ and } \varphi; \Phi \not\models J \leq I \end{cases} \\ \downarrow_I^{\varphi; \Phi}(\forall_J \tilde{i}. \mathbf{serv}_K^\sigma(\tilde{T})^{-L}) &= \begin{cases} \forall_0 \tilde{i}. \mathbf{serv}_K^{\sigma \cap \{\text{out}\}}(\tilde{T}) & \text{if } \varphi; \Phi \models J \leq L + I \\ \forall_J \tilde{i}. \mathbf{serv}_K^{\sigma \cap \{\text{out}\}}(\tilde{T})^{-(L+I)} & \text{if } \varphi; \Phi \not\models J \leq L + I \end{cases} \end{aligned}$$

This in essence introduces a form of *lazy* time advancement, where time is not advanced until partial monus allows us to do so. Then, as the type rules for servers require a server type of the form  $\forall_J \tilde{i}. \mathbf{serv}_K^\sigma(\tilde{T})$ , the summed advancement of time must always be less than or equal, or always greater than or equal to the time of the server, and so typing is invariant to the use of congruence rule (SC-SUM). Revisiting the above example, we have that  $\downarrow_5^{(\cdot, i); (\cdot, i \leq 3)}(\forall_{3-i} \epsilon. \mathbf{serv}_0^{\{\text{in}, \text{out}\}}()) = \downarrow_3^{(\cdot, i); (\cdot, i \leq 3)}(\downarrow_2^{(\cdot, i); (\cdot, i \leq 3)}(\forall_{3-i} \epsilon. \mathbf{serv}_0^{\{\text{in}, \text{out}\}}()))$ , and so we obtain the original typing

$$(\cdot, i); (\cdot, i \leq 3); \Gamma \vdash !a().0 \mid 2 : 3 : \overline{a} \triangleleft \{5\}$$

**Undecidability of polynomial constraint judgements** As we have seen, verifying that a constraint imposes no further restrictions onto index valuations amounts to checking whether all possible index valuations that satisfy a set of known constraints are also contained in the model space of our new constraint. It also amounts to checking whether the feasible region of the constraint contains the feasible region of a known system of inequality constraints, or checking whether the feasible region of the inverse constraint does not intersect the feasible region of a known system of inequality constraints. This turns out to be a difficult problem, and we can in fact prove it undecidable for diophantine constraints, i.e. multivariate polynomial inequalities with integer coefficients, when index variables must have natural (or integer) interpretations. The main idea is to reduce Hilbert's tenth problem [8] to that of verification of judgements on constraints, as this problem has been proven undecidable [7]. That is, we show that assuming some complete algorithm that verifies judgements on constraints, we can verify whether an arbitrary diophantine equation has a solution with all unknowns taking integer values. We show this result in Lemma 16.

**Lemma 16.** *Let  $C$  and  $C' \in \Phi$  be diophantine inequalities with unknowns in  $\varphi$  and coefficients in  $\mathbb{N}$ . Then the judgement  $\varphi; \Phi \models C$  is undecidable.*

*Proof.* By reduction from Hilbert's tenth problem. ~~Let  $p = 0$  be an arbitrary diophantine equation. We show that assuming some algorithm that can verify a judgement of the form  $\varphi; \Phi \vdash C$ , we can determine whether  $p = 0$  has an integer solution. We must pay special attention to the non-standard definition of subtraction in the type system by Baillet and Chyslen [3] and to the fact that only non-negative integers substitute for index variables. We first replace each~~

Matijasevic (citation)

~~integer variable  $x$  in  $p$  with two non negative variables  $i_x, j_x$ , referring to the modified polynomial as  $p'$ . We can quickly verify that  $p' = 0$  has a non-negative integer solution if and only if  $p = 0$  has an integer solution~~

- ~~1. Assume that  $p' = 0$  has a non-negative integer solution. Then for each variable  $x$  in  $p$  we assign  $x = i_x - j_x$  reaching an integer solution to  $p$ .~~
- ~~2. Assume that  $p = 0$  has an integer solution. Then for each pair  $i_x$  and  $j_x$  in  $p'$  we assign  $i_x = x$  and  $j_x = 0$  when  $x \geq 0$  and  $i_x = 0$  and  $j_x = |x|$  when  $x < 0$  reaching a non-negative integer solution to  $p'$ .~~

~~Then, by the distributive property of integer multiplication and the associative property of integer addition, we can utilize that  $p'$  has an equivalent expanded form~~

$$~~p' = n_1 t_1 + \dots + n_k t_k + n_{k+1} t_{k+1} + \dots + n_{k+l} t_{k+l}~~$$

~~such that  $n_1, \dots, n_k \in \mathbb{N}$ ,  $n_{k+1}, \dots, n_{k+l} \in \mathbb{Z}^{\leq 0}$  and  $t_1, \dots, t_k, t_{k+1}, \dots, t_{k+l}$  are power products over the set of all index variables in  $p'$  denoted  $\varphi_{p'}$ . We can then factor the negative coefficients~~

$$~~\begin{aligned} p' &= n_1 t_1 + \dots + n_k t_k + n_{k+1} t_{k+1} + \dots + n_{k+l} t_{k+l} \\ &= (n_1 t_1 + \dots + n_k t_k) + (-1)(|n_{k+1}| t_{k+1} + \dots + |n_{k+l}| t_{k+l}) \\ &= (n_1 t_1 + \dots + n_k t_k) - (|n_{k+1}| t_{k+1} + \dots + |n_{k+l}| t_{k+l}) \end{aligned}~~$$

~~We use this to show that  $p' = 0$  has a non-negative integer solution if and only if the following judgement does not hold~~

$$~~\varphi_{p'}; \{|n_{k+1}| t_{k+1} + \dots + |n_{k+l}| t_{k+l} \leq n_1 t_1 + \dots + n_k t_k\} \vdash 1 \leq (n_1 t_1 + \dots + n_k t_k) - (|n_{k+1}| t_{k+1} + \dots + |n_{k+l}| t_{k+l})~~$$

~~We consider the implications separately~~

- ~~1. Assume that  $p' = 0$  has a non-negative integer solution. Then we have that  $n_1 t_1 + \dots + n_k t_k = |n_{k+1}| t_{k+1} + \dots + |n_{k+l}| t_{k+l}$ , and so there must exist a valuation  $\rho : \varphi_{p'} \rightarrow \mathbb{N}$  such that  $\llbracket n_1 t_1 + \dots + n_k t_k \rrbracket_\rho = \llbracket |n_{k+1}| t_{k+1} + \dots + |n_{k+l}| t_{k+l} \rrbracket_\rho$ . We trivially have that  $\rho$  satisfies  $\llbracket |n_{k+1}| t_{k+1} + \dots + |n_{k+l}| t_{k+l} \rrbracket_\rho \leq \llbracket n_1 t_1 + \dots + n_k t_k \rrbracket_\rho$ . But  $\rho$  is not in the model space of the constraint  $1 \leq (n_1 t_1 + \dots + n_k t_k) - (|n_{k+1}| t_{k+1} + \dots + |n_{k+l}| t_{k+l})$ , and so the judgement does not hold.~~
- ~~2. Assume that the judgement does not hold. Then there must exist a valuation  $\rho : \varphi_{p'} \rightarrow \mathbb{N}$  that satisfies  $\llbracket |n_{k+1}| t_{k+1} + \dots + |n_{k+l}| t_{k+l} \rrbracket_\rho \leq \llbracket n_1 t_1 + \dots + n_k t_k \rrbracket_\rho$ , but that is not in the model space of the constraint  $1 \leq (n_1 t_1 + \dots + n_k t_k) - (|n_{k+1}| t_{k+1} + \dots + |n_{k+l}| t_{k+l})$ . This implies that  $\llbracket n_1 t_1 + \dots + n_k t_k \rrbracket_\rho = \llbracket |n_{k+1}| t_{k+1} + \dots + |n_{k+l}| t_{k+l} \rrbracket_\rho$ , and so  $p'$  has a non-negative integer solution.~~

~~As such, we can verify that the above judgement does not hold if and only if  $p'$  has a non-negative integer solution, and by extension if and only if  $p$  has an integer solution. Thus, we would have a solution to Hilbert's tenth problem, which is undecidable.~~

This is an open problem!

As an unfortunate consequence of Lemma 16, we are forced into considering approximate algorithms for verification of judgements over polynomial constraints (in general). However, this result does not imply that type checking is undecidable. It may well be that problematic judgements are not required to type check any process, as computational complexity has certain properties, such as monotonicity. Note that the freedom of type checking, i.e. we can specify an arbitrary type context as well as type annotations, enables us to select indices that lead to undecidable judgements. To prove that type checking is undecidable, however, a more reasonable result would be that there exists a process that is typable if and only if an undecidable judgement is satisfied. This is out of the scope of this thesis, and so we leave it as future work.

#### 4.9 Normalization of linear indices

To make checking of judgements on constraints tractable, we reduce the set of function symbols on which indices are defined, such that indices may only contain integers and index variables, as well as addition, subtraction and scalar multiplication operators, such that we restrict ourselves to linear functions.

$$I, J ::= n \mid i \mid I + J \mid I - J \mid nI$$

Such indices can be written in a *normal* form, presented in Definition 17.

**Definition 17 (Normalized linear index).** *Let  $I$  be an index in index variables  $\varphi = i_1, \dots, i_n$ . We say that  $I$  is a normalized index when it is a linear combination of index variables  $i_1, \dots, i_n$ . Let  $m$  be an integer constant and  $I_\alpha \in \mathbb{Z}$  the coefficient of variable  $i_\alpha$ , we then define normalized indices as*

$$I = m + \sum_{\alpha \in \mathcal{E}(I)} I_\alpha i_\alpha$$

*We use the notation  $\mathcal{B}(I)$  and  $\mathcal{E}(I)$  to refer to the constant and unique identifiers of index variables of  $I$ , respectively.*

Any index can be transformed to an equivalent normalized index (i.e. it is a normal form) through expansion with the distributive law, reordering by the commutative and associative laws and then by regrouping terms that share variables. Therefore, the set of normalized indices in index variables  $i_1, \dots, i_n$  and with coefficients in  $\mathbb{Z}$ , denoted  $\mathbb{Z}[i_1, \dots, i_n]$ , is a free module with the variables as basis, as the variables are linearly independent. In Definition 18, we show how scalar multiplication, addition and multiplication of normalized indices (i.e. linear combinations of monomials) can be defined. Definition 19 shows how an equivalent normalized index can be computed from an arbitrary linear index using these operations.

**Definition 18 (Operations in  $\mathbb{Z}[i_1, \dots, i_n]$ ).** *Let  $I = n + \sum_{\alpha \in \varphi_1} I_\alpha i_\alpha$  and  $J = m + \sum_{\alpha \in \varphi_2} J_\alpha i_\alpha$  be normalized indices in index variables  $i_1, \dots, i_n$ . We*

define addition and scalar addition of such indices. Given a scalar  $n \in \mathbb{Z}$ , the scalar multiplication  $nI$  is

$$nI = n \cdot m + \sum_{\alpha \in \mathcal{E}(I)} nI_\alpha i_\alpha$$

When  $d$  is a common divisor of all coefficients in  $I$ , i.e.  $I_\alpha/n \in \mathbb{Z}$  for all  $\alpha \in \varphi$ , the inverse operation is defined

$$\frac{I}{d} = \frac{n}{d} + \sum_{\alpha \in \mathcal{E}(I)} \frac{I_\alpha}{d} i_\alpha \quad \text{if } \frac{I_\alpha}{d} \in \mathbb{Z} \text{ for all } \alpha \in \mathcal{E}(I)$$

The addition of  $I$  and  $J$  is the sum of constants plus the sum of scaled variables where coefficients  $I_\alpha$  and  $J_\alpha$  are summed when  $\alpha \in \varphi_1 \cap \varphi_2$

$$I + J = n + m + \sum_{\alpha \in \mathcal{E}(I) \cup \mathcal{E}(J)} (I_\alpha + J_\alpha) i_\alpha$$

where for any  $\alpha \in \varphi_1 \cup \varphi_2$  such that  $I_\alpha + J_\alpha = 0$  we omit the corresponding zero term. The inverse of addition is always defined for elements of a polynomial ring

$$I - J = n - m + \sum_{\alpha \in \mathcal{E}(I) \cup \mathcal{E}(J)} (I_\alpha - J_\alpha) i_\alpha$$

**Definition 19 (Index normalization).** The normalization of some index  $I$  in index variables  $i_1, \dots, i_n$  into an equivalent normalized index  $\mathcal{N}(I) \in \mathbb{Z}[i_1, \dots, i_n]$  is a homomorphism defined inductively

$$\begin{aligned} \mathcal{N}(n) &= ni_1^0 \cdots i_n^0 \\ \mathcal{N}(i_j) &= 1i_1^0 \cdots i_j^1 \cdots i_n^0 \\ \mathcal{N}(I + J) &= \mathcal{N}(I) + \mathcal{N}(J) \\ \mathcal{N}(I - J) &= \mathcal{N}(I) - \mathcal{N}(J) \\ \mathcal{N}(nI) &= n\mathcal{N}(I) \end{aligned}$$

We extend normalization to constraints. We first note that an equality constraint  $I = J$  is satisfied if and only if  $I \leq J$  and  $J \leq I$  are both satisfied. Thus, it suffices to only consider inequality constraints. A normalized constraint is of the form  $I \leq 0$  for some normalized index  $I$ , as formalized in Definition 20.

**Definition 20 (Normalized constraints).** Let  $C = I \leq J$  be an inequality constraint such that  $I$  and  $J$  are normalized indices. We say that  $I - J \leq 0$  is the normalization of  $C$  denoted  $\mathcal{N}(C)$ , and we refer to constraints in this form as normalized constraints.

Normalizing constraints has a number of benefits. First of all, it ensures that equivalent constraints are always expressed the same way. Secondly, having all constraints in a common form where variables only appear once means we can easily reason about individual variables of a constraint, which will be useful later when we verify constraint judgements.

#### 4.10 Checking for emptiness of model space

As explained in Section 3.2, we can verify a constraint judgement  $\varphi; \Phi \models C_0$  by letting  $C'_0$  be the inverse of  $C_0$  and checking if  $\mathcal{M}_\varphi(\Phi \cup \{C'_0\}) = \emptyset$  holds. Being able to check for non-emptiness of a model space is therefore paramount for verifying constraint judgements. For convenience, given a finite ordered set of index variables  $\varphi = \{i_1, i_2, \dots, i_n\}$ , we represent a normalized constraint  $I \leq 0$  as a vector  $(\mathcal{B}(I), I_1 \ I_2 \ \dots \ I_n)_\varphi$ . As such, the constraint  $-5i + -2j + -4k \leq 0$  can be represented by the vector  $(0 \ -5 \ -2 \ -4)_{\varphi_1}$  where  $\varphi_1 = \{i, j, k\}$ . Another way to represent that same constraint is with the vector  $(0 \ -5 \ -2 \ 0 \ -4)_{\varphi_2}$  where  $\varphi_2 = \{i, j, l, k\}$ . We denote the vector representation of a constraint  $C$  over a finite ordered set of index variables  $\varphi$  by  $\mathbf{C}_\varphi$ . We extend this notation to sets of constraints, such that  $\Phi_\varphi$  denotes the set of vector representations over  $\varphi$  of normalized constraints in  $\Phi$ .

Recall that the model space of any set of constraints  $\Phi$  is the set of all valuations satisfying all constraints in  $\Phi$ . Thus, to show that  $\mathcal{M}_\varphi(\Phi)$  is empty, we must show that no valuation  $\rho$  exists satisfying all constraints in  $\Phi$ . This is a linear constraint satisfaction problem (CSP) with an infinite domain. One method for solving such is by optimization using the simplex algorithm. If the linear program of the CSP has a feasible solution, the model space is non-empty and if it does not have a feasible solution, the model space is empty.

As is usual for linear constraints, our linear constraints can be thought of as hyper-planes dividing some n-dimensional space in two, with one side constituting the feasible region and the other side the non-feasible region. By extension, for a set of constraints their shared feasible region is the intersection of all of their individual feasible regions. Since the feasible region of a set of constraints is defined by a set of hyper-planes, the feasible region consists of a convex polytope. This fact is used by the simplex algorithm when performing optimization.

The simplex algorithm has some requirements to the form of the linear program it is presented, i.e. that it must be in *standard* form. The standard form is a linear program expressed as

$$\begin{aligned} & \text{minimize} && \mathbf{c}^T \mathbf{a} \\ & \text{subject to} && M\mathbf{a} = \mathbf{b} \\ & && \mathbf{a} \geq \mathbf{0} \end{aligned}$$

where  $M$  is a matrix representing constraints,  $\mathbf{a}$  is a vector of scalars, and  $\mathbf{b}$  is a vector of constants. As such, we first need all our constraints to be of the form  $a_0 \cdot i_0 + \dots + a_n \cdot i_n \leq b$ , after which we must convert them into equality constraints by introducing *slack* variables that allow the equality to also take on lower values. Since all of our constraints are normalized and of the form  $I \leq 0$ , all of our slack variables will have negative coefficients. In our specific case, we let row  $i$  of  $M$  consist of  $(\mathbf{C}_\varphi^i)_{-1}$ , where  $(\cdot)_{-1}$  removes the first element of the vector

(the constant term here). We must also include our slack variables, and so we augment row  $i$  of  $M$  with the  $n$ -vector with all zeroes except at position  $i$  where it is  $-1$ . We let  $\mathbf{a}$  be a column vector containing our variables in  $\varphi$  as well as our slack variables, and finally we let  $\mathbf{b}_i = -(\mathbf{C}_\varphi^i)_1$ .  $\mathbf{c}$  may be an arbitrary vector.

Checking feasibility of the above linear program can itself be formulated as a linear program that is guaranteed to be feasible, enabling us to use efficient polynomial time linear programming algorithms, such as interior point methods, to check whether constraints are covered. Let  $\mathbf{s}$  be a new vector, then we have the linear program

$$\begin{aligned} & \text{minimize} && \mathbf{1}^T \mathbf{s} \\ & \text{subject to} && M\mathbf{a} + \mathbf{s} = \mathbf{b} \\ & && \mathbf{a}, \mathbf{s} \geq \mathbf{0} \end{aligned}$$

where  $\mathbf{1}$  is the vector of all ones. We can verify the feasibility of this problem with the certificate  $(\mathbf{a}, \mathbf{s}) = (\mathbf{0}, \mathbf{b})$ . Then the original linear program is feasible if and only if the augmented problem has an optimal solution  $(\mathbf{x}^*, \mathbf{s}^*)$  such that  $\mathbf{s}^* = \mathbf{0}$ .

Given a constraint judgement  $\varphi; \Phi \models C_0$ , it should be noted that while the simplex algorithm can be used to check if a solution exists to the constraints  $\Phi \cup \{C_0'\}$ , there is no guarantee that the solution is an integer solution nor that an integer solution exists at all. Thus, in the case that a non-integer solution exists but no integer solution, this method will over-approximate. An example of such is the two constraints  $3i - 1 \leq 0$  and  $-2i + 1 \leq 0$  yielding the feasible region where  $\frac{1}{3} \leq i \leq \frac{1}{2}$ , containing no integers. For an exact solution, we may use integer programming.

*Example 3.* Given the constraints

$$\begin{aligned} C^1 &= 3i - 3 \leq 0 \\ C^2 &= j + 2k - 2 \leq 0 \\ C^3 &= -k \leq 0 \\ C^{new} &= i + j - 3 \leq 0 \end{aligned}$$

we want to check if the constraint judgement  $\{i, j, k\}; \{C^1, C^2, C^3\} \models C^{new}$  is satisfied

We first let  $C^{newinv}$  be the inversion of constraint  $C^{new}$ .

$$C^{newinv} = 1i + 1j - 2 \geq 0$$

We now want to check if the feasible region  $\mathcal{M}_\varphi(\{C^1, C^2, C^3, C^{newinv}\})$  is nonempty. To do so, we construct a linear program with the four constraints. To convert all inequality constraints into equality constraints, we add the slack variables  $s_1, s_2, s_3, s_4$



$$\begin{array}{ll}
\text{minimize} & i + j + k \\
\text{subject to} & 3i + 0j + 0k + s_1 = 3 \\
& 0i + 1j + 2k + s_2 = 2 \\
& 0i + 0j - 1k + s_3 = 0 \\
& 1i + 1j + 0k - s_4 = 2 \\
& i, j, k, s_1, s_2, s_3, s_4 \geq 0
\end{array}$$

Using an algorithm such as the simplex algorithm, we see that there is no feasible solution, and so we conclude that the constraint judgement  $\{i, j, k\}; \{C^1, C^2, C^3\} \models C^{new}$  is satisfied.

**Reducing polynomial constraints to linear constraints** Many programs do not run in linear time, and so we cannot type them if we are constrained to just verifying linear constraint judgements. In this section we show how we can reduce certain polynomial constraints to linear constraints, enabling us to use the techniques described above. We first extend our definition of indices such that they can be used to express multivariate polynomials. We assume a normal form for polynomial indices akin to that of linear indices. Terms are now monomials with integer coefficients.

$$I, J ::= n \mid i \mid I + J \mid I - J \mid IJ$$

When reducing normalized constraints with polynomial indices to normalized constraints with linear indices, we wish to construct new linear constraints that are only satisfied if the original polynomial constraint is satisfied. For example, given the constraint  $-i^2 + 10 \leq 0$ , one can see that this polynomial constraint can be simulated using the constraint  $-i + \sqrt{10} \leq 0$ . We notice that the reason this is possible is that when  $-i^2 + 10 \leq 0$  holds, i.e. when  $i \geq \sqrt{10}$ , the value of  $i$  can always be increased without violating the constraint. Similarly, when  $-i^2 + 10 \geq 0$  holds, the value of  $i$  can always be decreased until reaching its minimum value of 0 without violating the constraint. We can thus introduce a new simpler constraint with the same properties, i.e.  $-i + \sqrt{10} \leq 0$ . More specifically, the polynomials of the left-hand side of the two constraints share the same positive real-valued roots as well as the same sign for any value of  $i$ . In general, limiting ourselves to univariate polynomials, for any constraint whose left-hand side polynomial only has a single positive root, we can simulate such a constraint using a constraint of the form  $a \cdot i + c \leq 0$ . For describing complexities of programs, we expect to mostly encounter monotonic polynomials with at most a single positive real-valued root.

Note that the above has the consequence that we may end up with irrational coefficients for indices of constraints. While such constraints are not usually allowed, they can still represent valid bounds for index variables. As such, we

can allow them as constraints in this context. Furthermore, any irrational coefficient can be approximated to an arbitrary precision using a rational coefficient.

For finding the roots of a specific polynomial we can use either analytical or numerical methods. Using analytical methods has the advantage of being able to determine all roots with exact values, however, we are limited to polynomials of degree at most four as stated by the Abel-Ruffini theorem [2]. With numerical methods, we are not limited to polynomials of a specific degree, however, numerical methods often require a given interval to search for a root and do not guarantee to find all roots. Introducing constraints with false restrictions may lead to an under-approximating type system, and so we must be careful not to introduce such. We must therefore ensure we find all roots to avoid constraints with false restrictions. We can use Descartes rule of signs to get an upper bound on the number of positive real roots of a polynomial. Descartes's rule of signs states that the number of roots in a polynomial is at most the number of sign-changes in its sequence of coefficients.

For our application, we decide to only consider constraints whose left-hand side polynomial is univariate and monotonic with a single root. In some cases we may remove safely positive monomials in a normalized constraint to obtain such constraints. We limit ourselves to these constraints both to keep complexity down, as well as because we expect to mainly encounter such polynomials when considering complexity analysis of programs. We use Laguerre's method as a numerical method to find the root of the polynomial, which has the advantage that it does not require any specified interval when performing root-finding. Assuming we can find a root  $r$ , we add an additional constraint  $\pm(i - r) \leq 0$  where the sign depends on whether the original polynomial is increasing or decreasing.

Additionally, given a non-linear monomial, we may also treat this as a single unit and construct linear combinations of this by treating the monomial as a single fresh variable. For example, given normalized constraints  $C_1 = i^2 + 4 \leq 0$ ,  $C_2 = i - 2 \leq 0$ , and  $C_3 = 2ij \leq 0$ , we may view these as the linear constraints  $C'_1 = k + 4 \leq 0$ ,  $C'_2 = i - 2 \leq 0$ , and  $C'_3 = 2l \leq 0$  where  $k = i^2$  and  $l = ij$ . This may make the feasible region larger than it actually is, meaning we over-approximate when verifying judgements on polynomial constraints.

*Example 4.* We want to check if the following judgement holds

$$\{i, j\}; \{-2i \leq 0, -1i^2 + 1j + 1 \leq 0\} \models -2i + 2 \leq 0$$

We first try to generate new constraints of the form  $-a \cdot i + r \leq 0$  for some index variable  $i$  and some constants  $a$  and  $r$  based on our two existing constraints using the root-finding method. The first constraint is already of such form, so we can only consider the second. For the second, we first use the subconstraint relation to remove the term  $1j$  obtaining  $-1i^2 + 1 \leq 0$ . Next, we note that  $-1i^2 + 1$  is a monotonically decreasing polynomial as every coefficient excluding the constant term is negative. We then find the root  $r = 1$  of the polynomial

and add a new constraint  $-1i + 1 \leq 0$  to our set of constraints. Finally, we can invert the constraint  $-2i + 2 \leq 0$  obtaining the constraint  $-2i + 1 \geq 0$ . We now need to check if the feasible region  $\mathcal{M}_{\{i,j,i^2\}}(\{-2i \leq 0, -1i^2 + 1j + 1 \leq 0, -1i + 1 \leq 0, -2i + 1 \geq 0\})$  is empty. Treating the monomial  $i^2$  as its own separate variable and solving this as a linear program using an algorithm such as the simplex algorithm, we see that there is indeed no solution, and so the constraint is satisfied.

#### 4.11 Trivial judgements

We now show how some judgements may be verified without neither transforming constraints into linear constraints nor solving any integer programs. To do so, we consider an example provided by Baillot and Ghyselen [3], where we exploit the fact that all coefficients in the normalized constraints are non-positive. Judgements with such constraints can be answered in linear time with respect to the number of monomials in the normalized equivalent of constraint  $C$ . That is if all coefficients in the normalized constraint are non-positive, we can guarantee that the constraint is always satisfied, recalling that only naturals substitute for index variables. Similarly, if there are no negative coefficients and at least one positive coefficient, we can guarantee that the constraint is never satisfied.

In practice, it turns out that we can type check many processes by simply over-approximating constraint judgements using pair-wise coefficient inequality constraints. In Example 5, we show how all constraint judgements in the typings of both a linear and a polynomial time replicated input can be verified using this approach.

*Example 5.* Baillot and Ghyselen [3] provide an example of how their type system for parallel complexity of message-passing processes can be used to bound the time complexity of a linear, a polynomial and an exponential time replicated input process. We show that we can verify all judgements on constraints in the typings of the first two processes using normalized constraints. We first define the processes  $P_1$  and  $P_2$

$$P_i \stackrel{\text{def}}{=} !a(n, r). \text{tick.match } n \{0 \mapsto \bar{r}(); s(m) \mapsto (\nu r')(\nu r'')Q_i\}$$

for the corresponding definitions of  $Q_1$  and  $Q_2$

$$\begin{aligned} Q_1 &\stackrel{\text{def}}{=} \bar{a}(m, r') \mid \bar{a}(m, r'') \mid r'().r''().\bar{r}() \\ Q_2 &\stackrel{\text{def}}{=} \bar{a}(m, r') \mid r'().(\bar{a}(m, r'') \mid \bar{r}()) \mid r''() \end{aligned}$$

We type  $Q_1$  and  $Q_2$  under the respective contexts  $\Gamma_1$  and  $\Gamma_2$

$$\begin{aligned} \Gamma_1 &\stackrel{\text{def}}{=} a : \forall_0 i. \text{oserv}^{i+1}(\text{Nat}[0, i], \text{ch}_{i+1}()), n : \text{Nat}[0, i], m : \text{Nat}[0, i-1], \\ &\quad r : \text{ch}_i(), r' : \text{ch}_i(), r'' : \text{ch}_i() \\ \Gamma_2 &\stackrel{\text{def}}{=} a : \forall_0 i. \text{oserv}^{i^2+3i+2}(\text{Nat}[0, i], \text{ch}_{i+1}()), n : \text{Nat}[0, i], m : \text{Nat}[0, i-1], \\ &\quad r : \text{ch}_i(), r' : \text{ch}_i(), r'' : \text{ch}_{2i-1}() \end{aligned}$$

Note that in the original work, the bound on the complexity of server  $a$  in context  $\Gamma_2$  is  $(i^2 + 3i + 2)/2$ . However, we are forced to use a less precise bound, as the multiplicative inverse is not always defined for our view of indices. Upon typing process  $P_1$ , we amass the judgements on the left-hand side, with corresponding judgements with normalized constraints on the right-hand side

$$\begin{array}{ll}
\{i\}; \emptyset \models i + 1 \geq 1 & \iff \{i\}; \emptyset \models -i \leq 0 \\
\{i\}; \{i \geq 1\} \models i \leq i + 1 & \iff \{i\}; \{1 - i \leq 0\} \models -1 \leq 0 \\
\{i\}; \{i \geq 1\} \models i \geq i & \iff \{i\}; \{1 - i \leq 0\} \models 0 \leq 0 \\
\{i\}; \{i \geq 1\} \models 0 \geq 0 & \iff \{i\}; \{1 - i \leq 0\} \models 0 \leq 0
\end{array}$$

As all coefficients in the normalized constraints are non-positive, each judgement is trivially satisfied, and we can verify the bound  $i + 1$  on server  $a$ . For process  $P_2$  we correspondingly have the trivially satisfied judgements

$$\begin{array}{ll}
\{i\}; \emptyset \models i + 1 \geq 1 & \iff \{i\}; \emptyset \models -i \leq 0 \\
\{i\}; \{0 \leq 0\} \models i \leq i^2 + 3i + 2 & \iff \{i\}; \{0 \leq 0\} \models -i^2 - 2i - 2 \leq 0 \\
\{i\}; \{i \geq 1\} \models i \leq i + 1 & \iff \{i\}; \{1 - i \leq 0\} \models -1 \leq 0 \\
\{i\}; \{i \geq 1\} \models 2i - 1 \leq i^2 + 3i + 2 & \iff \{i\}; \{1 - i \leq 0\} \models -i^2 - i - 3 \leq 0 \\
\{i\}; \{i \geq 1\} \models i \geq i & \iff \{i\}; \{1 - i \leq 0\} \models 0 \leq 0 \\
\{i\}; \{i \geq 1\} \models 0 \geq i^2 + i & \iff \{i\}; \{1 - i \leq 0\} \models -i^2 - i \leq 0 \\
\{i\}; \{i \geq 1\} \models i^2 + 2i \geq i^2 + 3i + 2 & \iff \{i\}; \{1 - i \leq 0\} \models -i - 2 \leq 0
\end{array}$$

#### 4.12 Examples

The expressiveness and accuracy of the type checker depends on the guarantees we can deduce from the structure of processes, and express as sized types for terms and constraints on indices. For instance, the pattern match construct `match  $e$  {0  $\mapsto$   $P$ ;  $s(x)$   $\mapsto$   $Q$ }` provides us guarantees on the size of  $e$  in the subprocesses  $P$  and  $Q$ . This is expressed as constraints  $I \leq 0$  and  $J \geq 0$  in type rule (S-MATCH), where  $e$  is assigned a type  $\text{Nat}[I, J]$ . Additionally, the variable  $x$  bound in the successor pattern must subsequently have a size in the interval  $\text{Nat}[I - 1, J - 1]$ . We can use this information to determine that the recursion simulated by a server is in fact primitive, such that we can guarantee termination, and bound the parallel complexity.

However, these constraints and relationships between sized types are quite simple (and only express constant changes in size), whereas many interesting parallel algorithms use a divide and conquer approach, where the input is divided, rather than *subtracted* from. Thus, we are limited to expressing constant deficits in the size of inputs for such processes, resulting in imprecise complexity bounds for some processes.

In this section, we therefore show how the type checker can be extended with practical constructs that increase its accuracy and expressiveness. Specifically, we work towards typing a parallel merge sort implementation, and so we first extend expressions with the empty list constructor  $[]$  and the *cons* operator  $e_1 :: e_2$  that places  $e_1$  as the head of the list  $e_2$ . We correspondingly extend the language of types, now distinguishing between base types  $\mathcal{B}$  and types

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

The list type is read as follows. The interval describes a lower bound and upper bound on the length of the list, and the base type  $\mathcal{B}$  is the element type, which itself has size bounds. Each element must be typable as a super type of the element type of the list.

We next extend the language of processes with a pattern match constructor for lists  $\text{match } e \{ [] \mapsto P; x :: y \mapsto Q \}$ , a control structure that compares the sizes of two (natural) expressions **if**  $e_1 \leq e_2$  **then**  $P$  **else**  $Q$  and a constructor that splits a list into two equal-sized sublists **split**  $e$  **into**  $(x, y) \mapsto P$ . We introduce type rules accordingly.

$$\text{(S-LMATCH)} \quad \frac{\varphi; \Phi; \Gamma \vdash e : \text{List}[I, J](\mathcal{B}) \quad \varphi; (\Phi, I \leq 0); \Gamma \vdash P \triangleleft \kappa \quad \varphi; (\Phi, J \geq 1); \Gamma, x : \mathcal{B}, y : \text{List}[I - 1, J - 1](\mathcal{B}) \vdash Q \triangleleft \kappa'}{\varphi; \Phi; \Gamma \vdash \text{match } e \{ [] \mapsto P; x :: y \mapsto Q \} \triangleleft \text{basis}(\varphi, \Phi, \kappa \cup \kappa')}$$

Rule (S-LMATCH) types a pattern match on an expression that can be typed as a list. We type the subprocess associated with the empty list pattern under a set of constraints extended with the constraint that the lower bound on the size of the list is 0. When typing the other subprocess, we introduce the constraint that the upper bound on the size of the list is at least 1, as the pattern match construct can only reduce to this subprocess, if the matched list is non-empty. Correspondingly, we extend the type context according to the pattern  $x :: y$ , such that  $x$  is typed as the element type of the matched list, and  $y$  is assigned the type of its tail.

$$\text{(S-SIZE)} \quad \frac{\varphi; \Phi; \Gamma \vdash e_1 : \text{Nat}[I, J] \quad \varphi; \Phi; \Gamma \vdash e_2 : \text{Nat}[K, L] \quad \varphi; (\Phi, J \leq K); \Gamma \vdash P \triangleleft \kappa \quad \varphi; (\Phi, L + 1 \leq I); \Gamma \vdash Q \triangleleft \kappa'}{\varphi; \Phi; \Gamma \vdash \text{if } e_1 \leq e_2 \text{ then } P \text{ else } Q \triangleleft \text{basis}(\varphi, \Phi, \kappa \cup \kappa')}$$

Type rule (S-SIZE) types a control structure that compares the sizes of two expressions that can be typed as naturals. Upon typing the subprocess guarded by the pattern  $e_1 \leq e_2$ , we extend the set of known constraints on index variables with the constraint that the lower bound on the size of  $e_2$  is greater than or equal to the upper bound on the size of  $e_1$ . This rule allows for more accurate bounds

on the parallel complexity of some processes, as it allows us to tighten the size bounds on naturals.

$$(S\text{-SPLIT}) \frac{\varphi; \Phi; \Gamma \vdash e : \text{List}[I, J](\mathcal{B}) \quad \varphi; \Phi; \Gamma, x : \text{List}[I/2, J/2](\mathcal{B}), y : \text{List}[I/2, J/2](\mathcal{B}) \vdash P \triangleleft \kappa}{\varphi; \Phi; \Gamma \vdash \text{split } e \text{ into } (x, y) \mapsto P \triangleleft \kappa}$$

Finally, type rule (S-SPLIT) types the constructor that splits a list expression in two. Here, we enforce that the expression can be typed as a list, and that the continuation can be typed under the same context extended with associations for the variables bound in the constructor, such that they are typed as lists with half the size of the original expression. We are forced to use division to express the change in list size rather than factors due to the *instantiate* function, but these approaches are equivalent for indices with natural coefficients and valuations, as integer division by two is only defined when two is a common factor amongst the terms of an index. Type rule (S-SPLIT) is useful for typing implementations of parallel divide and conquer algorithms where the input is halved, particularly those with parallel complexities that can be expressed as recurrence relations of the form  $T(n) = T(n/2) + C(n)$  where  $C(n)$  is the *local* cost in time complexity of the algorithm.

We now show how the sequential merge function used in merge sort can be encoded in the  $\pi$ -calculus, and subsequently be typed. Note that there exist more sophisticated merge functions that make use of parallelism to provide better complexity bounds. However, these are out of the scope of this thesis. We encode the merge function as a server using a replicated input that receives two lists and a channel  $r$  on which the merged list is *returned*. We use a match construct for lists and the size comparison control structure for naturals to determine the subproblem that is then output to the server.

$$\begin{aligned} P_{\text{merge}} &\stackrel{\text{def}}{=} !\text{merge}(l_1, l_2, r). \text{match } l_1 \{ \\ &\quad [] \mapsto \bar{r}\langle l_2 \rangle; \\ &\quad n :: l'_1 \mapsto \text{match } l_2 \{ \\ &\quad \quad [] \mapsto \bar{r}\langle l_1 \rangle; \\ &\quad \quad m :: l'_2 \mapsto (\nu r' : T) \text{tick. if } n \leq m \\ &\quad \quad \quad \text{then } \overline{\text{merge}}\langle l'_1, l_2, r' \rangle \mid r'(l_3). \bar{r}\langle n :: l_3 \rangle \\ &\quad \quad \quad \text{else } \overline{\text{merge}}\langle l_1, l'_2, r' \rangle \mid r'(l_3). \bar{r}\langle m :: l_3 \rangle \} \} \end{aligned}$$

We use the type contexts, the set of index variables and the set of constraints defined below to type the process. Notably, we assign the server a linear complexity, as we do not make use of parallelism. We omit index variables that are not used in the typing, although more index variables are required to type outputs

on the server, due to the *instantiate* function.

$$\begin{aligned}
\Gamma_{\text{merge}} &\stackrel{\text{def}}{=} \Gamma_{\text{leq}}, \text{merge} : \forall_0 i, j, k, l. \text{serv}_{i+j}^{\{\text{in}, \text{out}\}} \left( \begin{array}{c} \text{List}[0, i](\text{Nat}[k, l]), \\ \text{List}[0, j](\text{Nat}[k, l]), \\ \text{ch}_{i+j}^{\{\text{out}\}}(\text{List}[0, i+j](\text{Nat}[k, l])) \end{array} \right) \\
\Delta &\stackrel{\text{def}}{=} \Gamma_{\text{merge}}, l_1 : \text{List}[0, i](\text{Nat}[k, l]), l_2 : \text{List}[0, j](\text{Nat}[k, l]), \\
&\quad r : \text{ch}_{i+j}^{\{\text{out}\}}(\text{List}[0, i+j](\text{Nat}[k, l])), r' : \text{ch}_{i+j}^{\{\text{out}\}}(\text{List}[0, i+j-1](\text{Nat}[k, l])) \\
\varphi &\stackrel{\text{def}}{=} \cdot, i, j, k, l \quad \Phi \stackrel{\text{def}}{=} \cdot, i \geq 1, j \geq 1
\end{aligned}$$

We use the following definition for the type annotation on the restriction

$$T \stackrel{\text{def}}{=} \text{ch}_{i+j-1}^{\{\text{out}\}}(\text{List}[0, i+j-1](\text{Nat}[k, l]))$$

We now show that the process is typable under context  $\Gamma_{\text{merge}}$ . We omit some of the branches in the derivation tree for conciseness, and we do not show the typing of the two match constructors, as these are trivial. The main points of interest are the outputs to the server where we use the *instantiate* function to construct substitutions. Note that we omit superfluous substitutions, i.e. those that have no effect on indices.

We are now ready to encode a parallel merge sort in the  $\pi$ -calculus. We use a replicated input that receives a list of naturals and a channel on which the sorted list of the same size is *returned*. We use the **split into** constructor to divide the list in two, with corresponding divided list size bounds, such that the two subproblems are half the size of the original problem.

$$\begin{aligned}
P_{\text{sort}} &\stackrel{\text{def}}{=} !\text{sort}(l, r). \text{split } l \text{ into } (l_1, l_2) \mapsto (\nu r_1 : T_1)(\nu r_2 : T_2)(\overline{\text{sort}}(l_1, r_1) \mid \overline{\text{sort}}(l_2, r_2) \mid r_1(l'_1). \\
&\quad r_2(l'_2).(\nu r_3 : T_3)(\overline{\text{merge}}(l'_1, l'_2, r_3) \mid r_3(l').\overline{r}(l')))
\end{aligned}$$

We type the process under the contexts and the set of index variables defined below. We assign the server a linear bound of  $2i$  where  $i$  is the upper bound on the size of the input list, as the two parallel subproblems then have bounds of  $i$ , the results of which are merged in  $2i$  time using the merge function encoding. This is quite a tight bound on the parallel complexity of the parallel merge sort encoding, as its pen and paper parallel complexity can be expressed as the recurrence relation  $T(n) = T(n/2) + n$ , the solution of which approaches  $2n$  as the size  $n$  of the list approaches infinity.

$$\begin{aligned}
\Gamma_{\text{sort}} &\stackrel{\text{def}}{=} \Gamma_{\text{merge}}, \text{sort} : \forall_0 i, k, l. \text{serv}_{2i}^{\{\text{in}, \text{out}\}}(\text{List}[0, i](\text{Nat}[k, l]), \text{ch}_{2i}^{\{\text{out}\}}(\text{List}[0, i](\text{Nat}[k, l]))) \\
\Delta &\stackrel{\text{def}}{=} \Gamma_{\text{sort}}, l_1 : \text{List}[0, i/2](\text{Nat}[k, l]), l_2 : \text{List}[0, i/2](\text{Nat}[k, l]), r_1 : T_1, r_2, T_2 \\
\Delta' &\stackrel{\text{def}}{=} \downarrow_i \Delta, l'_1 : \text{List}[0, i/2](\text{Nat}[k, l]), l'_2 : \text{List}[0, i/2](\text{Nat}[k, l]) \\
\Delta'' &\stackrel{\text{def}}{=} \downarrow_i \Delta', l' : \text{List}[0, i](\text{Nat}[k, l]) \\
\varphi &\stackrel{\text{def}}{=} \cdot, i, k, l
\end{aligned}$$

We use the following definitions for the type annotations on restrictions

$$\begin{aligned} T_1 &\stackrel{\text{def}}{=} \text{ch}_i^{\{\text{out}\}}(\text{List}[0, i/2](\text{Nat}[k, l])) \\ T_2 &\stackrel{\text{def}}{=} \text{ch}_i^{\{\text{out}\}}(\text{List}[0, i/2](\text{Nat}[k, l])) \\ T_3 &\stackrel{\text{def}}{=} \text{ch}_i^{\{\text{out}\}}(\text{List}[0, i](\text{Nat}[k, l])) \end{aligned}$$

Finally, we show that the process is in fact typeable under type context  $\Gamma_{\text{sort}}$ . The type derivation tree is shown below. We again omit some of the more trivial steps of the typing and simplify the substitutions computed using the *instantiate* function for conciseness.

It is possible to introduce more interesting constructors that increase the expressiveness and precision of encodings of various parallel algorithms, by introduction of constraints and by modifying sizes of terms. Such constructors are useful because of one general limitation to this work on sized types for parallel complexity. That is, we cannot express relationships between the sizes of multiple message types of channels and servers. Consider for instance the partition function used in the *quicksort* algorithm to partition a list on a pivot based on the  $\leq$  relation. We would like to encode the function as a server of the form

$$!\text{partition}(l, p, r).P$$

where  $l$  is the list to be partitioned,  $p$  is the pivot element and  $r$  is a channel on which the partitioned list is returned as two lists, whose upper bounds on sizes sum to the upper bound on the size of  $l$ . That is, we are interested in the type

$$\forall_0 i, j, k. \text{server}_j^{\{\text{in}, \text{out}\}}(\text{List}[0, i](\mathcal{B}), \mathcal{B}, \text{ch}_j^{\{\text{out}\}}(\text{List}[0, j](\mathcal{B}), \text{List}[0, k](\mathcal{B})))$$

such that  $j + k = i$ . However, we cannot guarantee this constraint based on the typing above, and so we cannot verify the bound  $j$ .

## 5 Inference of sized types for parallel complexity

In this section, we address type inference for the type system introduced in Section ???. The main idea is to perform inference over several steps, starting with inference of simple types, and annotation of server types with index variables. We then perform a second pass over the process, inferring constraints on use-capabilities and indices. For unknown indices, we introduce *templates* based on the available index variables at any point in the derivation. That is, we restrict indices to be linear functions over index variables with coefficient variables in place of coefficients as used in [12,10,9]. The result should be a triple  $(\Gamma, \mathcal{C}, K)$  of a type context, a set of type variable constraints and an index, such that when provided a solution  $\theta$  to  $\mathcal{C}$  (a substitution of naturals for type variables), we have that  $\varphi; \cdot; \theta\Gamma \vdash P \triangleleft \theta K$  where  $P$  is the original process,  $\varphi$  is the set of free index variables in  $\Gamma$ . Here,  $\theta\Gamma$  and  $\theta K$  denote the application of substitution  $\theta$  to  $\Gamma$  and  $K$ , respectively. Note that this requires  $P$  to be closed with respect to index



variable constraints. We are interested in a secondary property: if  $P$  is well-typed, then  $K$  is the least bound on the span of  $P$ , in the sense that there then exists a solution  $\theta$  to  $\mathcal{C}$  such that  $\theta K$  is the least upper bound on the span of  $P$ . Later, we consider the more challenging problem of solving the kind of constraint satisfaction problem that we infer, and present a Haskell implementation. The whole chapter is inspired by inference techniques for usage types [17,16].

### 5.1 The principal typing property

In this section, we formalize the approach outlined above. We first modify the type rules from Chapter ?? to make the type system more suitable for type inference. As we intend to infer types bottom-up, we replace advancement of type by delaying of time, as introduced in Section 4.2. For instance, the type rule for inputs on channels becomes

$$(\text{BG-ICH}') \frac{\text{in} \in \sigma \quad \varphi; \Phi; \Gamma, a : \text{ch}_0^\sigma(\tilde{T}), \tilde{v} : \tilde{T} \vdash P \triangleleft K}{\varphi; \Phi; \uparrow^I \Gamma, a : \text{ch}_I^\sigma(\tilde{T}) \vdash a(\tilde{v}).P \triangleleft K + I}$$

Note that the rule (BG-ISERV) must be treated differently, due to time invariance, which is not compatible with delaying. Therefore, we do not modify this rule, and instead handle this in our inference algorithm. We now introduce types without indices and use-capabilities, referring to these as *simple* types. Inference of such types is straightforward, and they are useful for later inference of indices and use-capabilities. We distinguish simple types ( $\tilde{T}$  and  $\tilde{S}$  as metavariables) and types ( $T$  and  $S$  as metavariables). We also enrich the definition of types with type variables and restrict indices to be linear functions of index variables

$$\begin{aligned} \tilde{T}, \tilde{S} &::= \beta \mid \text{Nat} \mid \text{ch}(\tilde{T}) \mid \forall \tilde{i}. \text{serv}(\tilde{T}) \\ T, S &::= \text{Nat}[I, J] \mid \text{ch}_I^\sigma(\tilde{T}) \mid \forall_I \tilde{i}. \text{serv}_K^\sigma(\tilde{T}) \\ I, J, K, L &::= \mathbf{c}i + I \mid \mathbf{c} \\ \sigma &::= \gamma \mid \{\text{in}\} \mid \{\text{out}\} \mid \{\text{in}, \text{out}\} \\ \mathbf{c} &::= \alpha \mid n \mid \mathbf{c} + \mathbf{c}' \mid \mathbf{c}\mathbf{c}' \end{aligned}$$

where  $\beta$  is a type variable for simple types,  $\mathbf{c}$  is a coefficient,  $\alpha$  is a coefficient variable,  $n$  is a numeric constant and  $\gamma$  is a type variable for use-capabilities. Note that although indices are linear in index variables, coefficients may be polynomial in coefficient variables. This is due to type rule (BG-OSERV) in Table 5 where we require a substitution of indices for index variables quantified in a server type. Moreover, to accommodate these substitutions as well as time invariance, we allow for negative valuations of coefficient variables.

We write  $\text{ftv}(\Gamma)$ ,  $\text{ftv}(\mathcal{C})$  and  $\text{ftv}(I)$  for the set of free type variables in a set of index variable constraints, a type context, a set of type variable constraints or an

index, respectively. For a substitution of types and naturals for type variables, we use the metavariable  $\theta$ . Moreover, we write  $\theta\Gamma$ ,  $\theta\mathcal{C}$  and  $\theta I$  to apply a type variable substitution.

We next define the language of constraints. We first distinguish constraints for simple types, as such constraints must be inferred and solved before other kinds of constraints can be considered. The constraint  $\dot{T} \sim \dot{S}$  enforces that  $\dot{T}$  and  $\dot{S}$  must be the same simple type. We write  $?( \tilde{T} )$  for a channel or server type that has message types  $\tilde{T}$ , which is needed for outputs, as we cannot determine whether a channel or server is required from an output alone. We write  $\mathcal{C}_\beta$  for a set of simple type constraints.

$$\begin{aligned} c_\beta &::= \dot{T} \sim \dot{S} \mid \dot{T} \sim ?(\tilde{S}) \\ c_T &::= T \sim S \mid \varphi; \Phi \vdash \text{inv}(T) \mid \tilde{c}_\gamma \implies (\varphi; \Phi \vdash T \sqsubseteq S) \mid c_{IO} \\ c_{IO} &::= \tilde{c}_\gamma \implies (\varphi; \Phi \models I \leq J) \mid \tilde{c}_\gamma \implies c'_\gamma \mid c_I \\ c_I &::= I \sim J \mid \varphi; \Phi \models I \leq J \mid \text{false} \\ c_\gamma &::= \sigma \subseteq \sigma' \end{aligned}$$

The remaining constraint language is defined in steps, such that a *type constraint*  $c_T$  may be a use-constraint  $c_{IO}$ , which in turn may be an *index constraint*  $c_I$ . We make this distinction, as we are interested in reducing a set of type constraints to a set of *use-constraints*, and finally a set of use-constraints to a set of index constraints that we can attempt to solve. Note the distinction between constraints on index variables, as we have seen in constraint judgements previously, and constraints on indices. In the remainder of this thesis, we refer to the former as index variable constraints.

Type constraints may contain types. We have type equality constraints  $T \sim S$ , time invariance constraints  $\varphi; \Phi \vdash \text{inv}(T)$  and conditional subsumption constraints  $\tilde{c}_\gamma \implies (\varphi; \Phi \vdash T \sqsubseteq S)$ . The latter is read as a Horn-clause, where the antecedent  $\tilde{c}_\gamma$  is a sequence of inclusion use-constraints. We may write  $\varphi; \Phi \vdash T \sqsubseteq S$  if the antecedent is empty. For equivalence constraints on types, we may write  $\Gamma \sim \Delta$  to denote the set  $\{\Gamma(v) \sim \Delta(v) \mid v \in \text{dom}(\Gamma) \cap \text{dom}(\Delta)\}$ . Similarly, we write  $\varphi; \Phi \vdash \Gamma \sqsubseteq \Delta$  for the set  $\{\varphi; \Phi \vdash \Gamma(v) \sqsubseteq \Delta(v) \mid v \in \text{dom}(\Gamma) \cap \text{dom}(\Delta)\}$ . Finally, for a set of indices  $\mathbf{I}$  we write  $\varphi; \Phi \models \mathbf{I} \leq J$  for the set  $\{\varphi; \Phi \models I \leq J \mid I \in \mathbf{I}\}$ .

Use-constraints may contain inclusion use-constraints and conditions. Specifically, we have conditional index inequalities and conditional inclusion use-constraints. Such constraints are introduced upon reduction of conditional subsumption constraints, as we shall later see. Finally, index constraints may be coefficient equivalence constraints or non-conditional index inequality constraints. We include the constraint **false** here, as we may declare a process ill-typed at any point.

Note that some constraints require knowledge about a set of index variables and a set of index variable constraints. This is problematic, as such constraints

are then existentially and universally quantified, implying that they are often not solvable using existing constraint satisfaction solvers. That is, coefficient variables are existentially quantified and index variables are universally quantified. However, we can reduce type constraints to constraints on indices, which can be over-approximated by multiple constraints on coefficients that do not involve universal quantifiers. We return to these topics in Section 6 and 6.3.

In Definition 21 we introduce an operation on type constraints that extends sets of constraints on index variables where applicable. This is useful for the match constructor, for which constraints inferred from the two subprocesses should be extended with constraints regarding the size of the pattern match expression.

**Definition 21 (Constraint extension).** *We define an operator  $C \mathbb{M}_{c_T}$  on type constraints that extends the set of index variable constraints in a type constraint  $c_T$  with the index constraint  $C$ .*

$$\begin{aligned}
C \mathbb{M} T \sim S &\stackrel{\text{def}}{=} T \sim S \\
C \mathbb{M} I \sim J &\stackrel{\text{def}}{=} I \sim J \\
C \mathbb{M} \varphi; \Phi \vdash \text{inv}(T) &\stackrel{\text{def}}{=} \varphi; (\Phi, C) \vdash \text{inv}(T) \\
C \mathbb{M} \tilde{c}_\gamma \implies (\varphi; \Phi \vdash T \sqsubseteq S) &\stackrel{\text{def}}{=} \tilde{c}_\gamma \implies (\varphi; (\Phi, C) \vdash T \sqsubseteq S) \\
C \mathbb{M} \tilde{c}_\gamma \implies (\varphi; \Phi \vdash I \leq J) &\stackrel{\text{def}}{=} \tilde{c}_\gamma \implies (\varphi; (\Phi, C) \vdash I \leq J) \\
C \mathbb{M} \tilde{c}_\gamma \implies c'_\gamma &\stackrel{\text{def}}{=} \tilde{c}_\gamma \implies c'_\gamma \\
C \mathbb{M} \mathbf{c} \sim \mathbf{c}' &\stackrel{\text{def}}{=} \mathbf{c} \sim \mathbf{c}' \\
C \mathbb{M} \varphi; \Phi \models I \leq J &\stackrel{\text{def}}{=} \varphi; (\Phi, C) \models I \leq J \\
C \mathbb{M} \text{false} &\stackrel{\text{def}}{=} \text{false}
\end{aligned}$$

We extend the operator to sets of type constraints  $\mathcal{C}_T$

$$C \mathbb{M} \mathcal{C}_T \stackrel{\text{def}}{=} \{C \mathbb{M} c_T \mid c_T \in \mathcal{C}_T\}$$

To enforce time invariance, we constrain the time-tags of servers. For convenience, we therefore define a function  $\text{tag}(T)$  in Definition 22 that returns the time-tag of an argument channel or server  $T$ .

**Definition 22.** *We define a function on types  $\text{tag}(T)$  that returns the time-tag of a channel or server*

$$\begin{aligned}
\text{tag}(\text{ch}_I^\sigma(\tilde{T})) &= I \\
\text{tag}(\forall_I \tilde{i}. \text{serv}_K^\sigma(\tilde{T})) &= I
\end{aligned}$$

We extend the function to type contexts, such that  $\text{tag}(\Gamma) = \{I \mid v \in \text{dom}(\Gamma) \text{ and } \text{tag}(\Gamma(v)) = I\}$ .

We are now ready to formalize the triple discussed briefly above. We refer to this as a *principal typing* as seen in Definition 23. Note that a principal typing does not guarantee that a process is well-typed. Rather, it provides a constraint satisfaction problem, whose solutions provide bounds on the span. Moreover, there may be multiple possible solutions, with different implications to the bound on the span.

**Definition 23 (Principal typing).** *A triple  $(\Gamma, \mathcal{C}, K)$  of a type context, a set of type constraints and an index  $K$  is a principal typing of a closed process  $P$ , if it satisfies the properties*

1. *If provided a substitution  $\theta$  with  $\text{dom}(\theta) \subseteq \text{ftv}(\Gamma) \cup \text{ftv}(\mathcal{C}) \cup \text{ftv}(K)$  and  $\theta\mathcal{C}$  satisfied then there exists  $\varphi$  such that  $\varphi; \cdot; \theta\Gamma \vdash P \triangleleft \theta K$ .*
2. *If  $\varphi; \cdot; \Gamma' \vdash P \triangleleft K'$  then there exists a substitution  $\theta$  such that  $\theta\mathcal{C}$  is satisfied and  $\varphi; \cdot \models K' \leq \theta K$ .*

## 5.2 Inference of simple types

We are interested in an inference algorithm that only returns principal typings, as it then suffices to solve the corresponding constraint satisfaction problem. We first show how simple types can be inferred, as a foundation for the more novel inference phases. As this inference phase is quite straightforward, we only present our approach using an example. We consider a server that performs an action  $n$  times in parallel, where  $n$  is a parameter. Each action has a cost of 1 in time complexity, and so we should be able to assign a constant bound to the server. The whole process is shown below, annotated with simple type variables.

$$\begin{aligned}
P_{\text{par}} &\stackrel{\text{def}}{=} \\
&(\nu \text{par} : \beta_1)( \\
&\quad !\text{par}(n, r).\text{match } n \{ \\
&\quad \quad 0 \mapsto \bar{r}\langle \rangle \\
&\quad \quad s(x) \mapsto (\nu r' : \beta_2)(\nu r'' : \beta_3)(\text{tick}.\bar{r}'\langle \rangle \mid \overline{\text{par}}\langle x, r'' \rangle \mid r'().r''().\bar{r}\langle \rangle) \} \\
&\quad | \\
&\quad (\nu r : \beta_4)(\overline{\text{par}}\langle s(0) \rangle, r \rangle \mid r().\mathbf{0})
\end{aligned}$$

We then traverse the process tree bottom-up and infer constraints on simple type variables. We solve the resulting constraint satisfaction problem yielding us a substitution of simple types for simple type variables. We assign each server type a fixed number of index variables as a heuristic. If the later inference phases do not yield satisfying results, we may increase the number of index variables. We opt to use a single index variable  $i$ , and so we obtain the following constraint satisfaction problem for the process above

$$\begin{aligned}
\beta_1 &\sim \forall i. \text{serv}(\beta_5, \beta_6) & \beta_5 &\sim \text{Nat} & \beta_6 &\sim ?() \\
\beta_2 &\sim \text{ch}() & \beta_1 &\sim \text{ch}(\text{Nat}, \beta_3) & \beta_3 &\sim \text{ch}() \\
\beta_1 &\sim ?(\text{Nat}, \beta_3) & \beta_1 &\sim ?(\text{Nat}, \beta_4)
\end{aligned}$$

For outputs, we cannot determine whether a channel or server type is required, and so we introduce the constraint  $?(T)$  that specifies that either of the two is acceptable. We solve the constraints by substitution of simple types for simple type variables, and so we get the following solution

$$[\beta_1 \mapsto \forall i.\text{serv}(\text{Nat}, \text{ch}()), \beta_2 \mapsto \text{ch}(), \beta_3 \mapsto \text{ch}(), \beta_4 \mapsto \text{ch}()]$$

Finally, by application of the substitution to simple type variable annotations in the process, we get

$$\begin{aligned} &(\nu \text{npair} : \forall i.\text{serv}(\text{Nat}, \text{ch}()))( \\ &\quad !\text{npair}(n, r).\text{match } n \{ \\ &\quad \quad 0 \mapsto \bar{r}() \\ &\quad \quad s(x) \mapsto (\nu r' : \text{ch}())(\nu r'' : \text{ch}())(\text{tick}.\bar{r}'() \mid \overline{\text{npair}}(x, r'') \mid r'().r''().\bar{r}()) \} \\ &\quad | \\ &\quad (\nu r : \text{ch}())(\overline{\text{npair}}(s(0), r) \mid r().0) \end{aligned}$$

### 5.3 Inference of constraints on use-capabilities and coefficients

We now show how a triple  $(\Gamma, \mathcal{C}, K)$  can be inferred when provided information about simple types. We present our approach as a collection of algorithmic inference rules, based on the premises of the augmented type rules. Judgements for expressions are of the form  $\varphi; \dot{\Gamma} \vdash_{\mathcal{C}} e : (\Gamma, T)$  and are read as *provided a set of index variables and a context of simple types, expression  $e$  can be assigned type  $\theta T$  using context  $\theta \Gamma$  for any substitution  $\theta$  such that  $\theta \mathcal{C}$  is satisfied*. Similarly, we have judgements for processes of the form  $\varphi; \dot{\Gamma} \vdash_{\mathcal{C}} P \triangleleft (\Gamma, K)$  that are read as *provided a set of index variables and a context of simple types, process  $P$  can be assigned complexity bound  $\theta K$  using context  $\theta \Gamma$  for any substitution  $\theta$  such that  $\theta \mathcal{C}$  is satisfied*.

We first address inference of indices. Recall that indices are restricted to linear functions over index variables, which allows us to introduce *templates* for indices. That is, provided a set of known index variables (We obtain this directly from simple types), we represent an unknown index as a coefficient variable representing a constant term plus a sum of the index variables each scaled by a unique coefficient variable. For instance, given two index variables  $i$  and  $j$ , we have the template  $\alpha_0 + \alpha_1 i + \alpha_2 j$  where  $\alpha_0$ ,  $\alpha_1$  and  $\alpha_2$  are coefficient variables. Templates are widely used, as they simplify constraint generation for coefficient variables [12,10,9]. For convenience, we introduce an algorithm *fresh* that granted a set of index variables provides a template with all coefficient variables fresh. We extend the function to simple types, returning a corresponding type with fresh templates. Here, we also introduce use-capability variables for channel and server types. The function is shown in Definition 24

**Definition 24 (Fresh variables).** *We write  $\alpha$  fresh and  $\gamma$  fresh to denote that  $\alpha$  and  $\gamma$ , respectively, are new unique variables. We define an algorithm *fresh* that*

provided a set of index variables returns a template with all coefficient variables fresh

$$\text{fresh}(\{i_1, \dots, i_n\}) = \alpha_0 + \sum_{1 \leq j \leq n} \alpha_j i_j \quad \text{with } \alpha_0, \alpha_1, \dots, \alpha_n \text{ fresh}$$

We extend the definition to types

$$\begin{aligned} \text{fresh}(\text{Nat}, \varphi) &= \text{Nat}[I, J] \quad \text{with } I, J = \text{fresh}(\varphi) \\ \text{fresh}(\text{ch}(\tilde{T}), \varphi) &= \text{ch}_I^\gamma(\tilde{T}) \quad \text{with } \gamma \text{ fresh}, I = \text{fresh}(\varphi) \text{ and } \tilde{T} = \text{fresh}(\tilde{T}, \varphi) \\ \text{fresh}(\forall i. \text{serv}(\tilde{T})) &= \forall i. \text{serv}_K^\gamma(\tilde{T}) \quad \text{with } \gamma \text{ fresh}, I = \text{fresh}(\varphi), K = \text{fresh}((\varphi, i)) \text{ and } \tilde{T} = \text{fresh}(\tilde{T}, (\varphi, i)) \end{aligned}$$

We introduce a variant of the algorithm that replaces the time-tags of channels and servers with fresh templates

$$\begin{aligned} \text{freshtag}(\text{Nat}[I, J], \varphi) &= \text{Nat}[I, J] \\ \text{freshtag}(\text{ch}_I^\sigma(\tilde{T}), \varphi) &= \text{ch}_J^\sigma(\tilde{T}) \quad \text{with } J = \text{fresh}(\varphi) \\ \text{freshtag}(\forall i. \text{serv}_K^\sigma(\tilde{T}), \varphi) &= \forall i. \text{serv}_K^\sigma(\tilde{T}) \quad \text{with } J = \text{fresh}(\varphi) \end{aligned}$$

We extend the definitions to type contexts, such that if  $\text{dom}(\Gamma) = \{v_1, \dots, v_n\}$  then  $\text{fresh}(\Gamma, \varphi) = v_1 : \text{fresh}(\Gamma(v_1), \varphi), \dots, v_n : \text{fresh}(\Gamma(v_n), \varphi)$ . Moreover, for a sequence of simple types  $\tilde{T}$  we write  $\tilde{T} = \text{fresh}(\tilde{T}, \varphi)$  to denote  $T = \text{fresh}(\tilde{T}, \varphi)$  for each  $\tilde{T}$  in the sequence  $\tilde{T}$ .

We are now ready to introduce the inference rules. We implicitly introduce constraints of the form  $\varphi; \cdot \models 0 \leq I$  for all indices  $I$  as both complexity and size must be non-negative. The rules for expressions are shown in Table 7. Note that we do not include an inference rule for subtyping of expressions, as the subsumption rule is not algorithmic. Instead, we incorporate subtyping into the remaining rules. For the 0-constructor we allow for an arbitrary non-negative bound on the span to accommodate subtyping. For variables, we assign a fresh instantiation of the simple type it is bound to. We implicitly introduce constraints for all occurrences of the natural type constructor in the type assigned to a variable that ensure upper bounds cannot be smaller than lower bounds. Finally, for the successor constructor  $s(e)$ , we introduce a constraint that enables subtyping and enforces a size increase of 1.

We now consider the inference rules of processes, which are shown in Table 8 and 9. Inference rule (I-NU) infers a triple  $(\Gamma, \mathcal{C}, K)$  for a restriction. We discard the name bound in the restriction from the inferred type context. For parallel compositions, we use rule (I-PAR) where we infer a tuple for each subprocess and constrain the types of names bound in both subsequent contexts to be equivalent, as per type rule (BG-PAR) in Table 5. Similarly, we constrain the two complexity bounds to be equivalent, which is a reasonable constraint due to

(I-ZERO)	$\frac{}{\varphi; \dot{T} \vdash_{\emptyset} 0 : (\cdot, \mathbf{Nat}[0, J])}$	where $J = \text{fresh}(\varphi)$
(I-VAR)	$\frac{}{\varphi; \dot{T}, v : \dot{T} \vdash_{\emptyset} v : ((\cdot, v : T), T)}$	where $T = \text{fresh}(\dot{T}, \varphi)$
(I-SUCC)	$\frac{\varphi; \dot{T} \vdash_c e : (I, \mathbf{Nat}[I, J])}{\varphi; \dot{T} \vdash_{\mathcal{C} \cup \{\varphi; \vdash \mathbf{Nat}[I+1, J+1] \sqsubseteq \mathbf{Nat}[I', J']\}} s(e) : (I, \mathbf{Nat}[I', J'])}$	where $I', J' = \text{fresh}(\varphi)$

**Table 7.** Inference rules for expressions.

subtyping in other inference rules.

Rule (I-TICK) infers a tuple for a tick constructor. Here, we use the delaying operator from Chapter 4 to add 1 to each time tag in the context inferred for the continuation. We introduce a fresh template for the complexity and constrain it to be greater than or equal to the complexity bound inferred for the continuation plus 1. We use rule (I-MATCH) for the match constructor. Here, the type inferred for the pattern match expression must be a natural type, and for the two subprocesses, we extend the inferred constraints with size information about the expression. That is, for the subprocess guarded by the zero pattern, we can guarantee that the lower-bound on the size of the expression is 0, and for the other subprocess, the upper-bound must be at least 1. As we do for (I-PAR), we constrain the subsequent complexity bounds to be equivalent, and the contexts to be equivalent for shared names. Finally, if the variable bound in the successor pattern is free in the corresponding subprocess, we constrain its size bounds according to the type of the pattern match expression.

We next consider the inference rules for inputs and outputs. Rule (I-SERV) infers a triple  $(I, \mathcal{C}, K)$  for a replicated input. We infer a tuple for the continuation, and delay the subsequent type context according to the fresh time tag of the server, and discard associations for the names bound in the replicated input. Moreover, we introduce time invariance constraints for all names that are free in the continuation and unbound in the input. Time invariance constraints do not account for the time-tags of servers that must be zero within the continuation of the replicated input, yet may still be advanced. We enable this by introducing fresh templates for the time-tags of the servers that are free in the continuation, which we then constrain to be less than or equal to the time-tag of the server of the replicated input. This way, we essentially discard the original time-tags of the time invariant servers, which is sound as these must be non-negative (by our implicit constraints on all indices). We use inference rule (I-OSERV) and

(I-NU) $\frac{\varphi; \dot{I}, a : \dot{T} \vdash_c P \triangleleft (\Gamma, K)}{\varphi; \dot{I} \vdash_c (\nu a : \dot{T})P \triangleleft (\Gamma \setminus \{a\}, K)}$	(I-TICK) $\frac{\varphi; \dot{I} \vdash_c P \triangleleft (\Gamma, K)}{\varphi; \dot{I} \vdash_{\mathcal{C} \cup \{\varphi; \models K+1 \leq K'\}} \mathbf{tick}.P \triangleleft (\uparrow^1 \Gamma, K')}$ with $K' = \text{fresh}(\varphi)$
(I-PAR) $\frac{\varphi; \dot{I} \vdash_{c_1} P \triangleleft (\Gamma, K) \quad \varphi; \dot{I} \vdash_{c_2} Q \triangleleft (\Delta, K')}{\varphi; \dot{I} \vdash_{c_1 \cup c_2 \cup \Gamma \sim \Delta \cup \{K \sim K'\}} P \mid Q \triangleleft ((\Gamma, \Delta), K)}$	(I-NIL) $\frac{}{\varphi; \dot{I} \vdash_{\{\varphi; \models 0 \leq K\}} \mathbf{0} \triangleleft (\cdot, K)}$ with $K = \text{fresh}(\varphi)$
(I-MATCH) $\frac{\varphi; \dot{I} \vdash_{c_1} e : (\Delta, \mathbf{Nat}[I, J]) \quad \varphi; \dot{I} \vdash_{c_2} P \triangleleft (\Gamma, K) \quad \varphi; \dot{I}, x : \mathbf{Nat} \vdash_{c_3} Q \triangleleft (\Gamma', K')}{\varphi; \dot{I} \vdash_{c_1 \cup (I=0 \wedge c_2) \cup (J \geq 1 \wedge c_3) \cup c_4 \cup c_5} \mathbf{match} \ e \ \{0 \mapsto P; \ s(x) \mapsto Q\} \triangleleft (\Delta, (\Gamma, \Gamma') \setminus \{x\}, K)}$ where $\mathcal{C}_4 = \begin{cases} \{\varphi; \cdot \vdash \mathbf{Nat}[I, J] \sqsubseteq \mathbf{Nat}[J' + 1, I' + 1]\} & \text{if } \Gamma'(x) = \mathbf{Nat}[I', J'] \\ \emptyset & \text{if } x \notin \text{dom}(\Gamma') \end{cases}$ $\mathcal{C}_5 = \Gamma \sim \Gamma' \sim \Delta \cup \{K \sim K'\}$	

**Table 8.** Process time inference rules (1).

(I-och) for outputs. We distinguish between outputs on servers and channels using the simple type context. For the former, we introduce constraints that ensure the expressions to be output are *instantiations* of the message types of the server. That is, there must exist a substitution  $\{\tilde{J}/\tilde{i}\}$  of indices  $\tilde{J}$  for index variables  $\tilde{i}$ . Note that this is where polynomial coefficients are necessary, and so by over-approximation of substitutions, we can simplify the resulting constraint satisfaction problems if necessary. The rules for normal channels are similar, and so we do not consider them in detail. Example 6 shows some of the constraints inferred for a process.

*Example 6.* We show the constraints inferred for a part of the process introduced in Section 5.2. We assume a simple type context  $\cdot, \text{npar} : \forall i. \mathbf{serv}(\mathbf{Nat}, \mathbf{ch}())$ . We consider the following part of the process

$$(\nu r : \mathbf{ch}())(\overline{\text{npar}}(s(s(0)), r) \mid r().\mathbf{0})$$

We infer the tuple below for the process. In the following section, we show how the constraint satisfaction problem can be reduced to constraints of the forms  $\varphi; \Phi \models I \leq J$  and  $I \sim J$ .

$$((\cdot, \text{npar} : \forall_{I_4} i. \mathbf{serv}_{K_1}^{\gamma_4}(\mathbf{Nat}[J_{12}, J_{13}], \mathbf{ch}_{I_1}^{\gamma_1}()), \mathcal{C}, K_2)$$



(I-SERV)	$\frac{\varphi, \tilde{i}; \dot{I}, \tilde{v} : \tilde{T} \vdash_{c_1} P \triangleleft (\Gamma, K)}{\varphi; \dot{I}, a : \forall \tilde{i}. \mathbf{serv}(\tilde{T}) \vdash_{c_1 \cup c_2 \cup c_3 \cup c_4} !a(\tilde{v}).P \triangleleft ((\Delta, a : \forall \tilde{i}. \mathbf{serv}_{K''}^\gamma(\tilde{T})), K')}$ <p>where <math>\forall \tilde{i}. \mathbf{serv}_{K''}^\gamma(\tilde{T}) = \text{fresh}(\forall \tilde{i}. \mathbf{serv}(\tilde{T}), \varphi)</math>      <math>K' = \text{fresh}(\varphi)</math></p> <p><math>\mathcal{C}_2 = \{\varphi; \cdot \models \mathbf{inv}(\Gamma(w)) \mid w \in \text{dom}(\Gamma \setminus \{\tilde{v}\})\}</math></p> <p><math>\mathcal{C}_3 = \{\{\mathbf{in}\} \subseteq \gamma, K \sim K'', \varphi; \cdot \models I \leq K', (\varphi, \tilde{i}); \cdot \vdash (\cdot, \tilde{v} : \tilde{T}) \sqsubseteq \Gamma\}</math></p> <p><math>\Delta = \text{fresh}(\dot{I} \setminus \{a\}, \varphi)</math></p> <p><math>\mathcal{C}_4 = \{\varphi; \cdot \models \text{tag}(\Delta) \leq I, \varphi; \cdot \vdash \Delta, a : \forall \tilde{i}. \mathbf{serv}_{K''}^\gamma(\tilde{T}) \sqsubseteq \text{freshtag}(\Gamma, \varphi)\}</math></p>
(I-OSERV)	$\frac{\varphi; \dot{I} \vdash^{(1 \leq j \leq n)} e_j : (\Gamma_j, S_j)}{\varphi; \dot{I}, a : \forall \tilde{i}. \mathbf{serv}(\dot{I}_1, \dots, \dot{I}_n) \vdash_{c_1 \cup c_2 \cup c_3} \overline{a}(e_1, \dots, e_n) \triangleleft (\Delta, K)}$ <p>where <math>\forall \tilde{i}. \mathbf{serv}_{K'}^\gamma(T_1, \dots, T_n) = \text{fresh}(\forall \tilde{i}. \mathbf{serv}(\tilde{T}), \varphi)</math></p> <p><math>\Delta = \uparrow^I(\Gamma_1, \dots, \Gamma_n), a : \forall \tilde{i}. \mathbf{serv}_{K'}^\gamma(T_1, \dots, T_n)</math></p> <p><math>K, \tilde{J} = \text{fresh}(\varphi)</math></p> <p><math>\mathcal{C}_1 = \{\varphi; \cdot \vdash S_j \sqsubseteq T_j\{\tilde{J}/\tilde{i}\} \mid 1 \leq j \leq n\}</math></p> <p><math>\mathcal{C}_2 = \Gamma_1 \sim \dots \sim \Gamma_n</math></p> <p><math>\mathcal{C}_3 = \{\varphi; \cdot \models I + K'\{\tilde{J}/\tilde{i}\} \leq K, \{\mathbf{out}\} \subseteq \gamma\}</math></p>
(I-ICH)	$\frac{\varphi; \dot{I}, a : \mathbf{ch}(\tilde{T}), \tilde{v} : \tilde{T} \vdash_{c_1} P \triangleleft (\Gamma, K)}{\varphi; \dot{I}, a : \mathbf{ch}(\tilde{T}) \vdash_{c_1 \cup c_2 \cup c_3} a(\tilde{v}).P \triangleleft ((\uparrow^I(\Gamma \setminus \{a, \tilde{v}\}), a : \mathbf{ch}_I^\gamma(\tilde{T})), K')}$ <p>where <math>\mathbf{ch}_I^\gamma(\tilde{T}) = \text{fresh}(\mathbf{ch}(\tilde{T}), \varphi)</math>      <math>K' = \text{fresh}(\varphi)</math></p> <p><math>\mathcal{C}_2 = \varphi; \cdot \vdash (\cdot, a : \mathbf{ch}_I^\gamma(\tilde{T}), \tilde{v} : \tilde{T}) \sqsubseteq \Gamma</math></p> <p><math>\mathcal{C}_3 = \{\{\mathbf{in}\} \subseteq \gamma, \varphi; \cdot \models K + I \leq K'\}</math></p>
(I-OCH)	$\frac{\varphi; \dot{I} \vdash^{(1 \leq j \leq n)} e_j : (\Gamma_j, S_j)}{\varphi; \dot{I}, a : \mathbf{ch}(\dot{T}_1, \dots, \dot{T}_n) \vdash_{c_1 \cup c_2 \cup c_3} \overline{a}(e_1, \dots, e_n) \triangleleft (\Delta, K)}$ <p>where <math>\mathbf{ch}_I^\gamma(T_1, \dots, T_n) = \text{fresh}(\mathbf{ch}(\dot{T}_1, \dots, \dot{T}_n), \varphi)</math>      <math>K = \text{fresh}(\varphi)</math></p> <p><math>\Delta = \uparrow^I(\Gamma_1, \dots, \Gamma_n), a : \mathbf{ch}_I^\gamma(T_1, \dots, T_n)</math></p> <p><math>\mathcal{C}_1 = \{\varphi; \cdot \vdash S_j \sqsubseteq T_j \mid 1 \leq j \leq n\}</math></p> <p><math>\mathcal{C}_2 = \Gamma_1 \sim \dots \sim \Gamma_n</math></p> <p><math>\mathcal{C}_3 = \{\varphi; \cdot \models I \leq K, \{\mathbf{out}\} \subseteq \gamma\}</math></p>

**Table 9.** Process time inference rules (2).

$$\begin{aligned}
\mathcal{C} = & \{ \text{ch}_{I_2}^{\gamma_2}() \sim \text{ch}_{I_1}^{\gamma_1}(\{J_1/i\}, \cdot; \cdot \vdash \text{ch}_{I_3}^{\gamma_3}()) \sqsubseteq \text{ch}_{I_2}^{\gamma_2}(), \\
& \text{Nat}[0, 0] \sim \text{Nat}[J_2, J_3], \cdot; \cdot \vdash \text{Nat}[J_2 + 1, J_3 + 1] \sqsubseteq \text{Nat}[J_4, J_5], \\
& \text{Nat}[J_4, J_5] \sim \text{Nat}[J_6, J_7], \cdot; \cdot \vdash \text{Nat}[J_6 + 1, J_7 + 1] \sqsubseteq \text{Nat}[J_8, J_9], \\
& \text{Nat}[J_{10}, J_{11}] \sim \text{Nat}[J_{12}, J_{13}]\{J_1/i\}, \cdot; \cdot \vdash \text{Nat}[J_8, J_9] \sqsubseteq \text{Nat}[J_{10}, J_{11}], \\
& \{\text{out}\} \subseteq \gamma_4, \cdot; \cdot \models I_4 + K_1\{J_1/i\} \leq K_2, \\
& \{\text{in}\} \subseteq \gamma_5, \cdot; \cdot \models K_3 + I_5 \leq K_4, \text{ch}_{I_3}^{\gamma_3}() \sim \text{ch}_{I_5}^{\gamma_5}(), K_2 \sim K_4 \}
\end{aligned}$$

## 6 Reduction of constraint satisfaction problems

In this section, we show how a set of constraints can be reduced to a set of coefficient equivalence constraints and index inequality constraints. We first show how a set of type constraints can be reduced to a set of use-constraints. We then consider reduction of use-constraints to index constraints.

### 6.1 Reduction from type constraints to use-constraints

We consider rules for reduction of time invariance constraints, conditional subtyping constraints and equality constraints separately. The rules for reduction of time invariance constraints are shown in Table 10. As natural types are always time invariant, we can safely discard invariance constraints on such types. Similarly, normal channel types cannot be time invariant, and so if the constraint satisfaction problem includes such a constraint, we know there exists no solution. We constrain server types to only have output capability. Note that for a server to be time invariant, its time tag must also be zero. However, as we use delaying rather than advancement of time in the inference rules, we cannot enforce such a constraint here. We handle this separately in the inference rule for replicated inputs.

$$\begin{aligned}
& (I, \mathcal{C} \cup \{\varphi; \Phi \vdash \text{inv}(\text{Nat}[I, J])\}, K) \rightsquigarrow (I, \mathcal{C}, K) \\
& (I, \mathcal{C} \cup \{\varphi; \Phi \vdash \text{inv}(\text{ch}_I^\sigma(\tilde{T}))\}, K) \rightsquigarrow (I, \{\text{false}\}, K) \\
& (I, \mathcal{C} \cup \{\varphi; \Phi \vdash \text{inv}(\forall i. \tilde{\text{serv}}_K^\sigma(\tilde{T}))\}, K') \rightsquigarrow (I, \mathcal{C} \cup \{\sigma \subseteq \{\text{out}\}\}, K')
\end{aligned}$$

**Table 10.** Rules for reduction of time invariance constraints.

We next consider reduction of conditional subtyping constraints, as shown in Table 11. These constraints are satisfied when either the antecedent is false or the consequent is satisfied. The idea behind using Horn-clauses is that reduction

of subtyping constraints depends on use-capabilities, yet may introduce new use-constraints. Thus, their use enables reduction of subtyping constraints prior to solving use-constraints, thereby allowing us to solve them separately. For subtyping constraints on natural types, we simply introduce conditional constraints on the lower and upper size bounds. For channels and servers, we introduce conditional constraints for each type of variance, i.e. covariance, contravariance and invariance, and extend the antecedents of the implications accordingly.

$$\begin{aligned}
& (\Gamma, \mathcal{C} \cup \{\tilde{c}_\gamma \implies (\varphi; \Phi \vdash \text{Nat}[I, J] \sqsubseteq \text{Nat}[I', J'])\}, K) \rightsquigarrow (\Gamma, \mathcal{C} \cup \left\{ \begin{array}{l} \tilde{c}_\gamma \implies (\varphi; \Phi \models I' \leq I), \\ \tilde{c}_\gamma \implies (\varphi; \Phi \models J \leq J') \end{array} \right\}, K) \\
& (\Gamma, \mathcal{C} \cup \{\tilde{c}_\gamma \implies (\varphi; \Phi \vdash \text{ch}_I^\sigma(\tilde{T}) \sqsubseteq \text{ch}_{J'}^{\sigma'}(\tilde{S}))\}, K) \rightsquigarrow (\Gamma, \mathcal{C} \cup \left\{ \begin{array}{l} \tilde{c}_\gamma \implies \sigma' \subseteq \sigma, I \sim J, \\ (\tilde{c}_\gamma, \{\text{in}\} \subseteq \sigma') \implies (\varphi; \Phi \vdash \tilde{T} \sqsubseteq \tilde{S}), \\ (\tilde{c}_\gamma, \{\text{out}\} \subseteq \sigma') \implies (\varphi; \Phi \vdash \tilde{S} \sqsubseteq \tilde{T}) \end{array} \right\}, K) \\
& (\Gamma, \mathcal{C} \cup \{\tilde{c}_\gamma \implies (\varphi; \Phi \vdash \forall_{I\tilde{i}}.\text{serv}_K^\sigma(\tilde{T}) \sqsubseteq \forall_{J\tilde{i}}.\text{serv}_{K'}^{\sigma'}(\tilde{S}))\}, K'') \rightsquigarrow \\
& \quad (\Gamma, \mathcal{C} \cup \left\{ \begin{array}{l} \tilde{c}_\gamma \implies \sigma' \subseteq \sigma, I \sim J, \\ (\tilde{c}_\gamma, \{\text{in}\} \subseteq \sigma') \implies ((\varphi, \tilde{i}); \Phi \vdash \tilde{T} \sqsubseteq \tilde{S}), \\ (\tilde{c}_\gamma, \{\text{in}\} \subseteq \sigma') \implies ((\varphi, \tilde{i}); \Phi \models K' \leq K), \\ (\tilde{c}_\gamma, \{\text{out}\} \subseteq \sigma') \implies ((\varphi, \tilde{i}); \Phi \vdash \tilde{S} \sqsubseteq \tilde{T}), \\ (\tilde{c}_\gamma, \{\text{out}\} \subseteq \sigma') \implies ((\varphi, \tilde{i}); \Phi \models K \leq K') \end{array} \right\}, K'')
\end{aligned}$$

**Table 11.** Rules for reduction of conditional subsumption constraints.

In Table 12, we show rules for reducing type equivalence constraints. Such constraints are satisfied only if the two operands are equivalent, and so we simply replace them with new equivalence constraints for their constituents. Example 7 shows how a set of type constraints can be reduced.

$$\begin{aligned}
& (\Gamma, \mathcal{C} \cup \{\text{Nat}[I, J] \sim \text{Nat}[I', J']\}, K) \rightsquigarrow (\Gamma, \mathcal{C} \cup \{I \sim I', J \sim J'\}, K) \\
& (\Gamma, \mathcal{C} \cup \{\text{ch}_I^\sigma(\tilde{T}) \sim \text{ch}_{J'}^{\sigma'}(\tilde{S})\}, K) \rightsquigarrow (\Gamma, \mathcal{C} \cup \{\sigma \subseteq \sigma', \sigma' \subseteq \sigma, I \sim J, \tilde{T} \sim \tilde{S}\}, K) \\
& (\Gamma, \mathcal{C} \cup \{\forall_{I\tilde{i}}.\text{serv}_K^\sigma(\tilde{T}) \sim \forall_{J\tilde{i}}.\text{serv}_{K'}^{\sigma'}(\tilde{S})\}, K'') \rightsquigarrow (\Gamma, \mathcal{C} \cup \{\sigma \subseteq \sigma', \sigma' \subseteq \sigma, I \sim J, K \sim K', \tilde{T} \sim \tilde{S}\}, K'')
\end{aligned}$$

**Table 12.** Rules for reduction of type equivalence constraints.

*Example 7.* We now return to the example from the previous section, and show how the inferred type constraints can be reduced to use-constraints. As there are no time invariance constraints in the example, we first apply the reduction rules of conditional subsumption constraints iteratively, until no changes can be made to the set of constraint. The result is

$$\begin{aligned}
& \{\text{ch}_{I_2}^{\gamma_2}() \sim \text{ch}_{I_1}^{\gamma_1}()\{J_1/i\}, \gamma_2 \subseteq \gamma_3, I_3 \sim I_2, \\
& \text{Nat}[0, 0] \sim \text{Nat}[J_2, J_3], \cdot \vdash J_4 \leq J_2 + 1, \cdot \vdash J_3 + 1 \leq J_5, \\
& \text{Nat}[J_4, J_5] \sim \text{Nat}[J_6, J_7], \cdot \vdash J_8 \leq J_6 + 1, \cdot \vdash J_7 + 1 \leq J_9, \\
& \text{Nat}[J_{10}, J_{11}] \sim \text{Nat}[J_{12}, J_{13}]\{J_1/i\}, \cdot \vdash J_{10} \leq J_8, \cdot \vdash J_9 \leq J_{11}, \\
& \{\text{out}\} \subseteq \gamma_4, \cdot \vdash I_4 + K_1\{J_1/i\} \leq K_2, \\
& \{\text{in}\} \subseteq \gamma_5, \cdot \vdash K_3 + I_5 \leq K_4, \text{ch}_{I_3}^{\gamma_3}() \sim \text{ch}_{I_5}^{\gamma_5}(), K_2 \sim K_4\}
\end{aligned}$$

We then apply the rules to reduce type equivalence constraints. The result is

$$\begin{aligned}
& \gamma_2 \subseteq \gamma_1, \gamma_1 \subseteq \gamma_2, I_2 \sim I_1\{J_1/i\}, \gamma_2 \subseteq \gamma_3, I_3 \sim I_2, \\
& 0 \sim J_2, 0 \sim J_3, \cdot \vdash J_4 \leq J_2 + 1, \cdot \vdash J_3 + 1 \leq J_5, \\
& J_4 \sim J_6, J_5 \sim J_7, \cdot \vdash J_8 \leq J_6 + 1, \cdot \vdash J_7 + 1 \leq J_9, \\
& J_{10} \sim J_{12}\{J_1/i\}, J_{11} \sim J_{13}\{J_1/i\}, \cdot \vdash J_{10} \leq J_8, \cdot \vdash J_9 \leq J_{11}, \\
& \{\text{out}\} \subseteq \gamma_4, \cdot \vdash I_4 + K_1\{J_1/i\} \leq K_2, \\
& \{\text{in}\} \subseteq \gamma_5, \cdot \vdash K_3 + I_5 \leq K_4, \\
& \gamma_3 \subseteq \gamma_5, \gamma_5 \subseteq \gamma_3, I_3 \sim I_5, K_2 \sim K_4\}
\end{aligned}$$

## 6.2 Reduction from use-constraints to index constraints

We are now ready to consider reduction from a set of use-constraints to a set of index constraints. We take inspiration from previous work on type reconstruction of linear input/output types by Igarashi and Kobayashi [?], for which constraints on use-capabilities are solved using an algorithm that iteratively assigns capabilities to use-variables until all use-constraints are satisfied or the process can be deemed ill-typed. A necessary condition for termination is that all operations on use-capabilities must be monotonic. For our purposes, it suffices to use set-union, and so the condition is trivially satisfied.

We define Algorithm 1 that takes a set of use-constraints as input, and returns a set of index constraints. The algorithm iteratively computes a valuation of sets of use-capabilities for free use-variables in the set of constraints, by extending the use-capabilities assigned to some variable with the least set necessary to satisfy a corresponding use-constraint. When it is no longer possible to satisfy more use-constraints by extension of use-capability sets, we check for the existence of an unsatisfied constraint on use-capabilities. If one exists, we return the set **{false}** to denote that the process is ill-typed. Otherwise, we discard all constraints on

use-capabilities. Conditional index inequality constraints are discarded if their antecedents are not satisfied, and are otherwise replaced by their consequents. When using the algorithm to solve a set of constraints on use-capabilities, we apply the computed use-variable valuation on our type context. We write  $c_{\gamma_f}$  for the condition that  $c_\gamma$  is satisfied when its variables are instantiated according to valuation  $f$ . We extend this to sequences of constraints on use-capabilities, writing  $\tilde{c}_{\gamma_f}$  for the condition  $c_{\gamma_f^1} \wedge \dots \wedge c_{\gamma_f^n}$  where  $\tilde{c}_\gamma = c_{\gamma^1} \wedge \dots \wedge c_{\gamma^n}$ .

```

Data:  $\mathcal{C}_{IO}$  ; // A set of use-constraints
Result:  $(\mathcal{C}_I, f)$  ; // An index constraint set and a use-variable
                    valuation
 $f, f' \leftarrow$  Use-variable valuations with  $f(\gamma) = f'(\gamma) = \emptyset$  for  $\gamma \in \text{ftv}(\mathcal{C}_{IO})$ ;
repeat
   $f \leftarrow f'$ ;
  if  $\exists(\tilde{c}_\gamma \implies \gamma \subseteq \gamma') \in \mathcal{C}_{IO}.\tilde{c}_{\gamma_f}$  and  $f(\gamma) \not\subseteq f(\gamma')$  then
    |  $f' \leftarrow f[\gamma' \mapsto f(\gamma) \cup f(\gamma')]$ 
  else if  $\exists(\tilde{c}_\gamma \implies \sigma \subseteq \gamma') \in \mathcal{C}_{IO}.\tilde{c}_{\gamma_f}$  and  $\sigma \not\subseteq f(\gamma')$  then
    |  $f' \mapsto f[\gamma' \mapsto \sigma \cup f(\gamma')]$ 
  end
until  $f'(\gamma) = f(\gamma)$  for  $\gamma \in \text{ftv}(\mathcal{C}_{IO})$ ;
if  $\exists(\tilde{c}_\gamma \implies c'_\gamma) \in \mathcal{C}_{IO}.\tilde{c}_\gamma \wedge \neg c'_{\gamma_f}$  then
  |  $\mathcal{C}_I \leftarrow \{\text{false}\}$ 
else
  |  $\mathcal{C}_I \leftarrow \emptyset$ ;
  foreach  $(\tilde{c}_\gamma \implies (\varphi; \Phi \models I \leq J)) \in \mathcal{C}_{IO}.\tilde{c}_{\gamma_f}$  do  $\mathcal{C}_I \leftarrow \mathcal{C}_I \cup \{\varphi; \Phi \models I \leq J\}$ ;
  foreach  $c_I \in \mathcal{C}_{IO}$  do  $\mathcal{C}_I \leftarrow \mathcal{C}_I \cup \{c_I\}$ ;
end

```

**Algorithm 1:** Iterative use-constraint solver

### 6.3 Solving constraint satisfaction problems

In this section, we show how a solution to a set of index constraints can be over-approximated by a set of potentially polynomial inequalities. We first consider index equivalence constraints. Due to the restrictions on how indices may be formed, such constraints can be directly reduced to coefficient-wise equivalence constraints, by the following reduction rule

$$(\Gamma, \mathcal{C}_I \cup \{\mathbf{c}_0 + \mathbf{c}_1 i_1 + \dots + \mathbf{c}_n i_n \sim \mathbf{c}'_0 + \mathbf{c}'_1 i_1 + \dots + \mathbf{c}'_n i_n\}, K) \rightsquigarrow (\Gamma, \mathcal{C}_I \cup \{\mathbf{c}_j \sim \mathbf{c}'_j \mid 1 \leq j \leq n\}, K)$$

Thus, equivalence constraints can be given directly to existing constraint satisfaction solvers. An index inequality constraint  $(\cdot, \tilde{i}); (\cdot, C_1, \dots, C_n) \models I \leq J$  is more difficult to solve, as it can be seen as a Horn-clause with  $C_1, \dots, C_n$  as antecedents and  $I \leq J$  as a single consequent, existentially quantified over coefficient variables and universally quantified over  $\tilde{i}$ , i.e. it takes the form

$$\exists \tilde{\alpha}. \forall \tilde{i}. C_1 \wedge \dots \wedge C_n \implies I \leq J$$

Such constraints are intractable in general, and so we must over-approximate them. We first present a useful result in Lemma 17. That is, it is an over-approximation to discard index variable constraints.

**Lemma 17.** *If  $\varphi; \Phi \models C$  and  $\Phi \subseteq \Phi'$  then  $\varphi; \Phi' \models C$ .*

*Proof.* Follows directly from  $\mathcal{M}_\varphi(\Phi') \subseteq \mathcal{M}_\varphi(\Phi)$ .

However, discarding antecedents comes at the price of expressiveness and precision. Consider for instance the judgement

$$(\cdot, i); (\cdot, i \geq 1) \models (i - 1) + 1 \leq i$$

It holds precisely because  $i \geq 1$ , ensuring  $-1$  and  $1$  cancel out. If we discard the antecedent, the consequent must hold under valuation  $\rho(i) = 0$ , which is not the case due to the monus operator, i.e.  $\llbracket (i - 1) + 1 \rrbracket_\rho = 1$ . Based on our experience, we believe only bounds on constant time servers can be inferred without antecedents of the form  $i \geq 1$  (using our inference algorithm), therefore we show how they can be simulated by substitution in some cases.

For constraints of the form  $(\varphi, i); (\Phi, i \geq I) \models C$ , we substitute  $i + I$  for  $i$ , i.e.  $(\varphi, i); \Phi\{i + I/i\} \models C\{i + I/i\}$ . For the above example, we then have the trivially satisfied judgement

$$(\cdot, i); \cdot \models (i + 1 - 1) + 1 \leq i + 1$$

This applies to index variable equality constraints as well. They can be simulated by substitution when one side is a single index variable. For instance  $\varphi; (\Phi, i = I) \models C$  if and only if  $\varphi; \Phi \models C\{I/i\}$ . Therefore, we can safely assume that  $\Phi$  is empty, at the cost of expressiveness and precision. We can now try to solve our constraints using an existing constraint satisfaction solver, which may be possible in some cases. Another option is to perform naive quantifier elimination. A reasonable but restrictive over-approximation is to reduce inequality constraints to coefficient-wise inequality constraints. Using this approach, we end up with a set of possibly polynomial equality and inequality constraints that is hopefully solvable in practice.

## 6.4 Simplification of constraint satisfaction problems

The number of constraints introduced by our inference algorithm grows rapidly with the size of processes, thereby making it difficult to find solutions, greatly exacerbated by polynomial constraints from substitutions. This is exemplified by terms of natural numbers, as the successor constructor introduces a constraint. Thus, whereas we introduce 4 constraints for the expression  $s(s(s(0)))$ , we require 6 for  $s(s(s(s(s(0)))))$ , and so the number of constraints scales linearly with the size of inputs to servers. However, it suffices to introduce 2 constraints for either expression, by treating a sequence of successors as a single joint successor using

a rule of the form

$$(I\text{-succ}^*) \frac{\varphi; \dot{I} \vdash_{\mathcal{C}} e : (I, \mathbf{Nat}[I, J])}{\varphi; \dot{I} \vdash_{\mathcal{C} \cup \{\varphi; \vdash \mathbf{Nat}[I+n, J+n] \sqsubseteq \mathbf{Nat}[I', J']\}} \tilde{s}(e) : (I, \mathbf{Nat}[I', J'])}$$

where  $I', J' = \text{fresh}(\varphi)$

$$\tilde{s}(e) = \overbrace{s(\cdots s(e) \cdots)}^n$$

Another concern is the flexibility of server types in the work of Baillot and Ghyselen [3]. That is, the quantified index variables introduced in a server type need not be associated with the size of a natural number, which forces us to treat all unknown sizes as templates for indices. However, many interesting primitive recursive functions can be encoded as servers using types of the form

$$\forall_I i_1, \dots, i_{2n}. \text{serv}_K^\sigma(\mathbf{Nat}[i_1, i_2], \dots, \mathbf{Nat}[i_{2n-1}, i_{2n}], \text{ch}_{I'}^{\sigma'}(\tilde{T}))$$

That is, each size bound on *outer-most* natural message types of the server is a unique index variable, and the time-tags and sizes of all other message types are indices quantified over these index variables. If we enforce server types to be of this form, we can greatly decrease the number of coefficient variables needed. Moreover, restricting index variables to be quantified over size bounds on naturals enables us to directly derive the substitution needed to instantiate a server. Consider for instance the server  $f$  bound to type  $\forall_{I_1} i, j. \text{serv}_K^\sigma(\mathbf{Nat}[i, j], \text{ch}_{I_2}^{\sigma'}(\mathbf{Nat}[J_1, J_2]))$ . Then, for some expression  $e$  with type  $\mathbf{Nat}[J_3, J_4]$ , the following constraints are sufficient for the judgement  $\varphi; \Phi; (I, f : \forall_{I_1} i, j. \text{serv}_K^\sigma(\mathbf{Nat}[i, j], \text{ch}_{I_2}^{\sigma'}(\mathbf{Nat}[J_1, J_2]))) \vdash \bar{f}(e) \triangleleft K'$

$$\begin{aligned} \varphi; \Phi \vdash \mathbf{Nat}[J_3, J_4] &\sqsubseteq \mathbf{Nat}[L_1, L_2] \\ \varphi; \Phi \vdash \text{ch}_{I_2}^{\sigma'}(\mathbf{Nat}[J_1, J_2]) &\sqsubseteq \text{ch}_{I_3}^{\sigma''}(\mathbf{Nat}[J_5, J_6])\{L_1/i, L_2/j\} \\ \varphi; \Phi \models I_1 + K\{L_1/i, L_2/j\} &\leq K' \end{aligned}$$

where  $L_1, L_2, I_3, J_5, J_6$  and  $\sigma''$  are *fresh*. Another benefit of restricting server types to this form is that we can more often simulate antecedents in index inequality constraints. That is, we can utilize that size bounds on message natural types have exactly one positive coefficient on an index variable, i.e.  $\mathbf{Nat}[i, j]$ , and as we often pattern match on naturals received on replicated inputs, we would expect our antecedents to often be of the form  $i \geq 1$ . However, this is complicated by subsumption constraints. That is, as we may increase the upper bound on a natural type, we introduce a fresh copy of the natural type, i.e.  $\mathbf{Nat}[I, J]$  with the constraint  $\varphi; \cdot \vdash \mathbf{Nat}[i, j] \sqsubseteq \mathbf{Nat}[I, J]$ . Thus, our antecedents will typically be of the form  $J \geq 1$ , which is more difficult to simulate as an index variable is not isolated. However, due to our over-approximation of the constraint  $\varphi; \cdot \models j \leq J$ , i.e. pair-wise coefficient inequality constraints, we can deduce that  $J = L + cj$  with  $c$  being positive for some index  $L$  with index variables in  $\varphi \setminus \{j\}$ . Thus, we

can safely isolate  $j$  and simulate the antecedent by substituting  $(1 - L)/c + j$  for  $j$ . Note that this may be an under-approximation of the antecedent (due to constraint over-approximation) and thereby an over-approximation of the constraint, and so for maximum expressiveness, we may try all combinations of discarding and simulating antecedents.

## 6.5 Haskell implementation

We have implemented our type inference algorithm in Haskell using the Z3 SMT-solver [20]. We use the over-approximations from the previous section, and we further omit lower-bounds on natural sizes, i.e. they are always 0. These together simplify the generated constraint satisfaction problems substantially, for some of the processes we have tested. Moreover, the simulation of antecedents described in the previous section enables us to infer bounds on some linear time servers, which we have not been able to do when antecedents are simply discarded. Our implementation is available at GitHub<sup>1</sup>.

We optimize solutions to the reduced and over-approximated constraint satisfaction problems, by minimizing the coefficients of the complexity bound inferred for a process by our inference algorithm. This works quite well in practice, however, for servers that are not invoked, we might get imprecise bounds on server complexities. This can be alleviated by either using a different objective function (minimizing the server complexity bound) or by invoking the server with arguments with sizes based on new index variables.

We are now ready to show some examples of processes that we can and cannot infer reasonable bounds on, respectively. We first consider two simple *recursive* servers that differ only by tick-placement. The processes are shown below, where  $m$  is a free variable with the type  $\text{Nat}[0, j]$ , and so  $P_{\text{np\textsubscript{ar}}}$  has a span of 1 and work of  $j$  whereas  $P'_{\text{np\textsubscript{ar}}}$  has a span and work of  $j$ . Our Haskell implementation can infer these bounds in a few seconds when run on a normal desktop workstation. In particular, we infer the two types  $\forall_0 i_0. \text{serv}_{1+0i_0}^{\{\text{in}, \text{out}\}}(\text{Nat}[0, 0 + 1i_0], \text{ch}_{1+0i_0}^{\{\text{out}\}}())$  and  $\forall_0 i_0. \text{serv}_{0+1i_0}^{\{\text{in}, \text{out}\}}(\text{Nat}[0, 0 + 1i_0], \text{ch}_{0+1i_0}^{\{\text{out}\}}())$  for server  $\text{np\textsubscript{ar}}$  in the two processes, respectively. The inferred and reduced constraint satisfaction problems are available in Appendix ??.

---

<sup>1</sup> <https://github.com/MikkellLauridsen/bg-inference>.



$$\begin{array}{c}
P_{\text{np\textit{ar}}} \stackrel{\text{def}}{=} \\
(\nu \text{np\textit{ar}})( \\
\quad !\text{np\textit{ar}}(n, r).\text{match } n \{ \\
\quad \quad 0 \mapsto \bar{r}\langle \rangle \\
\quad \quad s(x) \mapsto (\nu r')(\nu r'') ( \\
\quad \quad \quad \text{tick}.\bar{r'}\langle \rangle \mid \overline{\text{np\textit{ar}}}(x, r'') \mid r'().r''().\bar{r}\langle \rangle) \} \\
\quad \mid \\
(\nu r)(\overline{\text{np\textit{ar}}}(m, r) \mid r().\mathbf{0})
\end{array}
\qquad
\begin{array}{c}
P'_{\text{np\textit{ar}}} \stackrel{\text{def}}{=} \\
(\nu \text{np\textit{ar}})( \\
\quad !\text{np\textit{ar}}(n, r).\text{match } n \{ \\
\quad \quad 0 \mapsto \bar{r}\langle \rangle \\
\quad \quad s(x) \mapsto (\nu r')(\nu r'') \text{tick}.( \\
\quad \quad \quad \bar{r'}\langle \rangle \mid \overline{\text{np\textit{ar}}}(x, r'') \mid r'().r''().\bar{r}\langle \rangle) \} \\
\quad \mid \\
(\nu r)(\overline{\text{np\textit{ar}}}(m, r) \mid r().\mathbf{0})
\end{array}$$

As for processes that we cannot infer reasonable bounds on, we distinguish between those where imprecise bounds are inferred, those where the constraint satisfaction problems are not solved in reasonable time and those where our over-approximations lead to unsatisfiable constraint satisfaction problems. We have not been able to find an example of the former, but even for quite simple processes, our inferred constraint satisfaction problems are sometimes difficult to solve. Consider for instance the encoding of the addition operator of naturals as a server seen below. Although the process contains zero ticks, we were unable to infer a bound on its span within 3 hours.

$$P_{\text{add}} \stackrel{\text{def}}{=} !\text{add}(x, y, r).\text{match } x \{ 0 \mapsto \bar{r}\langle y \rangle; s(z) \mapsto \overline{\text{add}}(z, s(y), r) \}$$

An example of a process for which the corresponding over-approximated constraint satisfaction problem is unsatisfiable is the Fibonacci number encoding from Chapter ?? . Although its span is linear, the size upper-bound on the *returned* natural is not linear in the size of the *parameter* natural number, and so it cannot be expressed using our restricted server types.

## 7 Conclusion

In this paper, we have explored the challenges of implementing both a type checker and type inference for the type system for parallel complexity of  $\pi$ -calculus processes by Baillot and Ghyselen [3]. In this chapter, we first present and discuss our results, after which we discuss limitations of our implementations. Finally, we consider future work, and discuss how some of the limitations may be relaxed.

### 7.1 Results

Type checking and type inference are limited by our ability to respectively verify or satisfy constraint judgements. As such judgements are universally quantified over index variables, this becomes non-trivial. We can reduce verification of constraint judgements to linear programming, where a constraint judgement holds if there is no solution to the corresponding linear program. We have also shown

that certain polynomial constraint judgements can be reduced to linear constraint judgements, enabling type checking of some processes with polynomial time behavior. We have introduced algorithmic type rules for type checking, and proved their soundness with respect to parallel complexity.

Our type inference algorithm performs multiple passes over a program to first infer simple types, which are then used to infer constraints on the variance of input/output types, sizes of naturals, bounds on channel synchronization and complexity bounds. We can reduce such constraints to a set of constraint judgements with existentially quantified variables representing coefficients that when solved provide a bound on the span. These constraints are significantly more difficult to solve than those that emerge for type checking, and so we over-approximate them using naive quantifier elimination. We provide a Haskell implementation based on the Z3 SMT solver.

Overall, we find that we can type check many constant and linear time processes and some polynomial time processes. Similarly, we can infer precise bounds on many constant time and some linear time processes in reasonable time. However, both type checking and type inference are limited not only by the expressiveness of complexity bounds, but also by size bounds on naturals, as some encoded algorithms may return naturals with size bounds that exceed their complexity bounds. As our type inference algorithm infers polynomial constraints, we are also limited by the ability to find solutions, which we find to be difficult for many simple processes.

## 7.2 Discussion

During type checking, we limit ourselves to linear indices such that we can reduce constraint judgements to a linear program that can be solved by an algorithm such as the simplex algorithm. We have primarily made this choice for the sake of simplicity, however, tools such as *Gröbner bases* exist that can be used to help solving systems of polynomial equations. Even more generally, existing provers such as the Z3 prover, which utilizes, among other things, Gröbner bases, could be used to solve both linear and some super-linear systems.

As for type inference, we are able to infer precise bounds on some constant and linear time servers. We notice that inferred constraint satisfaction problems for servers quickly grow in difficulty based on the number of index variables. Even simple processes with no ticks take significantly longer time to solve, when an extra index variable is introduced. We ascribe this to the polynomial constraints introduced upon instantiating servers, and so it may make sense to consider further over-approximations of substitutions.

An interesting observation is that our restriction to linear indices makes us unable to infer bounds on some linear time processes. One such example is that of the Fibonacci number encoding, where we are unable to express size bounds

on the *returned* Fibonacci number, which grows according to the Fibonacci sequence, yet we can express the complexity of the server. Although the complexity is not directly affected by the size of this natural, we are thus unable to infer a sized type for the server. As the complexity of a process does not necessarily depend on all size bounds, it may be possible to let some of them be unknown, and thereby relax the restriction.

We also notice the importance of antecedents in constraint judgements. When all antecedents are discarded, we are unable to infer bounds on any linear time server. Moreover, we observe that coefficient variables do not take negative valuations unless we simulate antecedents. We believe this is due to our over-approximations of index inequality constraints. That is, we implicitly infer the constraint  $\varphi; \Phi \models 0 \leq I$  for all indices  $I$ , which we over-approximate using coefficient-wise inequality constraints, which means all coefficients must be non-negative. This seems to translate to all coefficient variables being non-negative as well. However, when antecedents are simulated, we may substitute a positive term into an index, providing more flexibility to coefficient variable valuations. We believe this is why our simulation of inequality antecedents greatly increases expressiveness of our type inference algorithm.

### 7.3 Future work

We have introduced algorithmic type rules for the type system by Baillot and Ghyselen [3], enabling us to implement a type checker that given some type environment and process, can verify if process is well-typed under the environment, and thereby provide us a bound on the parallel complexity. As such, a natural next step is to implement the type checker in for instance Haskell.

We have primarily limited ourselves to linear indices, and thereby linear complexity bounds, as the corresponding constraint judgements are then simpler to verify or satisfy. It may be worthwhile to explore how this restriction may be relaxed. Hofmann and Hoffmann [10] and Hoffmann et al. [9], show how linear constraints can be derived from polynomial bounds, by representing polynomials as sums of binomial coefficients. However, this is in the setting of first-order functional programs, and so it may be interesting to see if this method can be applied to message-passing processes.

Our type inference algorithm for parallel complexity of  $\pi$ -calculus processes provides correct bounds on the parallel complexity of the processes we have tried (when a bound can be inferred in reasonable time). However, we have yet to formally prove its correctness. In particular, we are interested in proving that our algorithm always infers principal typings. We expect this property to be quite straightforward to prove, as our inference rules are based on the type rules. One exception to this is our treatment of time invariance.

In this thesis, we have limited ourselves to the type system for parallel complexity by Baillot and Ghyselen [3]. However, the usage-based generalization of the type system provided by Baillot et al. [4] increases the expressiveness and precision. Usages allow for more precise description of the behavior of channels, for instance making processes with some forms of indeterministic communication typable. Usages are well-suited for inference, and as this type system shares many similarities with the type system by Baillot and Ghyselen, it may be interesting to see if our type inference algorithm can be extended to usages. Constraint-based inference of usage types have been studied previously, and judging from this work, we expect usage reliability to be the main challenge due to universal quantification over index variables [17,?].

## References

1. Martin Avanzini and Ugo Dal Lago. Automating sized-type inference for complexity analysis. *Proc. ACM Program. Lang.*, 1(ICFP), August 2017.
2. Raymond G. Ayoub. Paolo ruffini’s contributions to the quintic. *Archive for History of Exact Sciences*, 23(3):253–277, 1980.
3. 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.
4. 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.
5. 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.
6. Ankush Das, Jan Hoffmann, and Frank Pfenning. Parallel complexity analysis with temporal session types. *Proc. ACM Program. Lang.*, 2(ICFP), July 2018.
7. Martin Davis. Hilbert’s tenth problem is unsolvable. *The American Mathematical Monthly*, 80(3):233–269, 1973.
8. David Hilbert. Mathematical problems. *Bull. Amer. Math. Soc.*, 8(10):437–479, 1902.
9. Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. Multivariate amortized resource analysis. *ACM Trans. Program. Lang. Syst.*, 34(3), nov 2012.
10. 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.
11. 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.
12. Martin Hofmann and Steffen Jost. Static prediction of heap space usage for first-order functional programs. *SIGPLAN Not.*, 38(1):185–197, jan 2003.

13. 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.
14. Richard M. Karp. *Reducibility among Combinatorial Problems*, pages 85–103. Springer US, Boston, MA, 1972.
15. Naoki Kobayashi. A partially deadlock-free typed process calculus. *ACM Trans. Program. Lang. Syst.*, 20(2):436–482, mar 1998.
16. Naoki Kobayashi. Type-based information flow analysis for the  $\pi$ -calculus. *Acta Inf.*, 42(4):291–347, dec 2005.
17. 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.
18. Ugo Dal Lago and Marco Gaboardi. Linear Dependent Types and Relative Completeness. *Logical Methods in Computer Science*, 8(4), 2012.
19. Cédric Lhoussaine. Type inference for a distributed  $\pi$ -calculus. *Sci. Comput. Program.*, 50(1–3):225–251, mar 2004.
20. Microsoft Research. Z3. <https://github.com/Z3Prover/z3>. Accessed: June 11th 2022.
21. Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.
22. Robin Milner. The polyadic  $\pi$ -calculus: a tutorial. In Friedrich L. Bauer, Wilfried Brauer, and Helmut Schwichtenberg, editors, *Logic and Algebra of Specification*, pages 203–246, Berlin, Heidelberg, 1993. Springer Berlin Heidelberg.
23. Benjamin Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science*, 6(5):409–453, 1996.
24. 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).