


Numéro National de Thèse: 2021LYSEN036

THÈSE DE DOCTORAT DE L'UNIVERSITÉ DE LYON
opéréé par
l'École Normale Supérieure de Lyon

École Doctorale N° 512
École Doctorale en Informatique et Mathématiques de Lyon

Discipline : Informatique

Soutenue publiquement le 20/09/2021, par:

Alexis GHYSELEN

Sized Types Methods and their Applications to Complexity Analysis in Pi-Calculus

Les Types à Tailles et leurs Applications pour l'Analyse de Complexité dans le Pi-Calcul

Devant le jury composé de:

MAZZA Damiano, Directeur de Recherche, Université Paris Nord, CNRS	Rapporteur
YOSHIDA Nobuko, Professor, Imperial College London	Rapporteure
HOFFMANN Jan, Associate Professor, Carnegie Mellon University	Examinateur
MARION Jean-Yves, Professeur des Universités, LORIA	Examinateur
BAILLOT Patrick, Directeur de Recherche, ENS de Lyon, CNRS	Directeur de thèse

Merci à ma famille et mes amis qui m'ont toujours soutenu pendant ces trois années depuis le Havre. Merci à Patrick pour son accompagnement, ses conseils et sa bienveillance. Merci à l'équipe PLUME pour la bonne ambiance et les différents exposés passionnants qui m'ont souvent aidé pendant ses années. Merci à Paul et Julien pour les discussions intéressantes que l'on a pu avoir en début de thèse sur des sujets diverses. Merci à Daniel pour sa bonne humeur et son aide précieuse dans l'enseignement et mes recherches. Merci également à l'équipe CASH et le LIP en général pour de nombreuses discussions passionantes et séminaires enrichissants.

Thanks to Damiano Mazza and Nobuko Yoshida for their reading of the manuscript.

Thanks to Jan Hoffmann and Jean-Yves Marion for being available for my defense.

Abstract

La complexité est une notion importante en informatique, aussi bien pour l'étude des programmes que pour des questions théoriques. Pour un programme ou un algorithme, la complexité correspond au nombre de ressources nécessaires pour calculer une sortie sur une entrée donnée, ce qui informe alors de son efficacité. En pratique, deux ressources sont principalement étudiées: le *temps* et l'*espace*. L'analyse statique de ces ressources est couramment appelée *l'analyse de complexité*. Pour ce qui est des questions théoriques, le point central est l'étude des classes de complexité, dans lesquelles les problèmes sont regroupés selon la complexité du meilleur programme qui peut les résoudre. Dans cette thèse, nous étudions des méthodes basées sur les systèmes de *types à tailles* pour la complexité en temps, en particulier pour l'analyse des processus du π-calcul, un modèle basé sur les communications pour les calculs parallèles. L'idée centrale des types à tailles est de tracer la taille des valeurs d'un programme et d'utiliser cette information pour contrôler la récursion et en déduire des bornes de complexité en temps.

Dans la première partie de cette thèse, nous nous intéressons à une approche théorique de la complexité, dans le cadre de la complexité computationnelle implicite (ICC). L'objectif de l'ICC est de caractériser des classes de complexité en utilisant la logique ou les types, généralement sans donner de bornes explicites. Cela permet en particulier une compréhension plus profonde des ces classes. Plusieurs méthodes ont été développées pour la complexité en temps. Une approche principale vient de la logique linéaire, en utilisant des versions restreintes de la modalité "!", qui contrôle la duplication. On peut citer en première instance la logique linéaire light, qui caractérise les calculs en temps polynomial. Une autre approche, illustrée par les types non-size increasing ou encore les types à tailles évoqués plus haut, est de se concentrer sur la taille des valeurs. Ces deux approches ont des défauts. La première approche induit une faible expressivité intentionnelle: certains programmes naturels en temps polynomial ne sont pas typables. Pour la seconde approche, elle est essentiellement linéaire, donc elle ne permet pas en particulier une utilisation non-linéaire des arguments d'ordres supérieurs. Dans cette thèse, nous surmontons ces contraintes en combinant les deux approches dans un système de type commun. Le langage que nous utilisons est un λ-calcul avec des types de données et un itérateur: un variant du système T de Gödel. Nous élaborons un système de type pour ce langage qui autorise des arguments fonctionnels non-linéaires avec une expressivité intentionnelle vraisemblablement correcte. Notre approche se base sur la logique linéaire élémentaire (ELL) combinée avec un système de types à tailles linéaires. Nous discutons de l'expressivité de ce système de type, appelé sEAL, et nous prouvons qu'il donne une caractérisation des classes de complexité FPTIME et 2k-FEXPTIME, pour $k \geq 0$.

Dans la seconde partie de cette thèse, nous étudions l'analyse de complexité avec des types. Cette approche pour analyser la complexité des programmes est un sujet de recherche important, en particulier pour les langages fonctionnels dans lesquels la notion de composition est essentielle. Parmi toutes les approches possibles, nous nous intéressons aux types à tailles. Nous explorons comment utiliser ces types pour l'analyse de complexité parallèle dans le π-calcul. Deux notions de complexités en temps sont étudiées: le temps de calcul total sans aucune parallélisation (le *travail*), et le temps de calcul avec une parallélisation maximale (la *profondeur*). Nous définissons des sémantiques opérationnelles pour refléter ces deux notions. La seconde sémantique est particulièrement importante car elle donne des preuves plus simple que d'autres notions apparentées à la profondeur. Nous présentons deux systèmes de types similaires à partir duquel on peut extraire une borne de complexité sur un processus. Ces systèmes sont inspirés à la fois par les types à tailles et les types entrée/sortie du π-calcul, auxquels on ajoute des informations temporelles. Cependant, cette extension des types à tailles fonctionnels pour l'analyse de profondeur a une expressivité limitée, en particulier en présence de certains comportements concurrents comme les sémaphores. Dans le but d'avoir une analyse plus expressive, nous élaborons un système de type qui repose sur le concept des usages, utilisés originalement pour l'analyse des deadlocks.

Abstract

Complexity is an important notion in computer science, both for the study of programs and theoretical problems. For a program or an algorithm, complexity corresponds to the amount of resources it needs to compute its output on a given input, indicating information about the efficiency. In practice, two resources are mainly studied: *time* and *space*. The static study of this is called *complexity analysis*. As for problems, the focus is on the study of complexity classes, where problems are regrouped depending on the complexity of the best program that can solve this problem. In this thesis, we study methods based on *sized type* systems for time complexity, especially for the analysis of processes in the π-calculus, considered as a communication-based model for parallel computation. The main idea of sized types is to track the size of values in a program, and to use this information to control recursion and deduce time complexity bounds.

In the first part of this thesis, we focus on a theoretical approach of complexity, following the lines of implicit computational complexity (ICC). The goal of ICC is to characterize complexity classes by means of logics or types, generally without explicit bounds. This allows in particular for a deeper understanding of complexity classes. Several methods have been proposed to characterize time complexity classes. One approach comes from linear logic and restricted versions of its !-modality controlling duplication. The first instance of this is light linear logic for polynomial time computation. Another approach, illustrated by non-size increasing types or the sized types defined before, is to focus on the sizes of values. However, both approaches suffer from limitations. The first one has a limited intensional expressivity, that is to say some natural polynomial time programs are not typable. Concerning the second approach, it is essentially linear, more precisely it does not allow for a non-linear use of higher-order arguments. In this thesis, we incorporate both approaches into a common type system, to overcome their respective constraints. The source language we consider is a λ-calculus with data-types and iteration, a variant of Gödel's system T. We design a type system for this language allowing non-linear functional arguments, with a seemingly good intensional expressivity. Our approach relies on elementary linear logic (ELL), combined with a system of linear sized types. We discuss the expressivity of this new type system, called sEAL, and prove that it gives a characterization of the complexity classes FPTIME and 2k-FEXPTIME, for $k \geq 0$.

In the second part of this thesis, we study complexity analysis with types. Type systems as a technique to analyse programs have been extensively studied, especially for functional programming languages where composition is essential. Among all the approaches for this, we focus on sized types. We explore how to extend those types to the analysis of parallel complexity in the π-calculus. Two notions of time complexity are studied: the total computation time without parallelism (*work*) and the computation time under maximal parallelism (*span*). We define operational semantics to capture those two notions. The semantics for span is particularly important, as it allows for simpler proof methods than related notions. We present then two similar type systems from which one can extract a complexity bound on a process, inspired both by sized types and input/output types, with additional temporal information about communications. However, this extension of functional sized types for span analysis has limited expressivity, especially in presence of concurrent behaviours such as semaphores. Aiming for a more expressive analysis, we design a type system which builds on the concepts of usages, originally used for deadlock-freedom analysis.

Contents

Chapter 1

Introduction

1.1 Introduction en Français

Contexte

Une question simple d'apparence mais intéressante en informatique est la comparaison des programmes: donnés deux programmes qui calculent la même fonction, est-ce que l'un est plus efficace que l'autre ? Une façon simple de faire cette comparaison serait de choisir un langage de programmation, d'implémenter ces deux programmes, de les lancer tous les deux sur une batterie de tests et de mesurer *dynamiquement* la consommation de ressource de ces différentes exécutions. Toutefois, cette méthode dépend de certains détails bas-niveaux comme le matériel utilisé ou encore le choix de l'implémentation, et ce n'est pas faisable en pratique si les programmes sont trop demandant en ressources, ou encore s'ils ne terminent pas. Ainsi, pour pouvoir comparer des programmes de façons théoriques, on préférait utiliser une méthode *statique*, telle que *l'analyse de complexité*. Le but de l'analyse de complexité est, sur un programme donné, d'approximer les ressources nécessaires pour calculer la sortie pour une entrée quelconque. Habituellement, les ressources que l'on considère sont *l'espace* et *le temps*. La complexité en espace correspond à l'espace mémoire dont le programme a besoin pour calculer et la complexité en temps correspond au temps d'exécution du programme. Celle-ci se base usuellement sur une notion abstraite du temps (par exemple, une multiplication arithmétique = une unité de temps) afin de simplifier le raisonnement. Tout comme l'analyse de terminaison des programmes, l'analyse de complexité est indécidable, mais cela reste cependant un problème intéressant et difficile.

En pratique, pour un langage donné, afin de faire une analyse de complexité en temps on doit d'abord décider comment abstraire la notion de temps et différents choix sont possibles selon le langage. Par exemple, pour un langage fonctionnel abstrait comme une extension du λ-calcul, on pourrait prendre pour unité de temps une étape de réduction, ou encore le temps d'exécution sur une machine abstraite. Si un programme interagit avec un réseau, on pourrait être intéressé par le nombre d'accès réseau, car c'est une opération coûteuse en pratique. De même, selon les effets d'un langage, on pourrait s'intéresser à différentes notions de complexité. Par exemple, dans un langage non-déterministe, on pourrait choisir la complexité dans le pire cas, ou selon une stratégie de réduction spécifique. En présence de programmes randomisés, on pourrait s'intéresser à l'espérance de la complexité, ou encore la probabilité de satisfaire une borne de complexité spécifique. En présence de parallélisme, on pourrait considérer la complexité totale séquentielle, appelé *le travail*, ou encore d'autres notions de complexité parallèle...

Pour des programmes parallèles, une notion intéressante par exemple est la complexité avec un nombre p de processeurs, ce qui veut dire intuitivement que p calculs peuvent être

fait en simultanés en parallèle, ce qui peut correspondre à la complexité en pratique d'un programme lancé sur un système multicœur. D'un point de vue plus théorique, on pourrait considérer un nombre infini de processeurs, ce qui donne la complexité sous l'hypothèse de parallélisme maximal, ce qu'on appelle usuellement la *profondeur*. La profondeur n'est pas en général atteignable en pratique, mais elle a un intérêt théorique puisqu'un résultat assez connu [55] dit qu'à partir du travail w et de la profondeur s d'un programme dans une certaine classe, on peut en déduire une borne sur le temps d'exécution en pratique avec p processeurs, en $O(\max(w/p), s)$. Intuitivement, la profondeur nous dit à quel point il est efficace de lancer un programme en parallèle, et une analyse de profondeur pourrait être complétée avec une analyse du nombre de processeurs nécessaires pour atteindre celle-ci.

L'analyse de complexité des programme a été largement étudiée, et de nombreuses approches ont été explorées pour permettre cette analyse. Parmi toutes ces différentes approches, une qui nous intéresse particulièrement est l'utilisation des systèmes de types. De façon générale, un système de type est une façon d'abstraire un programme afin de garantir certaines propriétés sur celui-ci, de telle manière qu'un *programme typé ne peut pas aller mal*. Une analyse par système de type a des avantages intéressants comparées à d'autre approches, vu qu'elle induit une certaine *compositionnalité* et une certaine *modularité*. En effet, un système de type consiste usuellement en un ensemble de règles qui correspondent aux constructeurs du langage, et donc l'analyse d'un programme en entier se base sur l'analyse de tout les sous-programmes, analyse qui peut ensuite être réutilisée dans des contextes différents. Cette approche est particulièrement utile pour les langage avec une notion de composition intégrée, comme les langages fonctionnels pour lesquels l'analyse de complexité doit prendre en compte la composition et la réutilisation des fonctions. Une fois qu'un système de type est élaboré pour une propriété spécifique d'un langage, deux questions principales surviennent: la vérification de types et l'inférence de types. La vérification de types consiste en, donnés un programme et une dérivation de typage candidate, vérifier automatiquement que cette dérivation de type est correcte. Pour la propriété associée, cela correspond à vérifier un certificat. L'inférence de type consiste en, donné un programme, trouver automatiquement une dérivation de type pour ce programme. Pour la propriété associée, cela correspond à vérifier automatiquement si un programme satisfait cette propriété. L'inférence de types est donc plus difficile que la vérification de types, et il y a usuellement une contrepartie pour les systèmes de type: plus un système est expressif (plus il peut typer de programmes), plus il est difficile de vérifier un type ou de l'inférer. Dans le cas particulier de l'analyse de complexité, le but est, à partir d'une dérivation de type d'un programme, d'extraire une borne sur la complexité. Ainsi, la vérification de types nous donne un certificat de complexité et l'inférence de type nous donne une analyse automatique de complexité.

De façon duale à l'analyse de complexité, pour laquelle on regarde la complexité d'un programme donné, il y a la théorie de le complexité, dans laquelle on utilise des modèles machines (comme les machines de Turing) pour étudier la complexité des problèmes. Un objectif principal de la théorie de la complexité est de définir des classes robustes de problèmes ou de fonctions avec une certaine borne de complexité, comme P, NP ou encore PSPACE. Tout comme l'analyse de complexité, la théorie de la complexité prend aussi en compte les effets, avec par exemple des classes probabilistes comme BPP, ou encore des classes parallèles avec la complexité de circuit. Élaborer des langages qui correspondent à une classe de complexité peut aussi être une approche intéressante pour étudier la complexité des programmes. En effet, si après une analyse de complexité, un programme ne satisfait pas la borne désirée, par exemple polynomiale, il n'est pas facile de trouver exactement où est le problème. Ainsi, une idée alternative pourrait être d'écrire directement le programme dans un langage où tous les programmes ont, par construction, une complexité polynomiale. Cette idée de caractériser des classes de complexité avec des systèmes de types ou des logiques est une notion centrale de la *complexité computationnelle*

implicite (ICC). Ces caractérisations donnent en général une forme rudimentaire d'analyse de complexité: si un programme peut être typé, alors il satisfait une certaine borne de complexité. Cependant, en pratique, même si un système de type est assez expressif pour représenter par exemple les machines de Turing polynomiales (ce qu'on appelle couramment *l'expressivité extensionnelle*), cela ne veut pas dire que tous les programmes en temps polynomial peuvent être typés, et il peut être difficile d'écrire des programmes bien typables pour ce système. Ainsi, en plus de cette expressivité extensionnelle, pour l'analyse de complexité on s'intéresserait plutôt à *l'expressivité intentionnelle*, c'est-à-dire l'ensemble des programmes vraiment typables. Il est important de noter que pour l'ICC, l'objectif peut aussi être philosophique: en donnant des caractérisations de classes de complexité, en particulier quand il n'y a pas de bornes explicites, on peut alors mieux comprendre ce que contient intuitivement une classe de complexité.

Dans cette thèse, nous considérons tout d'abord une approche ICC, en utilisant des types à tailles [67], et ensuite nous nous intéresserons à l'analyse de complexité des programmes parallèles écrit dans un calcul de processus concurrent, le π-calcul [90]. Mais tout d'abord, nous allons donner quelque contextes pour les méthodes introduites plus haut.

État de l'art

L'analyse de complexité a été grandement étudiée en informatique, pour différents types de langages et avec différentes méthodes. Dans les langages impératifs, une approche possible est de voir la complexité comme la valeur particulière d'une variable globale d'un programme. Ainsi, on peut analyser la complexité avec des méthodes d'analyses de variables, comme par exemple l'interprétation abstraite [27], l'exécutions symbolique [71], l'étude des invariants, des pré-conditions et de la logique de Hoare, par exemple pour des programmes randomisés [24, 25, 70, 87]. Pour les programmes parallèles, selon la topologie du système, on peut s'intéresser aux graphes de flots de contrôles [3] ou encore au rely-guarantee reasoning [4]...

De son côté, l'utilisation des systèmes de types n'est bien sûr pas limité à l'analyse de complexité, et plusieurs propriétés peuvent être capturées par des systèmes de types. On peut citer par exemples les types intersections, principalement pour la terminaison des programmes [26, 39, 40, 22]. Pour les programmes fonctionnels, on peut citer les types raffinés [49], pour lesquels intuitivement les types de données sont associés à des formules logiques, ce qui permet de dériver des propriétés de sûretés, de secret ou de continuité. [18, 19, 1].

Pour ce qui est des programmes parallèles, les propriétés de sûretés avec des systèmes de types ont été largement étudiées, avec par exemple la terminaison [46, 44], le progrès et l'absence de deadlock, avec par exemple les types comportementaux [50], les types intersections [28], les usages [72, 77, 75, 91] ou encore les types de sessions [47, 66, 53].

Pour l'analyse de complexité par système de types, ici aussi plusieurs approches ont été étudiées selon le langage choisi. Certaines de ses approches ont des liens avec des travaux de l'ICC. Par exemple, avec le système non-size increasing d'Hofmann [61], à partir duquel la l'analyse de complexité amortie par type a été dérivée [62, 63, 68], ce qui a été ensuite largement exploré pour les programmes d'ordres supérieurs [56, 57, 58, 65], les programmes probabilistes [69] et les programmes parallèles [59, 37]. Une autre approche, basée sur les types à tailles [67], où l'idée centrale est de tracer la taille des valeurs des programmes, a donné des résultats intéressants pour les langages fonctionnels [9, 29, 6, 79], les programmes probabilistes [30] et les programmes parallèles [51]

L'analyse de complexité pour les programmes parallèles par des types a aussi été étudiée avec d'autre paradigmes de typages, comme les usages [72], les types comportemantaux [80], et les types de sessions [36, 38, 21, 23].

Dans l'approche duale d'ICC, on peut trouver plusieurs méthodes. Dans les premières caractérisations des fonctions polynomiales en temps [20, 81, 82], une approche par exemple est

la récursion sûre (ou encore le tiering par extension), ce qui a donné plusieurs caractérisations du temps polynomial dans différents contextes [60, 33, 86] et aussi pour des programmes parallèles [54, 84, 45]. Une autre ligne principale de recherche, initiée par la logique linéaire light de Girard [52], sont les systèmes à niveaux inspirés de la logique linéaire, où la duplication est contrôlée par la modalité "!". Ansi, restreindre cette modalité peut donner des bornes de complexité sur l'élimination des coupures. Cela a par exemple permit de définir des langages polynomiaux [52, 11, 85, 17, 78] et aussi des langanges exponentiels ou élémentaires [8, 52, 34, 16]. Une fois encore, cette approche a été utilisée pour des travaux récents sur les programmes parallèles [48, 31, 83]. Pour finir, tracer la taille des valeurs dans les programmes et restreindre la récursion en conséquence peut aussi donner des caractérisations de certains comportements, comme les programmes non-size increasing mentionnés plus haut [61], on encore les programmes qui terminent [67, 29, 32].

Motivations

Dans la première partie de cette thèse, nous adressons un problème de l'approche par logique linéaire de l'ICC. Pour un langage comme la logique linéaire light [52], une façon de prouver la caractérisation de FP (fonctions en temps polynomial) est d'abord de montrer que toutes les exécutions de la logique linéaire light (élimination des coupures) terminent en temps polynomial, et inversement que toutes les fonctions en temps polynomial sont représentables dans cette logique. Ceci est fait habituellement en montrant l'expressivité extensionnelle: un encodage des machines de Turing polynomiales. En revanche, cela ne prend pas bien en compte l'expressivité intentionnelle de la logique, c'est-à-dire l'ensemble des programmes qui peuvent effectivement être typés par cette logique. En général, les approches par niveaux se comportent bien avec les fonctions d'ordres supérieurs, mais la façon de contrôler la récursion est souvent trop restrictive, et donc beaucoup de programmes naturels ne peuvent en fait pas s'écrire dans ces logiques. Un exemple typique est le tri par insertion, qui est évidemment polynomial en temps mais qui a besoin de deux boucles itératives, une sur la liste initiale, et pour chaque élément, une itération pour pouvoir insérer l'élément au bon endroit dans la liste finale. Ces itérations, où le calcul n'augmente pas la taille de la sortie par rapport à l'entrée, ne sont pas bien gérées par les systèmes à niveaux. Cependant, les approches par tailles se comportent bien avec ce type de programmes, par exemple le système non-size increasing [61] était motivé par ces exemples. En revanche, les approches par tailles ont aussi leurs inconvénients, en particulier pour les fonctions d'ordre supérieurs qui peuvent souvent être utilisés qu'une seule et unique fois. Afin de surmonter ce problèmes d'expressivité intentionnelle des systèmes à niveaux, nous élaborons dans la première partie de cette thèse un système de type avec à la fois des tailles et des modalités de la logique linéaire pour obtenir des caractérisations de classes de complexité. La récursion pour les programmes peut soit être contrôlée par la modalité "!" de la logique linéaire, soit par des tailles, ce qui donne plus de flexibilité.

Dans la seconde partie de cette thèse, nous nous intéressons à l'analyse de complexité. Comme expliqué précédemment, l'approche des types à tailles a donnée plusieurs résultats intéressants pour les langages fonctionnels d'ordre supérieurs. Cependant, les applications aux programmes parallèles n'ont pas été très explorées, en particulier pour le π-calcul. Le π-calcul est un langage de processus basé sur les communications dans lequel des valeurs peuvent être envoyés par des canaux. Ce calcul permet aussi la création dynamique de nouveau noms de canaux, et des communiquer ces noms de canaux, ce qui en fait un langage particulièrement expressif. Il est par exemple assez expressif pour représenter des programmes fonctionnels. Cela veut dire en particulier que l'analyse de complexité pour le π-calcul n'est pas triviale, et il est intéressant d'étudier si un système de type à tailles typique pour un langage fonctionnel peut s'étendre au π-calcul. De plus, dans ce calcul, plusieurs notions de complexité peuvent être

étudiées, et donc plusieurs systèmes de types peuvent être proposés. Une notion plutôt simple de complexité est le *travail*, le temps de calcul total d'un processus. Une notion alternative, plus spécifique aux langages parallèles est la *profondeur*, qui correspond à la complexité sous l'hypothèse de parallélisme maximal: les communications sur des canaux peuvent se faire simultanément. Cette seconde notion de complexité, aussi appelée *complexité parallèle*, a été définie de plusieurs façon dans la littérature. Une première approche est le *progrès maximal*, qui dit intuitivement qu'à chaque étape, on fait toutes les communications qui sont disponibles. Cela permet en effet de définit une sorte de complexité parallèle, mais à cause de cette définition, une course apparaît dans le calul: quand il y a un choix non-déterministe pour une communication, la plus rapide est toujours déclenchée. Cela peut être intéressant en pratique, mais en théorie certains chemins de réductions sont bloqués, ce qui peut être insatisfaisant. Une autre notion de complexité qui prend en compte entièrement ce non-déterminisme est la complexité causale, introduite dans [41, 42, 43] et récemment utilisée dans [45]. Cette notion de complexité se comporte bien, mais elle peut être difficile d'utilisation comme elle se base sur l'analyse des traces, avec une relation de causalité. Il peut donc être intéressant de donner une présentation alternative de la profondeur, facile d'utilisation afin d'avoir des preuves plus simples. Dans la seconde partie de cette thèse, nous nous intéressons au travail et à la profondeur, et nous proposons un cadre pour l'analyse de complexité dans le π-calcul.

Notre approche

Dans cette thèse, notre approche se base sur les types à tailles [67, 29]. L'idée centrale des types à tailles est de tracer la taille des types de données dans un programme. C'est une approche intéressante pour l'analyse de complexité car elle permet la compositionnalité. En effet, supposons données deux fonctions f et g, pour lesquelles nous connaissons une borne de complexité en temps sur les entrées de tailles n, $C_f(n)$ et $C_g(n)$. Peut-on en déduire une borne sur la complexité de $f \circ g$? Avec seulement ces informations, le mieux que l'on puisse faire est de supposer, par définition, que la taille de $g(n)$ est plus petite que $C_g(n)$ et donc que la complexité totale sur une entrée de taille n est d'abord de calculer $g(n)$ $(C_g(n))$, et ensuite de calculer $f(g(n))$ $(C_f(C_g(n)))$. Cette borne n'est pas très satisfaisante, en effet si g est, par exemple, une fonction qui n'augmente pas la taille de son entrée mais en temps quadratique, alors on obtient une borne en $O(n^4)$ au lieu de $O(n^2)$. Ainsi, connaître la taille de la sortie de g est nécessaire pour avoir une borne de complexité plus précise. De plus, contrôler la taille des valeurs peut aussi aider à contrôler le nombre de boucles dans un programme, ou le nombre d'appels récursifs à une fonction (en particulier avec une induction structurelle). C'est pourquoi les types à tailles peuvent être intéressants pour l'analyse de complexité.

Dans cette thèse, nous allons donner deux types de résultats. Dans la première partie, nous avons une approche ICC de la complexité, où nous donnons des caractérisations de classes de complexité grâce à un langage équipé d'un système de type. Ainsi, notre cible principale est le langage en entier, donnant une borne de complexité générique pour n'importe quel programme qui peut être écrit dans celui-ci. Dans la seconde partie, nous avons une approche d'analyse de complexité, où l'objectif est d'élaborer des systèmes de types à partir desquels ont peut dériver une borne (précise) sur la complexité d'un programme typé. Donc la cible n'est plus le langage en entier mais une analyse de complexité précise de chaque programme, cas par cas. Toutefois, pour ces deux résultats, il y a des similarités, et en particulier il y a un théorème de *correction* à prouver, disant que la borne de complexité donnée par le système de type est bien correcte. Dans notre approche, ce théorème de correction est toujours prouvé avec la même méthodologie: nous montrons que le système de type satisfait la *réduction du sujet*, ce qui veut dire qu'évaluer un programme ne change pas son type. Comme nous nous intéressons à la complexité, la réduction du sujet doit aussi satisfaire des propriétés quantitatives: si une étape

de réduction $M \to N$ décroît la complexité totale du programme, alors la borne dérivée pour le typage de N doit être plus petite que la borne dérivée pour le typage de M.

Comme expliquée précédemment, dans la première partie de cette thèse, nous considérons une approche ICC de la complexité. Notre but est de donner des caractérisations de classes de complexité, où la récursion peut être contrôlée soit par la modalité de logique linéaire ! soit par des tailles. Dans ce travail, nous utilisons la logique linéaire élémentaire (ELL) [52, 34], que l'on associe avec des types à tailles inspirés de [9]. La logique linéaire élémentaire est intuitivement la logique linéaire sans la co-unit ($!A \multimap A$) et la co-multiplication ($!A \multimap !!A$). Avec ces modifications, la *profondeur* d'un type, c'est-à-dire le nombre de ! au dessus d'un type, devient une informations importante pour l'analyse de complexité. Par exemple, avec un type W pour les mots, alors dans le type des fonctions des mots vers les mots $!\mathsf{W} \multimap !^k\mathsf{W}$, plus k est grand, plus la fonction dispose de ressources. En particulier, il est montré dans [8] qu'en présence de type récursifs, cela correspond à la classe de complexité k-FEXPTIME (fonctions de complexité $2^{2^{\cdots^{2^{poly(n)}}}}$, où k est la hauteur de cette tour d'exponentielle). Afin d'améliorer la faible expressivité intentionnelle de ELL, nous ajoutons dans le type $!W \multimap !W$ quelques fonctions polynomiales du premier-ordre, obtenues grâce à un système de types à tailles. Nous montrons alors qu'avec cette extension, le système de type donne une caractérisation de PTIME et $2k$-EXPTIME, encore une fois selon la profondeur de la sortie, avec une expressivité intentionnelle vraisemblablement meilleure et sans avoir besoin de types récursifs.

Dans la seconde partie de cette thèse, nous nous intéressons à l'analyse de complexité dans le π-calcul. Nous présentons sur un langage fonctionnel un cadre uniforme pour les types à tailles, inspiré par [6, 29], que nous modifions afin de capturer différentes notions de complexité dans le π-calcul: le travail et la profondeur. Cette approche se base sur des expressions entières afin de décrire les tailles des types de données. Par exemple, un entier n pourrait avoir un type $\mathsf{Nat}[I, J]$, indiquant ainsi que cette entier satisfait $I \leq n \leq J$, où I et J sont des expressions entières. De plus, nous utilisons une sorte de polymorphisme sur les tailles pour les fonctions récursives. Par exemple, une fonction qui n'augmente pas la taille de son entrée pourrait être typée avec $\forall i, \mathsf{Nat}[0, i] \to \mathsf{Nat}[0, i]$. Cette approche, à laquelle on ajoute des informations supplémentaires de complexité, est suffisante aussi bien pour le langage fonctionnel que pour le travail dans le π-calcul.

Cependant, pour l'analyse de la profondeur, l'extension n'est pas si simple. Tout d'abord, afin de simplifier les preuves, nous donnons une présentation de la profondeur, notion similaire à la complexité causale, avec une sémantique petit pas simple, ce qui permet encore une fois au théorème de correction d'être prouvé par réduction du sujet. Ensuite, nous enrichissons les types à taille avec des informations de temps, inspirés par [36]. Ces informations temporelles sont nécessaires pour pouvoir traiter la synchronisation des canaux, qui peuvent communiquer simultanément dans une analyse de profondeur. Dans une première approche, nous nous basons sur un système de type entrée/sortie [90]. Si cette approche semble assez expressive pour les programmes parallèles, elle ne permet pas l'analyse de quelques comportements concurrents comme les sémaphores. Ainsi, afin de prendre en compte ceux-ci, nous élaborons dans une seconde approche un système de type avec usages [72] enrichis avec des tailles pour l'analyse de complexité, en collaboration avec Naoki Kobayashi (Université de Tokyo).

Plan

Dans le Chapitre 2, nous décrivons le calcul basé sur la logique linéaire élémentaire avec taille, et nous donnons le théorème de caractérisation. Dans le Chapitre 3, nous décrivons un cadre pour les types à tailles sur un langage fonctionnel, que nous allons utiliser dans le reste de la thèse. Nous donnons également une méthode de preuve pour le théorème de correction. Dans

le Chapitre 4, nous présentons d'abords quelques définitions préliminaires sur le π-calcul et quelques paradigmes de typages que nous utilisons dans cette thèse. Puis, de la Section 4.2 jusqu'à la fin de la thèse, nous décrivons nos contributions pour l'analyse de complexité dans le π-calcul, avec un système de type pour le travail, la définition de la profondeur et le système de type entrée/sortie pour la profondeur, suivi de celui avec les usages.

1.2 English Introduction

Context

A simple yet challenging question in computer science is the comparison of programs: given two programs computing the same function, is one more efficient than the other? A simple way to do this comparison could be to choose a programming language, implement the two programs, let them run both on multiple inputs and then measure *dynamically* the resource consumption of those executions. This method however depends on some low-level details such as the hardware or the choice of implementation, and it is not feasible if a program consumes too many resources, or if, for example, it does not terminate. Thus, in order to compare programs theoretically, one may prefer to use a *static* method such as *complexity analysis*. The goal of complexity analysis is, given a program, to statically approximate the resources it needs to compute its output on a given input. The usual resources that are considered are *space* and *time*. Space complexity corresponds to the amount of memory space required to run the program, and time complexity corresponds to the execution time of the program. Usually, the analysis of time complexity relies on abstracting the notion of time (for example, an arithmetic multiplication = one unit of time...). Such as the analysis of termination, complexity analysis is undecidable, however, it remains an interesting and challenging problem.

In practice, given a language or an abstract programming language, for time complexity analysis one first needs to abstract the notion of time to carry the analysis, and different choices are possible depending on the language. For example, in a functional abstract language such as an extension of λ-calculus, one could take as a time unit a reduction step, or computation time on an abstract machine. If a program interacts with a network, one may be interested in the number of network accesses, as it could be a costly operation in practice. Also, depending on the effects in a language, one could be interested in different notions of complexity. For instance, in a non-deterministic setting, one could choose the worst-case complexity or the complexity under a specific reduction strategy, in presence of probabilities, one could consider expected time complexity or the probability of satisfying a specific complexity bound, in presence of parallelism, one could examine the total sequential complexity, usually called *work*, or other notions of parallel complexity...

For parallel programs, an interesting notion for instance could be the complexity with p processors, meaning intuitively that p computations can be done simultaneously in parallel, which can correspond to the practical complexity of a program that runs on a multicore system. In a more theoretical point of view, one could consider an infinite number of processors, which would give the complexity under maximal parallelism, usually called *span*. The span is in general not feasible in practice, however it has some theoretical interest since a well-known result [55] says that from the work w and the span s of a program in a certain class, one can derive a bound on the execution time with p processors, in $O(\max(w/p), s)$. Intuitively, the span gives some information about how efficient it would be run the program in parallel, and a span analysis could be completed with an analysis of how many processors are needed to obtain the span.

Complexity analysis of programs has been extensively studied, and several approaches have been explored to carry the analysis. Among all the different approaches, one that interests

us particularly is the type systems approach. Usually, a type system is a way to abstract programs in order to guarantee some properties, such that *typed programs cannot go wrong*. A type system analysis has some interesting benefits compared to other approaches, as it allows for *compositionality* and *modularity*. Indeed, generally a type system consists in a set of rules corresponding to simple constructors of the language, thus the analysis of the whole program relies on the analysis of all the subprograms, which can then be reused in different contexts. This approach is especially useful for systems with a built-in notion of composition, such as functional languages, where complexity analysis must take into account the composition and the reusability of functions. Once a type system is designed for a specific property of a language, two main questions arise: *type-checking* and *type inference*. Type-checking consists in, given a program and a candidate typing derivation for this program, verifying automatically that this type derivation is correct. For the examined property, this corresponds to checking a certificate for this property. Type inference consists in, given a program, finding automatically a type derivation for this program. For the inspected property, this corresponds to automatically verifying if a program satisfies the property. Type inference is thus harder than type checking and usually, there is a kind of trade-off for type systems: the more expressive the type system (the more programs it can type), the more complex it is to type-check a program or to infer its type. In the particular case of complexity analysis, the goal is, from a typing derivation of a program, to extract a bound on its complexity. Thus, type-checking gives a complexity certificate, and type inference gives automatic complexity analysis.

Dually to complexity analysis, where given a program, one looks at the complexity of this program, there is computational complexity theory, where one uses machine models (originally Turing machines) to talk about complexity of problems. One goal of computational complexity theory is to define robust classes of problems, or functions, with some complexity bound, such as P, NP or PSPACE. As for complexity analysis, computational complexity theory also considers some effects, with for example probabilistic classes such as BPP, or parallel classes with circuit complexity. Designing languages corresponding to a complexity class can also be an interesting approach when studying complexity of programs. Indeed, if after some complexity analysis, a program does not have the desired complexity, for example polynomial time, it is not direct to pinpoint specifically where the problem is. An alternative could be to directly try to write the program in a language where every program has, by construction, polynomial time complexity. This idea of characterizing complexity classes with type systems or logics is a central notion of *implicit computational complexity* (ICC). Those characterizations of complexity classes may also provide a rudimentary form of complexity analysis: if a program can be typed, it satisfies in particular some complexity bound. However, in practice, even if a type system is expressive enough to encode for example polynomial time Turing machines (we call this the *extensional completeness*), it does not mean all polynomial time programs can be typed, and it may not be easy to actually write well-typed programs for this type system. Thus, in addition to extensional completeness, for complexity analysis one may want to study *intensional expressivity*, i.e. the set of actually typable programs. Moreover, in ICC, the goal may also be philosophical: by giving characterizations of complexity classes, especially when there is no explicit complexity bound, one could have a better understanding of what contains a complexity class.

In this thesis, we first consider an ICC approach, using sized types [67], and then we focus on complexity analysis of parallel programs, written in a concurrent process calculus, the π-calculus [90]. But first, let us provide some background on the methods introduced above.

State of the art

Complexity analysis has been largely studied in computer science, for different kinds of languages, and with different methods. In imperative programming languages, an approach is to

see complexity as the particular value of a global variable of the program, and so, one can carry complexity analysis by using methods for the analysis of variables, with for example abstract interpretation [27], symbolic execution [71], invariants, weakest pre-condition and Hoare logic, also useful in presence of probabilities [24, 25, 70, 87]. As for parallel programs, depending on the topology of system, one could do flow graph analysis [3] or rely-guarantee reasoning [4]...

The use of type systems is of course not limited to complexity analysis, as several properties can be enforced by type systems. There are for instance intersection types, mainly for characterizing termination of programs [26, 39, 40, 22]. For functional programs, an interesting approach is refinement types [49], where intuitively data-types are associated with formulas, allowing to derive many safety properties, or even privacy or continuity [18, 19, 1].

Concerning parallel programs, safety properties with type systems have been largely studied, such as termination [46, 44], deadlock-freedom or progress, with for example behavioural types [50], intersection types [28], usages [72, 77, 75, 91] or session types [47, 66, 53].

As for complexity analysis by type systems, several approaches have been studied depending on the chosen language. Some of those approaches have a link with work on ICC. For example, from the non-size increasing system of Hofmann [61], the type-based amortized cost analysis was derived [62, 63, 68], which was then largely explored for higher order programs [56, 57, 58, 65], probabilistic programs [69] and for parallel programs [59, 37]. Another approach, based on sized types [67], where the main idea is to track the values of sizes in programs, has led to interesting results for functional programs [9, 29, 6, 79], for probabilistic programs [30] and parallel programs [51]

Complexity analysis of parallel programs by types has also been studied with some other type paradigms, such as usages [72], behavioural types [80], session types [36, 38] and multiparty session types [21, 23].

In the dual reasoning approach of ICC, we can find several approaches. In the early characterizations of poly-time functions [20, 81, 82], one approach is for instance safe recursion (or tiering by extension), which led to several characterizations of polynomial time in different contexts [60, 33, 86] and for parallel programs [54, 84, 45]. Another main line of research, initiated with Girard's light linear logic [52], are the level-based systems inspired by linear logic, where duplication is controlled by the ! modality, and so restraining this modality can lead to complexity bounds on cut elimination. This for example led to polytime languages [52, 11, 85, 17, 78], and exponential and elementary languages [8, 52, 34, 16]. Again, this approach was also used on recent work for parallel programs [48, 31, 83]. Finally, the approach of tracking sizes in programs, and restrict recursions, can also give characterizations of complexity classes, such as the non-size increasing programs mentioned above [61], or simply terminating programs [67, 29, 32].

Motivations

In the first part of this thesis, we address a problem of the linear logic approach to ICC. For a language such as light linear logic [52], the way to prove a characterization of FP (polynomial time functions) is to first show that all executions of light linear logic (cut-elimination) terminate in polynomial time, and conversely, that all polynomial time functions can be represented in this logic. This is usually done by showing the extensional completeness: an encoding of polynomial time input/output Turing machines. However, this does not account well for the intensional expressivity of the logic, i.e. the set of programs that can effectively be typed by this logic. Generally, level-based approaches behave well with higher-order functions, but the way recursion is controlled is often too restrictive, and thus many commons programs are in fact not expressible in those logics. A typical example is the insertion sort, which is obviously a polynomial time algorithm but needs two iterative loops, one iteration that ranges over the initial list, and for each element, an iteration to insert it in the final list. Those kinds of iterations, where in fact

the computation does not increase the size of the input, is not handled well by level-based systems. Nevertheless, the sized-based approach behaves well with this category of programs, for example the non-size increasing type system [61] was motivated by those examples. However, size-based approaches also have their limitations, especially with higher-order functions as it is often assumed that programs are linear, i.e. all higher-order functions can be used at most once. In order to tackle this problem of intensional expressivity of level-based systems, we design in the first part of this thesis a type system with both sizes and linear logic modalities to obtain characterizations of complexity classes, where recursion can be either controlled by the linear logic modality ! or by sizes, thereby giving more flexibility.

In the second part of this thesis, we focus on complexity analysis. As explained before, the sized types approach led to many interesting results in higher-order functional languages. Still, the applications to parallel languages have not been explored yet, especially for π-calculus. The π-calculus is a communication-based process language where values can be sent on channels. This calculus also allows for dynamic creation of channel names and name-passing, which makes it a particularly expressive language. It is for example expressive enough to encode some functional programs. Thus, this means complexity analysis in π-calculus is not trivial, and it is interesting to study if a typical sized type system for a functional language could be extended to π-calculus. Moreover, in this calculus several notions of complexity could make sense, and thus several type systems can be proposed corresponding to different notions of complexity. A simple notion of complexity is the *work*, the total computation time of a process. An alternative notion, and more specific to parallel languages, is the *span*, corresponding to complexity under maximal parallelism: communications on channels can happen simultaneously. This second notion of complexity, also called *parallel complexity*, has been defined in various ways in the literature. A first approach is *maximal progress*, stating intuitively that at each step, we trigger all communications that are available. This indeed defines a notion of maximal parallelism. Although, because of this, a notion of race appears in the calculus: when there is a choice for a communication, the faster one is always triggered, and the slower one never happens. This could be interesting in practice, but it theoretically blocks some reduction paths, which can be unsatisfactory. Another notion of complexity that accounts for full non-determinism is causal complexity, introduced in [41, 42, 43] and recently used in [45]. This notion of complexity behaves well, but it may not be easy to work with, as it relies on the analysis of *traces*, with a causality relation. Thus, it could be interesting to give an alternative presentation of span, easy to work with in order to have simpler proofs. In the second part of this thesis, we will focus on both work and span, and propose a sized types framework for complexity analysis in the π-calculus.

Our approach

In this thesis, our approach relies on sized types [67, 29]. The basic idea of sized types is to track sizes of data-types in a program. This is an interesting approach for complexity analysis as it allows for compositionality. Indeed, suppose given two functions f and g, for which we know some time complexity bound on inputs of size n, $C_f(n)$ and $C_g(n)$. Can we derive a bound on the complexity of $f \circ g$? With only this information, the best we can assume is, by definition, that the size of $g(n)$ is smaller than $C_g(n)$, and so the total complexity on an input of size n is first computing $g(n)$ $(C_g(n))$, and then computing $f(g(n))$, $(C_f(C_g(n)))$. This bound is unsatisfactory, indeed if g is, for example a non-size increasing function in quadratic time, and f is also in quadratic time, then we obtain a bound in $O(n^4)$ instead of $O(n^2)$. Thus, knowing the size of the output of g is necessary to have precise complexity bounds. Moreover, controlling the size of values can also help controlling the number of loops in a program, or the number of recursive calls to a function (especially with structural induction). That is why sized types can

be a good start for complexity analysis.

In this thesis, we will give two kinds of results. In the first part, we focus on an ICC approach of complexity, where we give a characterization of complexity classes by a language equipped with a type system. Thus, we focus on the language as a whole, giving a generic complexity bound for any programs that can be written in it. In the second part, we have a complexity analysis approach, where the goal is to design type systems from which one can derive a (precise) bound on the complexity of the typed program. So, the focus is no longer on the language as a whole, but on a precise complexity analysis for each program, case by case. Nevertheless, for both results, there are some similarities, and in particular there is a *soundness* theorem to prove, stating that the complexity bound given by the type system is indeed correct. In our approach, this soundness theorem is always proved with the same methodology: we show that the type system satisfies *subject reduction*, that is to say evaluating a program does not change the type of this program. As we focus on complexity, subject reduction must also satisfy some quantitative properties: if a reduction step $M \to N$ decreases the overall complexity of a program, then the bound derived from the typing for N should be smaller than the bound derived from the typing for M.

As stated before, in the first part of this thesis we consider an ICC approach to complexity. Our goal is to give a characterization of complexity classes, where recursion can be either controlled with the linear logic modality !, or with sizes. In this work, we use for the level-based system the elementary linear logic (ELL) [52, 34], combined with sized types inspired from [9]. Elementary linear logic is intuitively linear logic without the co-unit ($!A \multimap A$) and the co-multiplication ($!A \multimap !!A$). With these modifications, the *depth* of a type, that is to say the number of ! above a type, becomes a relevant information for complexity analysis. For example, with a type W for words, then in the type of functions from words to words $!\mathsf{W} \multimap !^k\mathsf{W}$, the greater k is, the more computational power the function can have. In particular, it is shown in [8] that with recursive type, this corresponds to the k-EXPTIME complexity class (function of complexity $2^{2^{\cdots^{2^{poly(n)}}}}$, where k is the height of this tower of exponentials). In order to improve the poor intensional expressivity of ELL, we add in the type $!W \multimap !W$ some first-order polynomial functions, obtained with a sized type system. We then show that with this extension, the type system gives a characterization of PTIME and $2k$-EXPTIME, again depending on the depth of the output, with a seemingly better intensional expressivity.

In the second part of this thesis, we focus on complexity analysis of the π-calculus. We present on a functional language a uniform framework for sized types, inspired by [6, 29], that we modify to capture different notions of complexity in the π-calculus, namely work and span. This approach relies on integers expressions in order to describe sizes of data-types. For example, an integer n could be given a type $\mathsf{Nat}[I, J]$, stating that this integer satisfies $I \leq n \leq J$, where I and J are integer expressions. Moreover, we use some kind of polymorphism on sizes to handle recursive functions. For example, a non-size increasing function could be typed with $\forall i, \mathsf{Nat}[0, i] \to \mathsf{Nat}[0, i]$. This approach, equipped with additional complexity information, is sufficient for both functional languages and work.

However, for the analysis of span in the π-calculus, the extension is not as simple. First, in order to simplify the proofs, we give a presentation of span, similar to a kind of causal complexity, described by a simple small-step semantics, allowing again the soundness theorem to be proved by subject reduction. Then, we enrich sized-types with time information, inspired by [36]. This time information is necessary to handle synchronization of channels, that can communicate simultaneously in a span analysis. In a first approach, we rely on an input/output type system [90]. If this approach seems expressive enough for parallel programs, it cannot handle some concurrent behaviours, such as semaphore. Thus, in order to account for those, we design in a second approach a usage type system [72] with sizes for complexity analysis, in

collaboration with Naoki Kobayashi (University of Tokyo).

Outline

In Chapter 2, we describe the calculus based on elementary linear logic with sizes, and we give the characterization theorem. In Chapter 3, we describe a sized type framework for a functional language, that we will use throughout the thesis. We also give the proof methodology for the soundness theorem. In Chapter 4, we first give some preliminaries on π-calculus and some type systems paradigms that we use in this thesis, and from Section 4.2 to the end, we describe our contributions for complexity analysis in π-calculus, with a type system for work, the definition of span and the input/output type system for span, followed by the usage type system for span.

Chapter 2

Combining Linear Logic and Sized Types for Implicit Complexity

The light logic approach for characterizing complexity classes, initiated by Girard [52], led to many interesting results for implicit computational complexity. This approach relies on the Curry-Howard correspondence, stating the link between proofs and programs. Thus, from a logic such as (intuitionistic) linear logic, we can derive a programming language where a proof corresponds to a program, and cut-elimination corresponds to reduction. In linear logic in particular, a focus is given on resource consumption: a formula can be used exactly once in a proof, and both duplication and erasure of formulas are controlled by modalities ("!" and its dual "?"). With the Curry-Howard correspondence, those modalities could also control erasure and duplication of resources, which is useful to carry a complexity analysis.

In particular, Girard's approach for characterizing complexity classes consists in restraining the power of the ! modality, controlling duplication, and thus restraining the overall complexity of the logic. Several logics have been derived in this way, such as light linear logic [52] and soft linear logic [78] for polynomial time, and elementary linear logic [52, 34] for elementary time.

As stated before however, those logics suffer from poor intensional expressivity: even if all functions of the complexity class can be represented, the actual set of programs that are expressible is not large enough, and the languages derived from the logic do not allow to write programs in a natural way. For light linear logic, a functional language with recursive definitions and pattern-matching has been designed [11]. This language satisfied the expected polytime complexity bound, however the proof was rather tedious and did not seem robust enough to support other extensions of the language. A similar question could be asked for elementary linear logic: is the characterization robust enough to support extensions/enrichments?

To answer this question, we show that enriching the usual type $\mathsf{W} \multimap \mathsf{W}$ from words to words of elementary linear logic by some polynomial time functions leads to characterizations of the complexity classes $2k$-EXPTIME, for $k \geq 0$, where k corresponds to the depth of the output : $!\mathsf{W} \multimap !^{k+1}\mathsf{Bool}$.

An advantage of this characterization is that, depending on how the polynomial-time first-order functions are defined, it can give a language in which it is easier to write programs, compared to usual elementary linear logic. Thus, we choose for this language a linear λ-calculus with primitive recursion and sized types, called $\mathsf{s}\ell\mathsf{T}$. This language is inspired by [9], and we restrict sizes to polynomials to enforce polynomial time complexity. Note that this choice is a bit arbitrary, as the results could be adapted for other polynomial-time languages. The overall enriched elementary linear calculus corresponds to the usual elementary linear logic, equipped with an additional specific constructor in order to use a first-order function of $\mathsf{s}\ell\mathsf{T}$. We call this new language sEAL.

As the goal of this chapter is the characterization of complexity classes, we are not interested in deriving precise and practical complexity bounds for programs, our only consideration is that those complexity bounds must correspond to the complexity class. Because of this, the sized types we use in this chapter differ from the sized types we will use for complexity analysis, where precise bounds are desired.

The bounds we derive are over-approximations of the number of reduction steps to reach a normal form, where we define a *weight* on a program from a typing, and we show that this weight strictly decreases in a reduction step. The polynomial weight for $\mathsf{s}\ell\mathsf{T}$ is inspired by [9] and then the elementary weight for sEAL is inspired by [83]. After showing this soundness theorem, we proceed to illustrate the expressivity of the type system on some examples, and we show how to encode Turing machines in this language, leading to the characterization of complexity classes. Those results have been published in [12, 13].

2.1 A Polynomial Time Language with Sizes: $\mathsf{s}\ell\mathsf{T}$

We present $\mathsf{s}\ell\mathsf{T}$ (for sized linear system T) which is a linear λ-calculus with constructors for base types and a constructor for high-order primitive recursion. Types are enriched with a polynomial index describing the size of the value represented by a term, and this index imposes a restriction on recursions. With this, we are able to derive a weight on terms in order to control the number of reduction steps.

2.1.1 Syntax, Semantics and Type System

Definition 2.1.1. *The set of* terms *and* values *of* $\mathsf{s}\ell\mathsf{T}$ *are defined by the following grammars:*

$$\begin{aligned} t, u ::= {} & x \mid \lambda x.t \mid t\ u \mid t \otimes u \mid \mathtt{let}\ x \otimes y = t\ \mathtt{in}\ u \mid \underline{0} \mid \mathtt{s}(t) \mid \mathtt{ifn}(t,u) \mid \mathtt{itern}(V,t) \\ & \mid \epsilon \mid \mathtt{s}_i(t) \mid \mathtt{ifw}(t_0,t_1,u) \mid \mathtt{iterw}(V_0,V_1,t) \mid \mathtt{tt} \mid \mathtt{ff} \mid \mathtt{if}(t,u) \end{aligned}$$

$$\begin{aligned} V, W ::= {} & x \mid \lambda x.t \mid V \otimes W \mid \underline{0} \mid \mathtt{s}(V) \mid \mathtt{ifn}(V,W) \mid \mathtt{itern}(V,W) \\ & \mid \epsilon \mid \mathtt{s}_i(V) \mid \mathtt{ifw}(V_0,V_1,W) \mid \mathtt{iterw}(V_0,V_1,W) \mid \mathtt{tt} \mid \mathtt{ff} \mid \mathtt{if}(V,W) \end{aligned}$$

with $i \in \{0,1\}$.

We define free variables and free occurrences as usual, and we work up to α-renaming. Here, we choose the alphabet $\{0,1\}$ for simplification, but we could have taken any finite alphabet Σ and in this case, the constructors `ifw` and `iterw` would need a term for each letter.

The definitions of those constructors will be more explicit with their reduction rules and their types. For intuition, the constructor $\mathtt{ifn}(t,t')$ defines a function on integers that does a pattern matching on its input, and the constructor $\mathtt{itern}(V,t)$ is such that $\mathtt{itern}(V,t)\ \underline{n} \to^* V^n\ t$, if $\underline{n}$ is the coding of the integer n, that is $\mathtt{s}^n(\underline{0})$.

Definition 2.1.2 (Substitution)**.** *For an object t with a notion of free variable and substitution we write $t\{t'/x\}$ the term t in which free occurrences of x have been replaced by t'.*

Base reductions in $\mathsf{s}\ell\mathsf{T}$ are given by the rules described in Figure 2.1. Note that in the `iterw` rule, the order in which we apply the iterated functions is the reverse of the one for iterators we see usually. In particular, it does not correspond to the reduction defined in [9]. Those base reductions can be applied in contexts C defined by the following grammar:

$$\begin{aligned} C := {} & [] \mid C\ t \mid V\ C \mid C \otimes t \mid t \otimes C \mid \mathtt{let}\ x \otimes y = C\ \mathtt{in}\ t \mid \mathtt{s}(C) \mid \mathtt{ifn}(C,t) \mid \mathtt{ifn}(V,C) \mid \mathtt{itern}(V,C) \\ & \mid \mathtt{s}_i(C) \mid \mathtt{ifw}(C,t,u) \mid \mathtt{ifw}(t,C,u) \mid \mathtt{ifw}(V,W,C) \mid \mathtt{iterw}(V_0,V_1,C) \mid \mathtt{if}(C,t) \mid \mathtt{if}(t,C). \end{aligned}$$

$\mathtt{if}(V,W)\ \mathtt{tt}$	$\to V$	$(\lambda x.t)\ V$	$\to t\{V/x\}$
$\mathtt{if}(V,W)\ \mathtt{ff}$	$\to W$	$\mathtt{let}\ x\otimes y = V\otimes W\ \mathtt{in}\ t$	$\to t\{V/x\}\{W/y\}$
$\mathtt{ifn}(V,W)\ \underline{0}$	$\to W$	$\mathtt{ifn}(V,V')\ \mathtt{s}(W)$	$\to V\ W$
$\mathtt{itern}(V,W)\ \underline{0}$	$\to W$	$\mathtt{itern}(V,V')\ \mathtt{s}(W)$	$\to \mathtt{itern}(V, V\ V')\ W$
$\mathtt{ifw}(V_0,V_1,W)\ \epsilon$	$\to W$	$\mathtt{ifw}(V_0,V_1,V')\ \mathtt{s}_i(W)$	$\to V_i\ W$
$\mathtt{iterw}(V_0,V_1,W)\ \epsilon$	$\to W$	$\mathtt{iterw}(V_0,V_1,V')\ \mathtt{s}_i(W)$	$\to \mathtt{iterw}(V_0,V_1,V_i\ V')\ W$

Figure 2.1: Base Reduction Rules for $\mathsf{s}\ell\mathsf{T}$.

We introduce the system of linear types with sizes. First, base types are given by the following grammar:

$$U := \mathsf{W}^I \mid \mathsf{N}^I \mid \mathsf{B} \qquad\qquad I, J := i \mid n \in \mathbb{N}^* \mid I{+}J \mid I \cdot J$$

$\mathbb{N}^*$ is the set of non-zero integers. I represents an *index* and i represents an *index variable*. We define for indices the notions of occurrences of a variable in the usual way, and we work up to renaming of variables. We also define the substitution of a variable in an index in the usual way. Then, we can generalize substitution to types, for example $\mathsf{N}^I\{J/a\} = \mathsf{N}^{I\{J/a\}}$. The intended meaning is that closed values of type N^I (resp. W^I) will be integers (resp. words) of size (resp. length) at most I. Usually, we use a finite set of index variables φ in indexes.

Definition 2.1.3 (Order on indices)**.** *For two indices I and J, we say that $I \leq J$ if for any valuation $\rho : \varphi \mapsto \mathbb{N}^*$ we have $I_\rho \leq J_\rho$ where I_ρ is I where all index variables have been replaced by their value in ρ, thus I_ρ is a non-zero integer. We also consider that if $I \leq J$ and $J \leq I$ then $I = J$ (i.e. we take the quotient set for the equivalence relation). Remark that by definition of indices, we always have $1 \leq I$.*

For two indices I and J, we say that $I < J$ if for any valuation ρ, we have $I_\rho < J_\rho$. This is not equivalent to $I \leq J$ and $I \neq J$, as we can see with $i \leq i \cdot j$.

For example, we have $i{+}1 \leq 2 \cdot i$, $i{+}j \cdot i = (j{+}1) \cdot i$ and $i{+}1 < i{+}j{+}k$. Here we only consider polynomial indices. This is a severe restriction w.r.t. usual indices, that we will use later and that can be found in [79, 9], in which indices can use any set of functions along with a semantic interpretation. But in the present setting this is sufficient because we only want $\mathsf{s}\ell\mathsf{T}$ to characterize polynomial time computation.

Definition 2.1.4. *Types are given by the grammar:*

$$D, E, F := U \mid D \multimap E \mid D \otimes E.$$

The subtyping order $\sqsubseteq$ on those types is described in Figure 2.2. This definition allows for example the subtyping $\mathsf{N}^{i+1} \multimap \mathsf{W}^{2i} \sqsubseteq \mathsf{N}^i \multimap \mathsf{W}^{3i}$, meaning that a function taking an integer of size smaller than $i{+}1$ and returning a word of size at most $2i$ can also be seen as a function taking an integer of size smaller than i and returning a word of size at most $3i$.

Definition 2.1.5 (Variable Contexts)**.** Variables contexts *are denoted Γ, with the shape $\Gamma \equiv x_1 : D_1, \ldots, x_n : D_n$. We say that $\Gamma \sqsubseteq \Gamma'$ when Γ and Γ' have exactly the same variables, and for $x : D$ in Γ and $x : D'$ in Γ' we have $D \sqsubseteq D'$.* Ground variables contexts*, denoted $d\Gamma$, are variables contexts in which all types are base types. We write $\Gamma = \Gamma', d\Gamma$ to denote the decomposition of Γ into a ground variable context $d\Gamma$ and a variable context Γ' in which types are non-base types. This allows us to decompose a context into his duplicable variables $d\Gamma$ and the non-duplicable ones. For a variable context without base type, we denote $\Gamma = \Gamma_1, \Gamma_2$ when Γ is the concatenation of Γ_1 and Γ_2, and Γ_1 and Γ_2 do not have any common variables.*

$$\frac{}{\mathsf{B} \sqsubseteq \mathsf{B}} \qquad \frac{I \leq J}{\mathsf{N}^I \sqsubseteq \mathsf{N}^J} \qquad \frac{I \leq J}{\mathsf{W}^I \sqsubseteq \mathsf{W}^I}$$

$$\frac{E \sqsubseteq D \qquad D' \sqsubseteq E'}{D \multimap D' \sqsubseteq E \multimap E'} \qquad \frac{D \sqsubseteq E \qquad D' \sqsubseteq E'}{D \otimes D' \sqsubseteq E \otimes E'}$$

Figure 2.2: Subtyping Rules.

We denote proofs as $\pi \lhd \Gamma \vdash t : D$ and we define an index $\omega(\pi)$ called the *weight* for such a proof. The idea is that the weight will be an upper-bound for the number of reduction steps of t. Note that since $\omega(\pi)$ is an index, this bound can depend on some index variables. As the goal of this section is to ensure that the language is polynomial, we do not pay attention to the precision of this upper bound, we only need it to be polynomial. The rules for those proofs are described by Figure 2.3. Here are some remarks:

- Observe that this system enforces a linear usage of variables of non-base types, this can be seen for instance in binary rules, such as application, where non-base variables (those in Γ, Γ') do not occur both in t and u.
- In the rule for `itern` and `iterw` described in Figure 2.3, the index variable i must be a fresh variable. Then $d\Gamma \vdash V : D \multimap D\{i{+}1/i\}$ means intuitively that for any index J, we have $d\Gamma \vdash V : D\{J/i\} \multimap D\{J{+}1/i\}$. This will be formalized in Lemma 2.1.3. Also, we need some monotonicity with respect to i (expressed here by the condition $E \sqsubseteq E\{i{+}1/i\}$), as it is essential for subtyping (see Lemma 2.1.7). Note that in this definition, D is not necessarily monotonous, but it must be a subtype of a monotonous type E. This gives us more freedom on the type D than directly asking for monotonicity. Finally, linearity in this rule is expressed by the impossibility to use higher-order variables in the iterated function, contrary to other notions of linearity for system T [5].

Examples in sℓT

For the sake of conciseness, we may write from now on $\lambda x, y, z.t$ instead of $\lambda x.\lambda y.\lambda z.t$.

Other iterators Using a function reversing a word `rev`, that one could construct easily, we can define an iterator on words doing operations in the usual order. We denote this iterator with `Riterw`. Formally it is defined by:

$$\texttt{Riterw}(V_0, V_1, t) := \lambda w.\texttt{iterw}(V_0, V_1, t)\ (\texttt{rev}\ w).$$

We have $\texttt{Riterw}(V_0, V_1, W)\ \underline{w_0 w_1 \dots w_n} \to^* V_{w_0}\ (V_{w_1}\ (\cdots (V_{w_n}\ W) \cdots))$.

We also show that for integers we can construct an iterator $\texttt{rec}(V, t)$ with:

$$\texttt{rec}(V, t)\ \underline{n} \to^* V\ \underline{n{-}1}\ (V\ \underline{n{-}2}\ (\dots\ (V\ \underline{0}\ t)\dots))$$

and such that the following rule is derivable.

$$\frac{\begin{array}{ccc} D \sqsubseteq E & E\{I/i\} \sqsubseteq F & E \sqsubseteq E\{i{+}1/i\} \\ d\Gamma \vdash V : D \multimap \mathsf{N}^i \multimap D\{i{+}1/i\} & & \Gamma, d\Gamma \vdash t : D\{1/i\} \end{array}}{\Gamma, d\Gamma \vdash \texttt{rec}(V, t) : \mathsf{N}^I \multimap F}$$

$$\pi\lhd \frac{D \sqsubseteq E}{\Gamma, x:D \vdash x:E} \qquad \omega(\pi) = 1$$

$$\pi\lhd \frac{\sigma \lhd \Gamma, x:D \vdash t:E}{\Gamma \vdash \lambda x.t : D \multimap E} \qquad \omega(\pi) = 1{+}\omega(\sigma)$$

$$\pi\lhd \frac{\sigma \lhd \Gamma, d\Gamma \vdash t : E \multimap D \qquad \tau \lhd \Gamma', d\Gamma \vdash u : E}{\Gamma, \Gamma', d\Gamma \vdash t\ u : D} \qquad \omega(\pi) = \omega(\sigma){+}\omega(\tau)$$

$$\pi\lhd \frac{\sigma \lhd \Gamma, d\Gamma \vdash t : D \qquad \tau \lhd \Gamma', d\Gamma \vdash u : E}{\Gamma, \Gamma', d\Gamma \vdash t \otimes u : D \otimes E} \qquad \omega(\pi) = \omega(\sigma){+}\omega(\tau){+}1$$

$$\pi\lhd \frac{\sigma \lhd \Gamma, d\Gamma, x:D, y:E \vdash u : F \qquad \tau \lhd \Gamma', d\Gamma \vdash t : D \otimes E}{\Gamma, \Gamma', d\Gamma \vdash \mathtt{let}\ x \otimes y = t\ \mathtt{in}\ u : F} \qquad \omega(\pi) = \omega(\sigma){+}\omega(\tau)$$

$$\pi\lhd \frac{}{\Gamma \vdash \underline{0} : \mathsf{N}^I} \qquad \omega(\pi) = 0$$

$$\pi\lhd \frac{J{+}1 \leq I \qquad \sigma \lhd \Gamma \vdash t : \mathsf{N}^J}{\Gamma \vdash \mathtt{s}(t) : \mathsf{N}^I} \qquad \omega(\pi) = \omega(\sigma)$$

$$\pi\lhd \frac{\sigma \lhd \Gamma, d\Gamma \vdash t : \mathsf{N}^I \multimap D \qquad \tau \lhd \Gamma', d\Gamma \vdash u : D}{\Gamma, \Gamma', d\Gamma \vdash \mathtt{ifn}(t,u) : \mathsf{N}^I \multimap D} \qquad \omega(\pi) = \omega(\sigma){+}\omega(\tau){+}1$$

$$\pi\lhd \frac{\begin{array}{c} D \sqsubseteq E \qquad E\{I/i\} \sqsubseteq F \qquad E \sqsubseteq E\{i{+}1/i\} \\ \sigma \lhd d\Gamma \vdash V : D \multimap D\{i{+}1/i\} \qquad \tau \lhd \Gamma, d\Gamma \vdash t : D\{1/i\} \end{array}}{\Gamma, d\Gamma \vdash \mathtt{itern}(V,t) : \mathsf{N}^I \multimap F} \qquad \omega(\pi) = \omega(\tau){+}I \cdot (\omega(\sigma){+}1)\{I/i\}$$

$$\pi\lhd \frac{}{\Gamma \vdash \epsilon : \mathsf{W}^I} \qquad \omega(\pi) = 0$$

$$\pi\lhd \frac{J{+}1 \leq I \qquad \sigma \lhd \Gamma \vdash t : \mathsf{W}^J}{\Gamma \vdash \mathtt{s}_i(t) : \mathsf{W}^I} \qquad \omega(\pi) = \omega(\sigma)$$

$$\pi\lhd \frac{\forall i, \sigma_i \lhd \Gamma_i, d\Gamma \vdash t_i : \mathsf{W}^I \multimap D \qquad \tau \lhd \Gamma', d\Gamma \vdash u : D}{\Gamma_1, \Gamma_2, \Gamma', d\Gamma \vdash \mathtt{ifw}(t_1, t_2, u) : \mathsf{W}^I \multimap D} \qquad \omega(\pi) = \omega(\sigma_1){+}\omega(\sigma_2){+}\omega(\tau){+}1$$

$$\pi\lhd \frac{\begin{array}{c} D \sqsubseteq E \qquad E\{I/i\} \sqsubseteq F \qquad E \sqsubseteq E\{i{+}1/i\} \\ \forall i, \sigma_i \lhd d\Gamma \vdash V_i : D \multimap D\{i{+}1/i\} \qquad \tau \lhd \Gamma, d\Gamma \vdash t : D\{1/i\} \end{array}}{\Gamma, d\Gamma \vdash \mathtt{iterw}(V_0, V_1, t) : \mathsf{W}^I \multimap F} \qquad \omega(\pi) = \omega(\tau){+}I \cdot (\omega(\sigma_1){+}\omega(\sigma_2){+}1)\{I/i\}$$

$$\pi\lhd \frac{}{\Gamma \vdash \mathtt{tt} : \mathsf{B}} \qquad \omega(\pi) = 0$$

$$\pi\lhd \frac{}{\Gamma \vdash \mathtt{ff} : \mathsf{B}} \qquad \omega(\pi) = 0$$

$$\pi\lhd \frac{\sigma \lhd \Gamma, d\Gamma \vdash t : D \qquad \tau \lhd \Gamma', d\Gamma \vdash u : D}{\Gamma, \Gamma', d\Gamma \vdash \mathtt{if}(t,u) : \mathsf{B} \multimap D} \qquad \omega(\pi) = \omega(\sigma){+}\omega(\tau){+}1$$

Figure 2.3: Type system for $\mathsf{s}\ell\mathsf{T}$.

We first give a term that takes a pair with an object x of type D and an integer n and returns the pair $(V\ n\ x, n{+}1)$:

$$t_{step} := \lambda r.\mathtt{let}\ x \otimes n = r\ \mathtt{in}\ (V\ n\ x) \otimes \mathtt{s}(n).$$

We have $d\Gamma \vdash t_{step} : (D \otimes \mathsf{N}^i) \multimap (D\{i{+}1/i\} \otimes \mathsf{N}^{i+1})$. Thus, we can iterate on this term using `itern` and this gives us the desired iterator:

$$\mathtt{rec}(V,t) := \lambda n.\mathtt{let}\ x \otimes m = (\mathtt{itern}(t_{step}, t \otimes \underline{0})\ n)\ \mathtt{in}\ x.$$

This constructor can be defined likewise for words.

Addition for unary integers In order to give an example of weight, we define the addition for unary integers in $\mathsf{s}\ell\mathsf{T}$. It is represented by:

$$\mathtt{add} := \lambda x.\mathtt{itern}(\lambda y.\mathtt{s}(y), x) : \mathsf{N}^I \multimap \mathsf{N}^J \multimap \mathsf{N}^{I+J}.$$

The typing derivation is:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\mathsf{N}^{I+i} \sqsubseteq \mathsf{N}^{I+i}}{I{+}i{+}1 \leq I{+}i{+}1 \qquad x : \mathsf{N}^I, y : \mathsf{N}^{I+i} \vdash y : \mathsf{N}^{I+i}}}{x : \mathsf{N}^I, y : \mathsf{N}^{I+i} \vdash \mathtt{s}(y) : \mathsf{N}^{I+i+1}}}{\pi_1 \lhd x : \mathsf{N}^I \vdash \lambda y.\mathtt{s}(y) : \mathsf{N}^{I+i} \multimap \mathsf{N}^{I+i+1}} \qquad \dfrac{\mathsf{N}^I \sqsubseteq \mathsf{N}^{I+1}}{\pi_2 \lhd x : \mathsf{N}^I \vdash x : \mathsf{N}^{I+1}}}{x : \mathsf{N}^I \vdash \mathtt{itern}(\lambda y.\mathtt{s}(y), x) : \mathsf{N}^J \multimap \mathsf{N}^{I+J}}}{\pi_0 \lhd \cdot \vdash \mathtt{add} : \mathsf{N}^I \multimap \mathsf{N}^J \multimap \mathsf{N}^{I+J}}$$

(1)

where (1) is $\mathsf{N}^{I+i} \sqsubseteq \mathsf{N}^{I+i}$, $\mathsf{N}^{I+i} \sqsubseteq \mathsf{N}^{I+J}$, $\mathsf{N}^{I+i} \sqsubseteq \mathsf{N}^{I+i+1}$, that is the conditions imposed by the iteration rule. We have $\omega(\pi_1) = 2$ and $\omega(\pi_2) = 1$. Thus, the final weight is $\omega(\pi_0) = 1{+}(1{+}J \cdot (2{+}1)) = 3J{+}2$.

Multiplication for unary integers We also sketch the multiplication in $\mathsf{s}\ell\mathsf{T}$. The multiplication can be represented by:

$$\mathtt{mult} := \lambda x.\mathtt{itern}(\lambda y.\mathtt{add}\ x\ y, \underline{0}) : \mathsf{N}^I \multimap \mathsf{N}^J \multimap \mathsf{N}^{I \cdot J}$$

and the typing derivation is:

$$\dfrac{\dfrac{\dfrac{x : \mathsf{N}^I, y : \mathsf{N}^{I \cdot a} \vdash \mathtt{add}\ x\ y : \mathsf{N}^{I \cdot a + I}}{\pi_1 \lhd x : \mathsf{N}^I \vdash \lambda y.\mathtt{add}\ x\ y : \mathsf{N}^{I \cdot a} \multimap \mathsf{N}^{I \cdot a}\{a{+}1/a\}} \qquad \dfrac{}{\pi_2 \lhd x : \mathsf{N}^I \vdash \underline{0} : \mathsf{N}^I}}{x : \mathsf{N}^I \vdash \mathtt{itern}(\lambda y.\mathtt{add}\ x\ y, \underline{0}) : \mathsf{N}^J \multimap \mathsf{N}^{I \cdot J}}}{\pi_0 \lhd \cdot \vdash \mathtt{mult} : \mathsf{N}^I \multimap \mathsf{N}^J \multimap \mathsf{N}^{I \cdot J}}$$

With the previous weight for `add`, we obtain $\omega(\pi_1) = 3 \cdot I \cdot a{+}5$. We also have $\omega(\pi_2) = 1$. Thus, we can compute the final weight:

$$\omega(\pi_0) = 1{+}\omega(\pi_2){+}J \cdot (\omega(\pi_1){+}1)\{J/a\} = 3IJ^2{+}6J{+}2.$$

Note that this way of doing multiplication is not optimal as we iterate on the largest number during the addition. However, this is a good example for the weight, so we presented this version of multiplication instead of a better one.

Addition on binary integers Now, we define some terms working on integers written in binary, with type W^I, as we will use them later in order to describe our term for SAT (see Section 2.3). First, we can define an addition on binary integers in $\mathsf{s}\ell\mathsf{T}$ with a control on the number of bits. More precisely, we can give a term $\mathtt{Cadd} : \mathsf{N}^I \multimap \mathsf{W}^{J_1} \multimap \mathsf{W}^{J_2} \multimap \mathsf{W}^I$ such that $\mathtt{Cadd}\ n\ w_1\ w_2$ outputs the least significant n bits of the sum $w_1{+}w_2$. For example, $\mathtt{Cadd}\ 3\ 101\ 110 = 011$, and $\mathtt{Cadd}\ 5\ 101\ 110 = 01011$. This will usually be used with a n greater than the expected number of bits, the idea being that those extra 0 can be useful for some other programs. The term follows the usual algorithm for addition: the result is computed bit by bit starting from the right, and we keep track of the carry.

Unary integers to binary integers We define a term $\mathtt{CUnToBi} : \mathsf{N}^I \multimap \mathsf{N}^J \multimap \mathsf{W}^I$ such that on the input n, n', this term computes the least n significant bits of the representation of n' in binary:

$$\mathtt{CUnToBi} = \lambda n.\mathtt{itern}(\lambda w.\mathtt{Cadd}\ n\ w\ (\mathsf{s}_1(\epsilon)), \mathtt{Cadd}\ n\ \epsilon\ \epsilon).$$

Binary integers to unary integers We would like a way to compute the unary integer corresponding to a given binary integer. However, this function is exponential in the size of its input, so it should have intuitively the type $\mathsf{W}^J \multimap \mathsf{N}^{2^J}$. As we cannot do exponentiation on indices, it is impossible to write such a function in $\mathsf{s}\ell\mathsf{T}$. Nevertheless, given an additional information bounding the size of this unary word, we can give a term $\mathtt{CBiToUn} : \mathsf{N}^I \multimap \mathsf{W}^J \multimap \mathsf{N}^I$ such that on an input n, w this term computes the minimum between n and the unary representation of w. What is important in this type is the discarding of J. In order to do that, we first define a term $\mathtt{min} : \mathsf{N}^I \multimap \mathsf{N}^J \multimap \mathsf{N}^I$. As N^I is the type of integers of size smaller than I, we have indeed that the minimum between an integer of size smaller than I and an integer of size smaller than J is smaller than I. It is not a precise bound but it is sufficient for its incoming use. The idea to construct $\mathtt{min}$ is to use the following term:

$$t_{step} := \lambda r.\mathtt{let}\ n \otimes m = r\ \mathtt{in}\ \mathtt{ifn}(\lambda m'.\mathsf{s}(n) \otimes m', n \otimes \underline{0})\ m.$$

The term t_{step} takes a pair of integers (n, m) and if $m = 0$, it does nothing, otherwise it returns $(n{+}1, m{-}1)$. We can derive the typing $\cdot \vdash t_{step} : (\mathsf{N}^i \otimes \mathsf{N}^J) \multimap (\mathsf{N}^{i+1} \otimes \mathsf{N}^J)$. Thus, we can iterate on this term, and if we iterate n times starting from the pair $(0, n')$ we indeed compute the minimum between n and n':

$$\mathtt{min} = \lambda n, n'.\mathtt{let}\ m \otimes m' = (\mathtt{itern}(t_{step}, \underline{0} \otimes n')\ n)\ \mathtt{in}\ m.$$

Now that we have the term $\mathtt{min}$, we can define the following term:

$$\mathtt{CBiToUn} = \lambda n.\mathtt{iterw}(\lambda n'.\mathtt{min}\ n\ (\mathtt{mult}\ n'\ \underline{2}), \lambda n'.\mathtt{min}\ n\ (\mathsf{s}(\mathtt{mult}\ n'\ \underline{2})), \underline{0}).$$

2.1.2 Subject Reduction and Soundness

Index Variable Substitution and Subtyping

In order to prove the subject reduction for $\mathsf{s}\ell\mathsf{T}$ and that the weight is a bound on the number of reduction steps of a term, we give some intermediate lemmas.

First, we show that typed values are linked to normal forms. In particular, this theorem shows that a value of type N is indeed of the form $\mathsf{s}(\mathsf{s}(\dots(\mathsf{s}(\underline{0}))\dots))$. From this it follows that in this call-by-value calculus, when an argument is of type N, it is the encoding of an integer.

Lemma 2.1.1. *Let t be a term in $\mathsf{s}\ell\mathsf{T}$, if t is closed and has a typing derivation $\vdash t : D$ then t is in normal form if and only if t is a value V.*

Proof. First, we prove by induction on values V that if V is closed and has a typing derivation then V is in normal form. We only show some cases and the others are easily deducible from those cases.

- If $V = \lambda x.t$ then V is in normal form since in the definition of contexts for reductions, we cannot reduce under a λ-abstraction.
- If $V = V_0 \otimes V_1$. V is closed so are V_0 and V_1. Moreover, V has a typing derivation, so it must end with the introduction of tensor rule, and we deduce that V_0 and V_1 have also a typing derivation. So by induction hypothesis, V_0 and V_1 are in normal form. Then V has no base reduction possible, and no contexts reductions since V_0 and V_1 are is normal form, so V is also in normal form.
- If $V = \underline{0}$ then V is in normal form.

Now for the other implication, we prove that if a closed typed term is in normal form then it is a value. We prove that by induction on terms, again we only detail some interesting cases.

- If $t := t_0\ t_1$. Suppose, by absurd, that t is a closed typed normal term. Since t has a typing derivation, we know that t_0 and t_1 are also closed typed terms. By definition of contexts in which we can apply reductions, t_0 is normal, and so by induction hypothesis, t_0 is a value. Again, by definition of contexts, t_1 is normal, and so by induction hypothesis, t_1 is a value. So t_0 is a value with an arrow type $D \multimap E$. By looking at the definition of values, either t_0 is a λ-abstraction, or it is one of the functional constructors like $\texttt{ifn}$. If t_0 is a λ-abstraction, as t_1 is a value, we could apply the usual β-rule, so this is not possible because t is in normal form. If t_0 is $\texttt{ifn}(V, V')$, as t_1 is a value of type N, it is the encoding of an integer, and so t is not normal since we could apply one of the $\texttt{ifn}$ rules. All the other cases work in the same way, and we deduce that t cannot be in normal form.
- If $t := \texttt{let}\ x \otimes y = t_0\ \texttt{in}\ t_1$. Suppose that t is a closed typed normal term. Since t has a typing derivation, we know that t_0 has also a typing derivation, and t_0 is closed. By definition of contexts, t_0 is in normal form and so by induction hypothesis, t_0 is a value. t_0 has a tensor type $D \otimes E$, by definition of values, t_0 is of the form $V \otimes W$, this is absurd since in this case t would not be normal. And so, we deduce that t cannot be a normal term.

□

We now give a list of intermediate lemmas for which we do not always detail the proofs if they are immediate.

Lemma 2.1.2 (Weakening)**.** *Let Δ, Γ be disjoint typing contexts, and $\pi \lhd \Gamma \vdash t : D$. Then, we have a proof $\pi' \lhd \Gamma, \Delta \vdash t : D$ with $\omega(\pi) = \omega(\pi')$.*

Lemma 2.1.3 (Index substitution)**.** *Let I be an index.*

1. *Let J_1, J_2 be indices such that $J_1 \leq J_2$ then $J_1\{I/i\} \leq J_2\{I/i\}$.*
2. *Let J_1, J_2 be indices such that $J_1 < J_2$ then $J_1\{I/i\} < J_2\{I/i\}$.*
3. *Let D, D' be types such that $D \sqsubseteq D'$ then $D\{I/i\} \sqsubseteq D'\{I/i\}$.*
4. *If $\pi \lhd \Gamma \vdash t : D$ then $\pi\{I/i\} \lhd \Gamma\{I/i\} \vdash t : D\{I/i\}$.*

5. $\omega(\pi\{I/i\}) = \omega(\pi)\{I/i\}$.

Proof. Point 1 and Point 2 are by definition of $\leq$ and $<$, then Point 3 is a direct induction on types using Point 1 for base types. Point 4 and Point 5 are proved by induction on π:

- In the case of $\mathtt{s}$ or $\mathtt{s}_i$, we use Point 1.
- In the axiom rule, we use Point 3.
- Then, the only interesting cases are iterations. We show here the iteration for integers. Suppose that we have the following proof:

$$\pi\lhd \frac{\begin{array}{ccc} D \sqsubseteq E & E \sqsubseteq E\{j{+}1/j\} & E\{J/j\} \sqsubseteq F \\ \sigma \lhd d\Gamma \vdash V : D \multimap D\{j{+}1/j\} & \tau \lhd \Gamma, d\Gamma \vdash t : D\{1/j\} & \end{array}}{\Gamma, d\Gamma \vdash \mathtt{itern}(V,t) : \mathsf{N}^J \multimap F}$$

  With $\omega(\pi) = \omega(\tau){+}J \cdot (\omega(\sigma){+}1)\{J/j\}$. We want to prove that $\pi\{I/i\} \lhd \Gamma\{I/i\} \vdash \mathtt{itern}(V,t) : \mathsf{N}^{J\{I/i\}} \multimap F\{I/i\}$.

  By induction hypothesis and Point 3 of Lemma 2.1.3 we have

$$\begin{array}{ccc} D\{I/i\} \sqsubseteq E\{I/i\} & E\{I/i\} \sqsubseteq E\{j{+}1/j\}\{I/i\} & E\{J/j\}\{I/i\} \sqsubseteq F\{I/i\} \\ \sigma\{I/i\} \lhd d\Gamma\{I/i\} \vdash V : D\{I/i\} \multimap D\{j{+}1/j\}\{I/i\} & & \tau\{I/i\} \lhd \Gamma\{I/i\}, d\Gamma\{I/i\} \vdash t : D\{1/j\}\{I/i\} \end{array}$$

  By using the fact that j must be a fresh variable in Γ, $d\Gamma$, J and F, we can suppose, by renaming, that j does not occur in I. Then, we obtain:

$$\pi\{I/i\}\lhd \frac{\begin{array}{ccc} D\{I/i\} \sqsubseteq E\{I/i\} & E\{I/i\} \sqsubseteq E\{I/i\}\{j{+}1/j\} & E\{I/i\}\{J\{I/i\}/b\} \sqsubseteq F\{I/i\} \\ \sigma\{I/i\} \lhd d\Gamma\{I/i\} \vdash V : D\{I/i\} \multimap D\{I/i\}\{j{+}1/j\} & & \tau\{I/i\} \lhd \Gamma\{I/i\}, d\Gamma\{I/i\} \vdash t : D\{I/i\}\{1/j\} \end{array}}{\Gamma\{I/i\}, d\Gamma\{I/i\} \vdash \mathtt{itern}(V,t) : \mathsf{N}^{J\{I/i\}} \multimap F\{I/i\}}$$

  with weight $\omega(\pi\{I/i\}) = J\{I/i\}{+}\omega(\tau)\{I/i\}{+}J\{I/i\} \cdot \omega(\sigma)\{I/i\}\{J\{I/i\}/b\}$.

  And so $\omega(\pi\{I/i\}) = \omega(\pi)\{I/i\}$.

□

Lemma 2.1.4 (Monotonic index substitution)**.** *Take J_1, J_2 such that $J_1 \leq J_2$.*

1. *Let I be an index, then $I\{J_1/i\} \leq I\{J_2/i\}$.*
2. *For any proof π, $\omega(\pi\{J_1/i\}) \leq \omega(\pi\{J_2/i\})$.*
3. *Let E be a type. If $E \sqsubseteq E\{i{+}1/i\}$ then $E\{J_1/i\} \sqsubseteq E\{J_2/i\}$ and if $E\{i{+}1/i\} \sqsubseteq E$ then $E\{J_2/i\} \sqsubseteq E\{J_1/i\}$.*

Proof. Point 1 can be proved by induction on indices, and then Point 2 is just a particular case of Point 1, by Lemma 2.1.3. Point 3 is proved by induction on the type E. Let us first show an intermediate lemma:

Lemma 2.1.5. *Let I be an index. If an index variables i has at least one occurrence in I then we have $I < I\{i{+}1/i\}$.*

By induction on I. On the base case, $I = i$ so obviously $i < i{+}1$. In the inductive case $I = J_1{+}J_2$, if i has at least one occurrence in I, then i has at least one occurrence in J_1 or in J_2. If i appears in J_1 and J_2, by induction hypothesis we have $J_1 < J_1\{i{+}1/i\}$ and $J_2 < J_2\{i{+}1/i\}$. So, we obtain directly $I < I\{i{+}1/i\}$. If i appears in J_1 and not in J_2, then we have $J_1 < J_1\{i{+}1/i\}$ and $J_2 = J_2\{i{+}1/i\}$. Thus, we obtain $I < I\{i{+}1/i\}$. The last case is symmetric. The case of the multiplication is similar to the one for addition, using the fact that all indices are at least 1 so multiplication conserves the strict ordering.

Now we can prove Point 3 of Lemma 2.1.4 by induction on E.

- Suppose that E is a base type. The boolean case is direct, so we suppose that $E = \mathsf{N}^I$ (the case for words is similar). By Point 1 of Lemma 2.1.4, we have $I\{J_1/i\} \leq I\{J_2/i\}$. This concludes the first case. Now, suppose that $E\{i{+}1/i\} \sqsubseteq E$. By definition, this means $I\{i{+}1/i\} \leq I$. By Lemma 2.1.5, i have no occurrence in I. Thus, we have directly $I\{J_2/i\} \leq I\{J_1/i\}$, and so $E\{J_2/i\} \sqsubseteq E\{J_1/i\}$.
- If $E = D \multimap D'$. Suppose that $E \sqsubseteq E\{i{+}1/i\}$. By definition, $D' \sqsubseteq D'\{i{+}1/i\}$ and $D\{i{+}1/i\} \sqsubseteq D$. By induction hypothesis, we have $D'\{J_1/i\} \sqsubseteq D'\{J_2/i\}$ and $D\{J_2/i\} \sqsubseteq D\{J_1/i\}$. Thus, we obtain $E\{J_1/i\} \sqsubseteq E\{J_2/i\}$. The other case $E\{i{+}1/i\} \sqsubseteq E$ is similar.
- The case for $E = D \otimes D'$ is direct by induction hypothesis.

□

Lemma 2.1.6 (Typing Base Values)**.** *If $\pi \lhd \Gamma, d\Gamma \vdash V : U$ then we have a proof $\pi' \lhd d\Gamma \vdash V : U$ with $\omega(\pi) = \omega(\pi')$. Moreover, $\omega(\pi') \leq 1$.*

Indeed, the only rules we can use to type values of base type are the axiom rule with a variable in $d\Gamma$ or the rules for base constructors on integers, words or boolean such as $\underline{0}$ or $\mathtt{s}$.

Another important lemma is the one for subtyping, it shows that we do not need an explicit rule for subtyping and subtyping does not harm the upper bound derived from typing. Moreover, this lemma is important in order to substitute variables, since the axiom rule allows subtyping.

Lemma 2.1.7 (Subtyping)**.** *If $\pi \lhd \Gamma \vdash t : D$ then for all Γ', D' such that $D \sqsubseteq D'$ and $\Gamma' \sqsubseteq \Gamma$, we have a proof $\pi' \lhd \Gamma' \vdash t : D'$ with $\omega(\pi') \leq \omega(\pi)$.*

Proof. This can be proved by induction on π. The only interesting cases are for iterations. We only detail the case of iteration for integers. Suppose that we have

$$\pi\lhd\ \frac{\begin{array}{ccc} D \sqsubseteq E & E\{I/i\} \sqsubseteq F & E \sqsubseteq E\{i/i{+}1\} \\ \sigma \lhd d\Gamma \vdash V : D \multimap D\{i{+}1/i\} & & \tau \lhd \Gamma, d\Gamma \vdash t : D\{1/i\} \end{array}}{\Gamma, d\Gamma \vdash \mathtt{itern}(V, t) : \mathsf{N}^I \multimap F}$$

with $\omega(\pi) = \omega(\tau){+}I \cdot (\omega(\sigma){+}1)\{I/i\}$. Let $\Gamma', d\Gamma', I', F'$ be such that $\mathsf{N}^I \multimap F \sqsubseteq \mathsf{N}^{I'} \multimap F'$ and $\Gamma', d\Gamma' \sqsubseteq \Gamma, d\Gamma$. By definition, we have $F \sqsubseteq F'$ and $I' \leq I$. By induction hypothesis we have $\sigma' \lhd d\Gamma' \vdash V : D \multimap D\{i{+}1/i\}$ and $\tau' \lhd \Gamma', d\Gamma' \vdash t : D\{1/i\}$ with $\omega(\sigma') \leq \omega(\sigma)$ and $\omega(\tau') \leq \omega(\tau)$. We give then the following proof π':

$$\pi'\lhd\ \frac{\begin{array}{ccc} D \sqsubseteq E & E\{I'/i\} \sqsubseteq F' & E \sqsubseteq E\{i/i{+}1\} \\ \sigma' \lhd d\Gamma' \vdash V : D \multimap D\{i{+}1/i\} & & \tau' \lhd \Gamma', d\Gamma' \vdash t : D\{1/i\} \end{array}}{\Gamma, d\Gamma \vdash \mathtt{itern}(V, t) : \mathsf{N}^{I'} \multimap F'}$$

with $\omega(\pi') = \omega(\tau')+I' \cdot (\omega(\sigma')+1)\{I'/i\}$. Indeed, by Point 3 of Lemma 2.1.4, we have $E\{I'/i\} \sqsubseteq E\{I/i\}$. Thus, by transitivity, we have $E\{I'/i\} \sqsubseteq E\{I/i\} \sqsubseteq F \sqsubseteq F'$. Moreover, $\omega(\pi') \leq \omega(\pi)$ since $\omega(\tau') \leq \omega(\tau)$, $I' \leq I$, and:

$$(\omega(\sigma')+1)\{I'/i\} \leq (\omega(\sigma')+1)\{I/i\} \leq (\omega(\sigma)+1)\{I/i\}$$

by Point 1 of Lemma 2.1.4 for the first inequality, and since $\omega(\sigma') \leq \omega(\sigma)$, we get the second inequality by Point 1 of Lemma 2.1.3. □

Term Substitution Lemma

In order to prove the subject reduction of the calculus, we examine what happens during a substitution of a value in a term. There are two cases, first the substitution of variables with base types, that is to say duplicable variables, and then the substitution of variables with a non-base type for which the type system imposes linearity.

Lemma 2.1.8 (Value Substitution)**.** *Suppose that $\pi \lhd \Gamma_1, d\Gamma, x : E \vdash t : D$ and $\sigma \lhd \Gamma_2, d\Gamma \vdash V : E$, then we have a proof $\pi' \lhd \Gamma_1, \Gamma_2, d\Gamma \vdash t\{V/x\} : D$. Moreover, if E is a base type then $\omega(\pi') \leq \omega(\pi)$. Otherwise, $\omega(\pi') \leq \omega(\pi)+\omega(\sigma)$.*

Proof. This is proved by induction on π. For the base type case, we use Lemma 2.1.6 to show that Γ_2 can be ignored, and then as $d\Gamma$ is duplicable, the proof is rather direct. For the non-base case, in multiplicative rules such as application and `if`, the property holds by the fact that x only appears in one of the premises, and so $\omega(\sigma)$ appears only once in the total weight. □

Subject Reduction and Upper Bound

We can now express the subject-reduction of the calculus and the fact that the weight of a proof strictly decreases during a reduction.

Theorem 2.1.1. *Suppose that $\tau \lhd \Gamma \vdash t_0 : D$ and $t_0 \to t_1$, then there is a proof $\tau' \lhd \Gamma \vdash t_1 : D$ such that $\omega(\tau') < \omega(\tau)$.*

Proof. By induction. We first consider the base-reduction case. Some cases are trivial and we will not develop them. Indeed the if-rules can be proved with weakening (Lemma 2.1.2).

- If $t_0 = (\lambda x.t)V$, and $t_1 = t\{V/x\}$, we have a proof:

$$\tau \lhd \dfrac{\dfrac{\pi \lhd \Gamma_1, d\Gamma, x : E \vdash t : D}{\Gamma_1, d\Gamma \vdash \lambda x.t : E \multimap D} \qquad \sigma \lhd \Gamma_2, d\Gamma \vdash V : E}{\Gamma_1, \Gamma_2, d\Gamma \vdash (\lambda x.t)V : D}$$

  with $\omega(\tau) = \omega(\sigma)+1+\omega(\pi)$.

  Then by using the value substitution lemma (Lemma 2.1.8) with π and σ, we obtain a proof $\pi' \lhd \Gamma_1, \Gamma_2, d\Gamma \vdash t\{V/x\} : D$. Moreover, we have $\omega(\pi') \leq \omega(\pi)+\omega(\sigma) < \omega(\tau)$. This concludes this case.

- If $t_0 = \texttt{let}\ x \otimes y = V_0 \otimes V_1\ \texttt{in}\ t$ and $t_1 = t\{V_0/x\}\{V_1/y\}$, we can conclude this case by using twice Lemma 2.1.8.

- If $t_0 = \texttt{itern}(V, V')\ \underline{0}$ and $t_1 = V'$. We have a proof:

$$\dfrac{\dfrac{\sigma_1\{1/i\} \lhd d\Gamma \vdash V : D\{1/i\} \multimap D\{i{+}1/i\}\{1/i\} \qquad \sigma_2 \lhd \Gamma, d\Gamma \vdash V' : D\{1/i\}}{\sigma_1\{i{+}1/i\} \lhd d\Gamma \vdash V : D\{i{+}1/i\} \multimap D\{i{+}1/i\}\{i{+}1/i\} \qquad \Gamma, d\Gamma \vdash V\ V' : D\{i{+}1/i\}\{1/i\}}\ (1)}{\Gamma, d\Gamma \vdash \mathtt{itern}(V, V\ V') : \mathsf{N}^J \multimap F}$$

where (1) is:

$$D\{a{+}1/a\} \sqsubseteq E\{a{+}1/a\} \qquad E\{a{+}1/a\} \sqsubseteq E\{a{+}1/a\}\{a{+}1/a\} \qquad E\{a{+}1/a\}\{J/a\} = E\{J{+}1/a\} \sqsubseteq E\{I/a\} \sqsubseteq F$$

Figure 2.4: A derivation for $\mathtt{itern}(V, V\ V')$.

$$\dfrac{\dfrac{\dfrac{D \sqsubseteq E \qquad E \sqsubseteq E\{i{+}1/i\} \qquad E\{I/i\} \sqsubseteq F}{\sigma_1 \lhd d\Gamma \vdash V : D \multimap D\{i{+}1/i\} \qquad \sigma_2 \lhd \Gamma, d\Gamma \vdash V' : D\{1/i\}} \qquad \dfrac{}{\Gamma', d\Gamma \vdash \underline{0} : \mathsf{N}^I}}{\tau\lhd\ \Gamma, \Gamma', d\Gamma \vdash \mathtt{itern}(V, V')\ \underline{0} : F}$$

with $\omega(\tau) = I{+}\omega(\sigma_2){+}I \cdot \omega(\sigma_1)\{I/a\} \geq 1{+}\omega(\sigma_2)$.

We have $D\{1/i\} \sqsubseteq E\{1/i\} \sqsubseteq E\{I/i\} \sqsubseteq F$ by Lemma 2.1.3 and Lemma 2.1.4 since $1 \leq I$. So, by subtyping and weakening (Lemma 2.1.7 and Lemma 2.1.2), we have a proof $\sigma_2' \lhd \Gamma, \Gamma', d\Gamma \vdash V' : F$ with $\omega(\sigma_2') \leq \omega(\sigma_2) < \omega(\tau)$.

This concludes this case. The proof for the rule `iterw` with ϵ follows the same pattern.

- If $t_0 = \mathtt{itern}(V, V')\ \mathtt{s}(W)$ and $t_1 = \mathtt{itern}(V, V\ V')\ W$. We have a proof:

$$\dfrac{\dfrac{\dfrac{D \sqsubseteq E \qquad E \sqsubseteq E\{i{+}1/i\} \qquad E\{I/i\} \sqsubseteq F}{\sigma_1 \lhd d\Gamma \vdash V : D \multimap D\{i{+}1/i\} \qquad \sigma_2 \lhd \Gamma, d\Gamma \vdash V' : D\{1/i\}} \qquad \dfrac{\pi \lhd \Gamma', d\Gamma \vdash W : \mathsf{N}^J \qquad J{+}1 \leq I}{\Gamma', d\Gamma \vdash \mathtt{s}(W) : \mathsf{N}^I}}{\tau\lhd\ \Gamma, \Gamma', d\Gamma \vdash \mathtt{itern}(V, V')\ \mathtt{s}(W) : F}$$

with $\omega(\tau) = \omega(\pi){+}I{+}\omega(\sigma_2){+}I \cdot \omega(\sigma_1)\{I/i\}$.

We can construct a proof τ_0' for $\mathtt{itern}(V, V\ V')$. The proof is described in Figure 2.4. This gives us a proof for t_1.

$$\tau'\lhd\ \dfrac{\tau_0' \lhd \Gamma, d\Gamma \vdash \mathtt{itern}(V, V\ V') : \mathsf{N}^J \multimap F \qquad \pi \lhd \Gamma', d\Gamma \vdash W : \mathsf{N}^J}{\Gamma, \Gamma', d\Gamma \vdash \mathtt{itern}(V, V\ V')W : F}$$

with $\omega(\tau') = \omega(\pi){+}J{+}\omega(\sigma_2){+}\omega(\sigma_1)\{1/i\}{+}J \cdot \omega(\sigma_1)\{i{+}1/i\}\{J/i\}$.

And we have $\omega(\tau') \leq \omega(\pi){+}J{+}\omega(\sigma_2){+}(J{+}1) \cdot \omega(\sigma_1)\{J{+}1/i\}$ so, since $J{+}1 \leq I$, we have $\omega(\tau') < \omega(\pi){+}I{+}\omega(\sigma_2){+}I \cdot \omega(\sigma_1)\{I/i\} = \omega(\tau)$.

The rules for `iterw` in the cases $\mathtt{s}_0$ and $\mathtt{s}_1$ follow the same pattern.

Now we need to verify that a reduction under context strictly decreases the weight. This can be proved directly by structural induction on contexts. □

As the indices can only define polynomials, the weight of a sequent can only be a polynomial on the index variables. And so, in $\mathsf{s}\ell\mathsf{T}$, we can only define terms that work in time polynomial in the size of their inputs.

Polynomial indices and Degree

For the following section on the elementary affine logic, we need to define a notion of degree of indices and make clear some properties of this notion.

Definition 2.1.6. *The indices can be seen as multi-variables polynomials, and we can define the* degree *of an index I by induction on I.*

- $\forall n \in \mathbb{N}^*, d(n) = 0$.
- *For an index variable i, $d(i) = 1$.*
- $d(I{+}J) = max(d(I), d(J))$.
- $d(I \cdot J) = d(I){+}d(J)$.

This definition of degree is essential for the control of reductions in sEAL, that we present in the following section. We obtain the following property for degree.

Theorem 2.1.2 (Degree)**.** *For all indices I and J, the following properties are verified:*

1. *For all non-zero integer k, we have $I\{k/i\} \leq k^{d(I)} \cdot I\{1/i\}$.*
2. *If $I \leq J$ then $d(I) \leq d(J)$.*

Proof. The first point is proved by induction on I. For the second point, let us first show the following lemma:

Lemma 2.1.9. *Let I be an index with at most one index variable i. Then, we have $i^{d(I)} \leq I \leq I\{1/i\} \cdot i^{d(I)}$.*

This is proved directly by induction on indices, and it uses the fact that the constant integers in indices are non-zero, the image of a variable in a valuation is non-zero and an index is always positive.

Now, we prove our theorem by contraposition. Given I, J such that $d(I) > d(J)$, we construct two new indices called I' and J' that are I and J in which we replaced all variables by a new fresh variable i. The degree stays the same, and we have, by Lemma 2.1.9:

$$i^{d(J)+1} \leq i^{d(I)} \leq I' \text{ and } J' \leq i^{d(J)} \cdot J'\{1/i\}.$$

If we replace i by $k = (J'\{1/i\}{+}1)$ (which is a non-zero integer), we obtain:

$$I'\{k/i\} \geq k^{d(J)+1} \text{ and } J'\{k/i\} \leq k^{d(J)} \cdot (k{-}1).$$

And so we have $I'\{k/i\} > J'\{k/i\}$. We deduce that we have a valuation ρ that sends all variables of I and J to k such that $I_\rho > J_\rho$, so we do not have $I \leq J$. By contraposition, we obtain Point 2 of Theorem 2.1.2. □

This second point shows that our notion of degree is well-defined w.r.t. the equivalence relation between indices.

2.2 Elementary Affine Logic and Sizes

2.2.1 An Elementary Affine Lambda Calculus

We work on an elementary affine lambda calculus based on [83] without multithreading and side-effects, that we present here. In this calculus, any sequence of reduction terminates in elementary time. The keystone of this proof is the use of the modality "!", called *bang*, inspired by linear logic. As in linear logic, there are restrictions for the duplication of variables in this

$$\frac{}{\Gamma, x : T \mid \Delta \vdash x : T}\ (\text{Lin Ax}) \qquad \frac{}{\Gamma \mid \Delta, x : T \vdash x : T}\ (\text{Glob Ax})$$

$$\frac{\Gamma, x : T \mid \Delta \vdash M : U}{\Gamma \mid \Delta \vdash \lambda x.M : T \multimap U}\ (\lambda) \qquad \frac{\Gamma \mid \Delta \vdash M : U \multimap T \qquad \Gamma' \mid \Delta \vdash N : U}{\Gamma, \Gamma' \mid \Delta \vdash M\ N : T}\ (\text{App})$$

$$\frac{\cdot \mid \Delta \vdash M : T}{\Gamma \mid \Delta', [\Delta] \vdash !M :\ !T}\ (!\ \text{Intro}) \qquad \frac{\Gamma \mid \Delta \vdash M :\ !T \qquad \Gamma' \mid \Delta, x : [T] \vdash N : U}{\Gamma, \Gamma' \mid \Delta \vdash \texttt{let}\ !x = M\ \texttt{in}\ N : U}\ (!\ \text{Elim})$$

$$\frac{\Gamma \mid \Delta \vdash M : T \qquad \alpha \text{ fresh in } \Gamma,\ \Delta}{\Gamma \mid \Delta \vdash M : \forall \alpha.T}\ (\forall\ \text{Intro}) \qquad \frac{\Gamma \mid \Delta \vdash M : \forall \alpha.T}{\Gamma \mid \Delta \vdash M : T\{U/\alpha\}}\ (\forall\ \text{Elim})$$

Figure 2.5: Type system for the EAL-calculus.

calculus. Moreover, the bang has a more limited use than in linear logic, this limitation gives birth to the notion of *depth*. This notion is crucial to derive the elementary bound on this calculus. To describe formally this calculus, we follow the presentation from [83] and we encode the usual restrictions in a type system.

Definition 2.2.1. *The set of* terms *is given by the grammar:*

$$M, N := x \mid \lambda x.M \mid M\ N \mid !M \mid \texttt{let}\ !x = M\ \texttt{in}\ N.$$

The constructor $\texttt{let}\ !x = M\ \texttt{in}\ N$ binds the variable x in N. We define as usual the notion of free variables, free occurrences and substitution.

Definition 2.2.2. *The* semantic *of this calculus is given by the rules:*

$$(\lambda x.M)\ N \to M\{N/x\} \qquad \texttt{let}\ !x = !M\ \texttt{in}\ N \to N\{M/x\}$$

Those rules can be applied in any context.

We add to this calculus a polymorphic type system that also restrains the possible terms we can write. Types are given by the grammar:

$$T, U := \alpha \mid T \multimap U \mid !T \mid \forall \alpha.T.$$

Definition 2.2.3 (Typing Contexts)**.** Linear variables contexts *are denoted* Γ*, with the shape* $\Gamma = x_1 : T_1, \ldots, x_n : T_n$*. We write* Γ_1, Γ_2 *the disjoint union between* Γ_1 *and* Γ_2*.* Global variables contexts *are denoted* Δ*, with the shape* $\Delta = x_1 : T_1, \ldots, x_n : T_n, y_1 : [T'_1], \ldots, y_m : [T'_m]$*. We say that* $[T]$ *is a discharged type, as we could see in light linear logic [52, 92]. A global variable context* $x_1 : T_1, \ldots, x_n : T_n, y_1 : [T'_1], \ldots, y_m : [T'_m]$ *may sometimes be denoted by* $\Delta, [\Delta']$*. In this case, we consider that* $\Delta = x_1 : T_1, \ldots, x_n : T_n$ *and* $\Delta' = y_1 : T'_1, \ldots, y_m : T'_m$*.*

Typing judgments have the shape $\Gamma \mid \Delta \vdash M : T$. The intended meaning is that variables in Γ are used linearly in M while variables in Δ can be used non-linearly in M.

The rules are given in Figure 2.5. Observe that all the rules are multiplicative for Γ and, seen from bottom to top, the "!Intro" rule erases linear contexts, non-discharged types and transforms discharged types into usual types. With this, we can see that some restrictions appear in a typed term. First, in $\lambda x.M$, x occurs at most once in M, and moreover, there is no "! Intro" rule below the axiom rule for x. Then, in $\texttt{let}\ !x = M\ \texttt{in}\ M'$, x can be used several times in M', but there is exactly one "! Intro" rule below each axiom rule for x. For example, with this type system, we can not type terms like $\lambda x.!x$, $\lambda f, x.f\ (f\ x)$, $\texttt{let}\ !x = M\ \texttt{in}\ x$ or $\texttt{let}\ !x = M\ \texttt{in}\ !!x$.

With this type system, we obtain as a consequence of the results exposed in [83] that any sequence of reductions of a typed term terminates in elementary time. This proof relies on the notion of depth linked with the modality "!" and a measure on terms bounding the number of reductions for this term. We will adapt those two notions in the following part on sEAL, but for now, let us present some terms and encoding in this EAL-calculus.

Examples of Terms in EAL and Church Integers

First, a useful term $\texttt{fonct} : \forall \alpha, \alpha'.!(\alpha \multimap \alpha') \multimap !\alpha \multimap !\alpha'$:

$$\texttt{fonct} := \lambda f, x.\texttt{let } !g = f \texttt{ in let } !y = x \texttt{ in } !(g\ y).$$

We sketch the typing derivation for this term.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{f{:}!(\alpha \multimap \alpha') \mid \cdot \vdash f{:}!(\alpha \multimap \alpha')} \quad \dfrac{\dfrac{}{x{:}!\alpha \mid g : [(\alpha \multimap \alpha')] \vdash x{:}!\alpha} \quad \dfrac{\dfrac{\dots}{\cdot \mid g : (\alpha \multimap \alpha'), y : \alpha \vdash g\ y : \alpha'}}{\cdot \mid g : [(\alpha \multimap \alpha')], y : [\alpha] \vdash !(g\ y){:}!\alpha'}}{x{:}!\alpha \mid g : [(\alpha \multimap \alpha')] \vdash \texttt{let } !y = x \texttt{ in } !(g\ y){:}!\alpha'}}{f{:}!(\alpha \multimap \alpha'), x{:}!\alpha \mid \cdot \vdash \texttt{let } !g = f \texttt{ in let } !y = x \texttt{ in } !(g\ y){:}!\alpha'}}{f{:}!(\alpha \multimap \alpha') \mid \cdot \vdash \lambda x.\texttt{let } !g = f \texttt{ in let } !y = x \texttt{ in } !(g\ y){:}!\alpha \multimap !\alpha'}}{\cdot \mid \cdot \vdash \texttt{fonct}{:}!(\alpha \multimap \alpha') \multimap !\alpha \multimap !\alpha'}}{\cdot \mid \cdot \vdash \texttt{fonct} : \forall \alpha'.!(\alpha \multimap \alpha') \multimap !\alpha \multimap !\alpha'}}{\cdot \mid \cdot \vdash \texttt{fonct} : \forall \alpha, \alpha'.!(\alpha \multimap \alpha') \multimap !\alpha \multimap !\alpha'}$$

This term allows application under a bang. Indeed, the following reduction can be derived:

$$\texttt{fonct } !M\ !N \to^* \texttt{let } !g = !M \texttt{ in let } !y = !N \texttt{ in } !(g\ y) \to^* !(M\ N)$$

Unary integers can be encoded in this calculus as Church integers, with $\mathsf{N} = \forall \alpha.!(\alpha \multimap \alpha) \multimap !(\alpha \multimap \alpha)$. For example, 3 is represented by the term:

$$\underline{3} = \lambda f.\texttt{let } !g = f \texttt{ in } !(\lambda x.g\ (g\ (g\ x))) : \mathsf{N}.$$

And we can represent addition and multiplication with type $\mathsf{N} \multimap \mathsf{N} \multimap \mathsf{N}$:

$$\texttt{add} = \lambda n, m, f.\texttt{let } !f' = f \texttt{ in let } !g = n\ !f' \texttt{ in let } !h = m\ !f' \texttt{ in } !(\lambda x.h\ (g\ x)).$$

$$\texttt{mult} = \lambda n, m, f.\texttt{let } !g = f \texttt{ in } n(m\ !g).$$

And finally, one can also define an iterator using integers:

$$\texttt{iter} = \lambda f, x, n.\texttt{fonct}\ (n\ f)\ x : \forall \alpha.!(\alpha \multimap \alpha) \multimap !\alpha \multimap \mathsf{N} \multimap !\alpha$$

with $\texttt{iter } !M\ !M'\ \underline{n} \to^* !(M^n\ M')$.

Intensional Expressivity

Those examples show that this calculus suffers from limitations. First, we need to work with Church integers, because of a lack of data structures. Furthermore, we need to be careful with the modality, and this can be sometimes a bit tricky, as one can observe with the addition. And finally if we want to do an iteration, we are forced to work with types with bangs. This implies that each time we need to use an iteration, we are forced to add a bang in the final type. Typically, this prevents from iterating a function which has itself been defined by iteration. It has been proved that polynomial and exponential complexity classes can be characterized

in a variant of this calculus with recursive types [10]. For example, with a type for words W and booleans B we have that $!\mathsf{W} \multimap !!\mathsf{B}$ characterizes polynomial time computation. However, because of the restrictions mentioned above, some natural polynomial time programs cannot be typed with the type $!\mathsf{W} \multimap !!\mathsf{B}$. We say that this calculus has a limited intensional expressivity. One goal of this chapter is to try to lessen this problem, and for that, we now present an enriched version of this calculus, sEAL, using the language $\mathsf{s}\ell\mathsf{T}$.

2.2.2 EAL enriched with sizes: sEAL

In order to solve improve the intensional expressivity of this calculus, we enrich it with constructors for integers, words and booleans, and some iterators on those types following the usual constraint on iteration in elementary affine logic (EAL). Then, based on the fact that the proof of correctness in [83] is robust enough to support functions computable in polynomial time with type $\mathsf{N} \multimap \mathsf{N}$ (Remark 2.2.1 will explain this in detail), we enrich EAL with the polynomial time calculus defined previously. We call this new language sEAL (EAL with sizes). More precisely, we add the possibility to use first-order $\mathsf{s}\ell\mathsf{T}$ terms in this calculus in order to work on those base types, particularly we can then do controlled iterations for those types. We then adapt the measure used in [83] to sEAL to find an upper-bound on the number of reductions for a term.

Let us first give some notations on vectors.

Definition 2.2.4 (Vectors)**.** *In the following we will work with vectors of $\mathbb{N}^{n+1}$, for $n \in \mathbb{N}$. We introduce here some notations on those vectors. We usually denote vectors by $\mu = (\mu(0), \dots, \mu(n))$.*

When there is no ambiguity with the value of n, for $0 \leq k \leq n$, we note $\mathbb{1}_k$ for the vector μ with $\mu(k) = 1$ and $\forall i, 0 \leq i \leq n, i \neq k, \mu(i) = 0$. We extend this notation for $k > n$. In this case, $\mathbb{1}_k$ is the zero-vector.

Let $\mu_0, \mu_1 \in \mathbb{N}^{n+1}$. We write $\mu_0 \leq \mu_1$ when $\forall i, 0 \leq i \leq n, \mu_0(i) \leq \mu_1(i)$. We also write $\mu_0 \leq_{lex} \mu_1$ for the lexicographic order on vectors.

For $k \in \mathbb{N}$, when there is no ambiguity with the value of n, we write $\widetilde{k}$ the vector μ such that $\forall i, 0 \leq i \leq n, \mu(i) = k$.

Then, the concatenation of two vectors is denoted by (μ_0, μ_1), the pointwise addition by $\mu_0 + \mu_1$ and the scalar product by $k \cdot \mu$.

Terms and Reductions

Terms of sEAL are defined by the following grammar:

$$\begin{aligned} M, N ::= {} & x \mid \lambda x.M \mid M\ N \mid !M \mid \mathtt{let}\ !x = M\ \mathtt{in}\ N \mid M \otimes N \mid \mathtt{let}\ x \otimes y = M\ \mathtt{in}\ N \\ & \mid \underline{0} \mid \mathtt{s}(M) \mid \mathtt{ifn}(M, N) \mid \mathtt{iter}^{!}_{\mathsf{N}}(M, N) \mid \mathtt{tt} \mid \mathtt{ff} \mid \mathtt{if}(M, N) \\ & \mid \epsilon \mid \mathtt{s}_i(M) \mid \mathtt{ifw}(M_0, M_1, N) \mid \mathtt{iter}^{!}_{\mathsf{W}}(M_0, M_1, N) \mid [\lambda x_n \dots x_1.t](M_1, \dots, M_n) \end{aligned}$$

with $i \in \{0, 1\}$.

Note that the t used in $[\lambda x_n \dots x_1.t](M_1, \dots, M_n)$ refers to terms defined in $\mathsf{s}\ell\mathsf{T}$. This notation means that we call the function t defined in $\mathsf{s}\ell\mathsf{T}$ with arguments $M_1, \dots, M_n$. Moreover, n can be any integer, even zero. Constructors for iterations directly follow from the ones we can usually define in EAL for Church integers or Church words, as we could see in the previous section on EAL. As usual, we work up to α-isomorphism and we do not explicit the renaming of variables. As before, for words, the choice of the alphabet $\Sigma = \{0, 1\}$ is arbitrary, we could have chosen any finite alphabet.

$$
\begin{array}{rl|rl}
(\lambda x.M)\ N & \to M\{N/x\} & \texttt{let}\ !x = !M\ \texttt{in}\ N & \to N\{M/x\} \\
\texttt{let}\ x \otimes y = M \otimes M'\ \texttt{in}\ N & \to N\{M/x\}\{M'/y\} & \texttt{ifn}(M, N)\ \underline{0} & \to N \\
\texttt{ifn}(M, M')\ \texttt{s}(N) & \to M\ N & \texttt{iter}^{!}_{\mathsf{N}}(!M, !N)\ \underline{n} & \to !(M^n\ N) \\
\texttt{ifw}(M_0, M_1, N)\ \epsilon & \to N & \texttt{ifw}(M_0, M_1, M')\ \texttt{s}_i(N) & \to M_i\ N \\
\texttt{iter}^{!}_{\mathsf{W}}(!M_0, !M_1, !N)\ \underline{w} & \to !(M^w\ N) & \texttt{if}(M, N)\ \texttt{tt} & \to M \\
\texttt{if}(M, N)\ \texttt{ff} & \to N & \text{if } t \to t' \text{ in } \texttt{s}\ell\mathsf{T} \text{ then } [t]() & \to [t']()
\end{array}
$$

$$
\begin{array}{rl}
[\lambda x_n \dots x_1.t](M_1, \dots, M_{n-1}, \underline{v}) & \to [\lambda x_{n-1} \dots x_1.t\{\underline{v}/x_n\}](M_1, \dots, M_{n-1}) \\
[\underline{v}]() & \to \underline{v}
\end{array}
$$

Figure 2.6: Base rules for sEAL.

Definition 2.2.5 (Base type values)**.** *We note $\underline{v}$ for base type values, defined by the grammar:*

$$\underline{v} := \underline{0} \mid \texttt{s}(\underline{v}) \mid \epsilon \mid \texttt{s}_i(\underline{v}) \mid \texttt{tt} \mid \texttt{ff}$$

with $i \in \{0, 1\}$.

In particular, if n is an integer and w is a binary word, we note $\underline{n}$ for the base value $\texttt{s}^n(\underline{0})$, and $\underline{w} = \underline{w_1 \cdots w_n}$ for the base value $\texttt{s}_{w_1}(\dots \texttt{s}_{w_n}(\epsilon)\dots)$. We also define the size *$|\underline{v}|$ of $\underline{v}$.*

$$|\underline{0}| = |\epsilon| = |\texttt{tt}| = |\texttt{ff}| = 1 \qquad |\texttt{s}(\underline{v})| = |\texttt{s}_i(\underline{v})| = 1 + |\underline{v}|$$

We may use the following notation for terms.

Definition 2.2.6 (Iterated Applications)**.** *For terms M, M' and an integer n, we write $M^n M'$ to denote n applications of M to M'. In particular, $M^0 M' = M'$. We also define for a word w, given terms M_a for all letter a, $M^w M'$. This is defined by induction on words with $M^\epsilon M' = M'$ and $M^{aw'} M' = M_a\ (M^{w'} M')$.*

Base reductions are defined by the rules given in Figure 2.6. Note that for some of these rules, for example the last one, $\underline{v}$ can denote either the $\texttt{s}\ell\mathsf{T}$ term or the sEAL term.

Contexts are defined by:

$$
\begin{array}{l}
C ::= [] \mid \lambda x.C \mid C\ N \mid M\ C \mid !C \mid \texttt{let}\ !x = C\ \texttt{in}\ N \mid \texttt{let}\ !x = M\ \texttt{in}\ C \mid C \otimes N \mid M \otimes C \\
\quad \mid \texttt{let}\ x \otimes y = C\ \texttt{in}\ N \mid \texttt{let}\ x \otimes y = M\ \texttt{in}\ C \mid \texttt{s}(C) \mid \texttt{ifn}(C, N) \mid \texttt{ifn}(M, C) \\
\quad \mid \texttt{iter}^{!}_{\mathsf{N}}(C, N) \mid \texttt{iter}^{!}_{\mathsf{N}}(M, C) \mid \texttt{if}(C, N) \mid \texttt{if}(M, C) \mid \texttt{s}_i(C) \mid \texttt{ifw}(C, M_1, N) \\
\quad \mid \texttt{ifw}(M_0, C, N) \mid \texttt{ifw}(M_0, M_1, C) \mid \texttt{iter}^{!}_{\mathsf{W}}(C, M_1, N) \mid \texttt{iter}^{!}_{\mathsf{W}}(M_0, C, N) \\
\quad \mid \texttt{iter}^{!}_{\mathsf{W}}(M_0, M_1, C) \mid [\lambda x_n \dots x_1.t](M_1, \dots, M_{j-1}, C, M_{j+1}, \dots, M_n).
\end{array}
$$

with $i \in \{0, 1\}$ and $j \in \{1, \dots, n\}$.

The reductions can be extended to any context, and so we have $M \to M'$ if there is a context C and a base reduction $M_0 \to M_0'$ such that $M = C[M_0]$ and $M' = C[M_0']$. In order to work with $\texttt{s}\ell\mathsf{T}$, we use the three last rules and contexts: from the term $[\lambda x_n \dots x_1.t](M_1, \dots, M_n)$, we could start by reducing the term M_n to obtain $[\lambda x_n \dots x_1.t](M_1, \dots, M_{n-1}, \underline{v})$, then use the second last reduction rule to obtain $[\lambda x_{n-1} \dots x_1.t\{\underline{v}/x_n\}](M_1, \dots, M_{n-1})$, and repeat n times to obtain a term of the form $[t']()$. We can then reduce this term t to a normal form $\underline{v}$ in $\texttt{s}\ell\mathsf{T}$ and we obtain in sEAL the term $[\underline{v}]()$. Finally, with the last rule we obtain the value $\underline{v}$. Note that this order for the reduction is not mandatory as contexts do not impose to always start by reducing M_n.

$$\pi \lhd \frac{}{\Gamma, x : T \mid \Delta \vdash x : T} \qquad \mu_n(\pi) = \mathbb{1}_0$$

$$\pi \lhd \frac{}{\Gamma \mid \Delta, x : T \vdash x : T} \qquad \mu_n(\pi) = \mathbb{1}_0$$

$$\pi \lhd \frac{\sigma \lhd \Gamma, x : T \mid \Delta \vdash M : U}{\Gamma \mid \Delta \vdash \lambda x.M : T \multimap U} \qquad \mu_n(\pi) = \mu_n(\sigma) + \mathbb{1}_0$$

$$\pi \lhd \frac{\sigma \lhd \Gamma \mid \Delta \vdash M : U \multimap T \qquad \tau \lhd \Gamma' \mid \Delta \vdash N : U}{\Gamma, \Gamma' \mid \Delta \vdash M\ N : T} \qquad \mu_n(\pi) = \mu_n(\sigma) + \mu_n(\tau) + \mathbb{1}_0$$

$$\pi \lhd \frac{\sigma \lhd \cdot \mid \Delta \vdash M : T}{\Gamma \mid \Delta', [\Delta] \vdash !M : \ !T} \qquad \mu_n(\pi) = (1, \mu_{n-1}(\sigma))$$

$$\pi \lhd \frac{\sigma \lhd \Gamma \mid \Delta \vdash M : \ !T \qquad \tau \lhd \Gamma' \mid \Delta, x : [T] \vdash N : U}{\Gamma, \Gamma' \mid \Delta \vdash \texttt{let}\ !x = M\ \texttt{in}\ N : U} \qquad \mu_n(\pi) = \mu_n(\sigma) + \mu_n(\tau) + \mathbb{1}_0$$

$$\pi \lhd \frac{\sigma \lhd \Gamma \mid \Delta \vdash M : T \qquad \tau \lhd \Gamma' \mid \Delta \vdash N : U}{\Gamma, \Gamma' \mid \Delta \vdash M \otimes N : T \otimes U} \qquad \mu_n(\pi) = \mu_n(\sigma) + \mu_n(\tau) + \mathbb{1}_0$$

$$\pi \lhd \frac{\sigma \lhd \Gamma \mid \Delta \vdash M : S \otimes U \qquad \tau \lhd \Gamma', x : S, y : U \mid \Delta \vdash N : T}{\Gamma, \Gamma' \mid \Delta \vdash \texttt{let}\ x \otimes y = M\ \texttt{in}\ N : T} \qquad \mu_n(\pi) = \mu_n(\sigma) + \mu_n(\tau) + \mathbb{1}_0$$

$$\pi \lhd \frac{\sigma \lhd \Gamma \mid \Delta \vdash M : T \qquad \alpha \text{ fresh in } \Gamma,\ \Delta}{\Gamma \mid \Delta \vdash M : \forall \alpha.T} \qquad \mu_n(\pi) = \mu_n(\sigma)$$

$$\pi \lhd \frac{\sigma \lhd \Gamma \mid \Delta \vdash M : \forall \alpha.T}{\Gamma \mid \Delta \vdash M : T\{U/\alpha\}} \qquad \mu_n(\pi) = \mu_n(\sigma)$$

Figure 2.7: Type and measure for generic constructors in sEAL.

Types

Types are usual types for intuitionistic linear logic enriched with some base types for booleans, integers and words.

$$A := \mathsf{B} \mid \mathsf{N} \mid \mathsf{W} \qquad T, U, S := \alpha \mid A \mid T \multimap U \mid !T \mid T \otimes U \mid \forall \alpha.T$$

A type A is called a base type. The type for words W depends on the choice of the alphabet Σ.

Definition 2.2.7 (Contexts and Type System)**.** Linear variables contexts *are denoted* Γ *and* global variables contexts *are denoted* Δ*. They are defined in the same way as in the previous part on the EAL-calculus. Typing judgments have the usual shape of dual contexts judgments* $\pi \lhd \Gamma \mid \Delta \vdash M : T$*. For such a proof* π*, and* $i \in \mathbb{N}$*, we define a weight* $\omega_i(\pi) \in \mathbb{N}$*.*

Definition 2.2.8 (Measure and Depth)**.** *For all integers* k *and* n*, we use the notation* $\mu_n^k(\pi) = (\omega_k(\pi), \ldots, \omega_n(\pi))$*, with the convention that if* $k > n$*, then* $\mu_n^k(\pi)$ *is the empty vector. We write* $\mu_n(\pi)$ *to denote the vector* $\mu_n^0(\pi)$*. In the definitions given in the type system, instead of defining* $\omega_i(\pi)$ *for all* i*, we define* $\mu_n(\pi)$ *for all* n*, from which one can recover the weights. We will often call* $\mu_n(\pi)$ *the measure of the proof* π*. The* depth *of a proof* π *(or a typed term), denoted depth*(π)*, is the greatest integer* i *such that* $\omega_i(\pi) \neq 0$*. It is always defined for any proof.*

The idea behind the definition of measure is to show that with a reduction step, this measure strictly decreases for the lexicographic order and we can control the growing of the weights. The rules are given on Figure 2.7, Figure 2.8 and Figure 2.9.

The rules given in Figure 2.7 represent the usual constructors in EAL. Those rules impose some restrictions on the use of variables similar to the ones described in the previous section

$$\pi\lhd \frac{}{\Gamma \mid \Delta \vdash \underline{0} : \mathsf{N}} \qquad \mu_n(\pi) = \mathbb{1}_1$$

$$\pi\lhd \frac{\sigma \lhd \Gamma \mid \Delta \vdash M : \mathsf{N}}{\Gamma \mid \Delta \vdash \mathtt{s}(M) : \mathsf{N}} \qquad \mu_n(\pi) = \mu_n(\sigma)+\mathbb{1}_1$$

$$\pi\lhd \frac{\sigma \lhd \Gamma \mid \Delta \vdash M : \mathsf{N} \multimap T \qquad \tau \lhd \Gamma' \mid \Delta \vdash N : T}{\Gamma,\Gamma' \mid \Delta \vdash \mathtt{ifn}(M,N) : \mathsf{N} \multimap T} \qquad \mu_n(\pi) = \mu_n(\sigma)+\mu_n(\tau)+\mathbb{1}_0$$

$$\pi\lhd \frac{\sigma \lhd \Gamma \mid \Delta \vdash M : \ !(T \multimap T) \qquad \tau \lhd \Gamma' \mid \Delta \vdash N : \ !T}{\Gamma,\Gamma' \mid \Delta \vdash \mathtt{iter}^!_{\mathsf{N}}(M,N) : \mathsf{N} \multimap !T} \qquad \mu_n(\pi) = \mu_n(\sigma)+\mu_n(\tau)+\mathbb{1}_0$$

$$\pi\lhd \frac{}{\Gamma \mid \Delta \vdash \epsilon : \mathsf{W}} \qquad \mu_n(\pi) = \mathbb{1}_1$$

$$\pi\lhd \frac{\sigma \lhd \Gamma \mid \Delta \vdash M : \mathsf{W}}{\Gamma \mid \Delta \vdash \mathtt{s}_i(M) : \mathsf{W}} \qquad \mu_n(\pi) = \mu_n(\sigma)+\mathbb{1}_1$$

$$\pi\lhd \frac{\forall i, \sigma_i \lhd \Gamma_i \mid \Delta \vdash M_i : \mathsf{W} \multimap T \qquad \tau \lhd \Gamma' \mid \Delta \vdash N : T}{\Gamma_1,\Gamma_2,\Gamma' \mid \Delta \vdash \mathtt{ifw}(M_0,M_1,N) : \mathsf{W} \multimap T} \qquad \mu_n(\pi) = \mu_n(\sigma_1)+\mu_n(\sigma_2)+\mu_n(\tau)+\mathbb{1}_0$$

$$\pi\lhd \frac{\forall i, \sigma_i \lhd \Gamma_i \mid \Delta \vdash M_i : \ !(T \multimap T) \qquad \tau \lhd \Gamma' \mid \Delta \vdash N : \ !T}{\Gamma_1,\Gamma_2,\Gamma' \mid \Delta \vdash \mathtt{iter}^!_{\mathsf{W}}(M,N) : \mathsf{W} \multimap !T} \qquad \mu_n(\pi) = \mu_n(\sigma_1)+\mu_n(\sigma_2)+\mu_n(\tau)+\mathbb{1}_0$$

$$\pi\lhd \frac{}{\Gamma \mid \Delta \vdash \mathtt{tt} : \mathsf{B}} \qquad \mu_n(\pi) = \mathbb{1}_1$$

$$\pi\lhd \frac{}{\Gamma \mid \Delta \vdash \mathtt{ff} : \mathsf{B}} \qquad \mu_n(\pi) = \mathbb{1}_1$$

$$\pi\lhd \frac{\sigma \lhd \Gamma \mid \Delta \vdash M : T \qquad \tau \lhd \Gamma' \mid \Delta \vdash N : T}{\Gamma,\Gamma' \mid \Delta \vdash \mathtt{if}(M,N) : \mathsf{B} \multimap T} \qquad \mu_n(\pi) = \mu_n(\sigma)+\mu_n(\tau)+\mathbb{1}_0$$

Figure 2.8: Type and measure for constructors on base types in sEAL.

$$\pi\lhd \frac{\forall \ell, (1 \le \ell \le k), \sigma_\ell \lhd \Gamma_\ell \mid \Delta \vdash \ M_\ell : A_\ell \qquad \tau \lhd x_1 : A_1^{(i_1)}, \ldots, x_k : A_k^{(i_k)} \vdash_{\mathsf{s}\ell\mathsf{T}} t : U}{\Gamma,\Gamma_1,\ldots,\Gamma_k \mid \Delta \vdash [\lambda x_k \ldots x_1.t](M_1,\ldots,M_k) : \mathtt{type}(U)}$$

$$\mu_n(\pi) = \sum_{\ell=1}^{k} \mu_n(\sigma_\ell)+k(d(\omega(\tau)+I)+1)\cdot\mathbb{1}_0+((\omega(\tau)+I)\{1/j_1\}\cdots\{1/j_l\}+1)\cdot\mathbb{1}_1$$
where $I = \mathtt{ind}(U)$ and $\{j_1,\ldots,j_l\} = \mathtt{Var}(\omega(\tau)) \cup \mathtt{Var}(I)$.

Figure 2.9: Typing rule and measure for the $\mathsf{s}\ell\mathsf{T}$ call in sEAL.

on classical EAL. Observe that the constructors for base type values such as $\underline{0}$ and $\mathtt{s}$ given in Figure 2.8 influence the weight only in position one and not zero like the others constructors. As a consequence, if you take for example a proof of $\cdot \vdash \underline{v} : \mathsf{W}$, then this proof has depth 1.

For the rule given by Figure 2.9, we first introduce some notations.

Definition 2.2.9 (Base Types in $\mathsf{s}\ell\mathsf{T}$ and sEAL)**.** *For a base type A of* sEAL *and an index I, we define a base type $A^{(I)}$ in* $\mathsf{s}\ell\mathsf{T}$*:*

$$\mathsf{B}^{(I)} := \mathsf{B} \qquad \mathsf{N}^{(I)} := \mathsf{N}^I \qquad \mathsf{W}^{(I)} := \mathsf{W}^I$$

Reciprocally, for a base type U in $\mathsf{s}\ell\mathsf{T}$*, we define a type in* sEAL $\mathtt{type}(U)$ *and an index* $\mathtt{ind}(U)$*.*

- $\mathtt{type}(\mathsf{B}) := \mathsf{B}$ *and* $\mathtt{ind}(\mathsf{B}) = 1$*.*
- $\mathtt{type}(\mathsf{N}^I) := \mathsf{N}$ *and* $\mathtt{ind}(\mathsf{N}^I) = I$*.*

- $\mathtt{type}(\mathsf{W}^I) := \mathsf{W}$ *and* $\mathtt{ind}(\mathsf{W}^I) = I$.

Note that we associate the index 1 *to* B *since boolean values are either* `tt` *or* `ff`*, thus values of size* 1*.*

The premise for t is a proof τ in $\mathsf{s}\ell\mathsf{T}$. In this proof, we add on each non-boolean base type A_i an index variable a_i. This proof τ must yield a base type U, and this converts to a base type in sEAL. Moreover, the previous section gives us a weight $\omega(\tau)$ for this proof in $\mathsf{s}\ell\mathsf{T}$.

Let us now comment on the definition of $\mu_n(\pi)$. First, $\mathtt{Var}(I)$ is a notation for the set of variables in I. Then, at position 0 in the weight, we put k times the degree of $\omega(\tau)$ and $\mathtt{ind}(U)$. Indeed, as one could see in the incoming definition of $\mathcal{R}ed$ (Definition 2.2.11) or more informally in Remark 2.2.1, having k times the degree at position 0 allows the future k substitution of $x_1, \ldots, x_k$ by their actual value. Then, as this term outputs a base type, and as base types have their size at position 1 in the weight, we add at position 1 an expression that will allow us to bound the number of reductions in $\mathsf{s}\ell\mathsf{T}$ and the size of the output. Furthermore, remark that when $k = 0$, the term $[t]()$ influences only the weight at position 1, such as constructors for base types.

2.2.3 Subject Reduction and Measure

In this section, we show that we can bound the number of reduction steps of a typed term using the measure. This is done by first showing that a reduction preserves some properties on the measure, and then by giving an explicit integer bound that will strictly decrease after a reduction. This proof is inspired by the methodology of [83]. The relation $\mathcal{R}ed$ defined in the following is a generalization of the usual requirements exposed in elementary linear logic in order to control reductions.

Let us first express that type variables can be substituted.

Lemma 2.2.1 (Substitution of Type Variables)**.** *Suppose that* $\pi \lhd \Gamma \mid \Delta \vdash M : T$*. Then, for every type variable* α *and for every type* U*, we can derive a proof* $\pi\{U/\alpha\} \lhd \Gamma\{U/\alpha\} \mid \Delta\{U/\alpha\} \vdash M : T\{U/\alpha\}$ *with* $\forall n, \mu_n(\pi) = \mu_n(\pi\{U/\alpha\})$*.*

Proof. By induction on π. All cases are straightforward, we just need to be a bit careful with the renaming of variables for the introduction of $\forall$ and choose the good instantiation for the elimination of $\forall$. □

Let us then express substitution lemmas for sEAL. There are 3 cases to consider: linear variables, discharged global variables and non-discharged global variables.

Lemma 2.2.2 (Linear Substitution)**.** *Suppose that* $\pi \lhd \Gamma_1, x : T' \mid \Delta \vdash M : T$ *and* $\sigma \lhd \Gamma_2 \mid \Delta \vdash M' : T'$*, then, we have a proof* $\pi' \lhd \Gamma_1, \Gamma_2 \mid \Delta \vdash M\{M'/x\} : T$*. Moreover, for all* n*,* $\mu_n(\pi') \leq \mu_n(\pi) + \mu_n(\sigma)$*.*

Proof. The proof comes from the fact that rules are multiplicative for Γ and so x only appears in one of the premises for each rule. Thus, the proof σ is used only once in the new proof π'. □

Lemma 2.2.3 (General Substitution)**.** *Suppose that* $\pi \lhd \Gamma \mid \Delta, x : T' \vdash M : T$ *and* $\sigma \lhd \cdot \mid \Delta \vdash M' : T'$ *and the number of occurrences of* x *in* M *is less than* K*, then we have a proof* $\pi' \lhd \Gamma \mid \Delta \vdash M\{M'/x\} : T$*. Moreover, for all* n*,* $\mu_n(\pi') \leq \mu_n(\pi) + K \cdot \mu_n(\sigma)$*.*

Proof. This time, the non-linearity of the variable x induces a duplication of the proof σ, that is why the measure $\mu_n(\sigma)$ is also duplicated. □

Lemma 2.2.4 (Discharged Substitution)**.** *If $\pi \lhd \Gamma \mid \Delta', [\Delta], x : [T'] \vdash M : T$ and $\sigma \lhd \cdot \mid \Delta \vdash M' : T'$ then we have a proof $\pi' \lhd \Gamma \mid \Delta', [\Delta] \vdash M\{M'/x\} : T$. Moreover, for all n, $\mu_n(\pi') \leq (\omega_0(\pi), (\mu_n^1(\pi) + \omega_1(\pi) \cdot \mu_{n-1}(\sigma)))$.*

Proof. The proof of this lemma relies directly on Lemma 2.2.3. Indeed, a variable with a discharged type can be used only after crossing a (!-Intro) rule and then the upper bound on $\mu_n(\pi')$ comes from the previous lemma since the number of occurrences of x in M is less than $\omega_1(\pi)$. □

Next, let us give two important definitions, t_α and $\mathcal{R}ed$, in order to derive the upper bound on the number of reduction steps in sEAL.

Definition 2.2.10 (Tower of Functions)**.** *We define a family of tower functions $t_\alpha(x_1, \ldots, x_n)$ on vectors of integers by induction on n, where we assume $\alpha \geq 1$ and $x_i \geq 2$ for all i: $t_\alpha() = 0$ and $t_\alpha(x_1, \ldots, x_n) = (\alpha \cdot x_n)^{2^{t_\alpha(x_1, \ldots, x_{n-1})}}$ for $n \geq 1$.*

For example, $t_\alpha(3, 4, 5) = (5\alpha)^{2^{(4\alpha)^{2^{3\alpha}}}}$. Note that $t_\alpha(x_1, \ldots, x_n)$ is a polynomial function in x_n (if $x_1, \ldots, x_{n-1}$ are fixed) and a tower of exponential of height $2n$ for x_1 (if $x_2, \ldots, x_n$ are fixed). This function gives us a bound on the measure of a given proof. However, it is not convenient to work with, so we give a sufficient condition on two vectors μ and μ' to have $t_\alpha(\mu') < t_\alpha(\mu)$.

Definition 2.2.11 ($\mathcal{R}ed$)**.** *We define a relation on vectors denoted $\mathcal{R}ed$. Intuitively, we want $\mathcal{R}ed(\mu, \mu')$ to express that a proof of measure μ has been reduced to a proof of measure μ'. Let $\mu, \mu' \in \mathbb{N}^{n+1}$. We have $\mathcal{R}ed(\mu, \mu')$ if and only if the following conditions are satisfied:*

1. *$\mu \geq \widetilde{2}$ and $\mu' \geq \widetilde{2}$.*
2. *$\mu' <_{lex} \mu$: there exists $0 \leq i_0 \leq n$, $\mu = (\omega_0, \ldots, \omega_n)$ and $\mu' = (\omega_0, \ldots, \omega_{i_0 - 1}, \omega'_{i_0}, \ldots, \omega'_n)$, with $\omega_{i_0} > \omega'_{i_0}$.*
3. *There exists $d \in \mathbb{N}, 1 \leq d \leq (\omega_{i_0} - \omega'_{i_0})$ such that for all $j > i_0$, we have $\omega'_j \leq \omega_j \cdot (\omega_{i_0+1})^{d-1}$.*

The first condition with $\widetilde{2}$, that can also be seen in the definition of t_α, makes calculation easier, since with this condition, exponentials and multiplications conserve the strict order between integers. This does not harm the proof, since we can simply add $\widetilde{2}$ to each vector we will consider. For an example of two vectors in relation, we have $\mathcal{R}ed((2, 5, 3, 2), (2, 2, 25, 15))$:

- $(2, 5, 3, 2) \geq (2, 2, 2, 2)$ and $(2, 2, 25, 15) \geq (2, 2, 2, 2)$.
- $(2, 2, 25, 15) <_{lex} (2, 5, 3, 2)$, with $i_0 = 1$.
- If we take $d = 3$, we have indeed $1 \leq d \leq 5 - 2$. Moreover, we have $25 \leq 3 \cdot 3^2 = 27$ and $15 \leq 2 \cdot 3^2 = 18$.

One can see on this example that $\mathcal{R}ed(\mu, \mu')$ indicates that $\mu' <_{lex} \mu$ and the components of the vector μ' are not "too big" compared to μ.

We can then connect those two definitions:

Theorem 2.2.1. *Let $\mu, \mu' \in \mathbb{N}^{n+1}$ and $\alpha \geq n$, $\alpha \geq 1$. If we have $\mathcal{R}ed(\mu, \mu')$ then $t_\alpha(\mu') < t_\alpha(\mu)$.*

In order to prove this theorem, we first give some properties on t_α and $\mathcal{R}ed$. The proofs are often only calculation.

Lemma 2.2.5. *If $\mu \leq \mu'$ then $t_\alpha(\mu) \leq t_\alpha(\mu')$.*

This is just a simple consequence of the fact that the exponentiation is monotonic.

Lemma 2.2.6 (Shift)**.** *Let $k \in \mathbb{N}^*$. Let $\mu = (\omega_0, \ldots, k{\cdot}\omega_{i-1}, \omega_i, \ldots \omega_n)$ and $\mu' = (\omega_0, \ldots, \omega_{i-1}, k{\cdot}\omega_i, \ldots \omega_n)$. Then $t_\alpha(\mu') \leq t_\alpha(\mu)$.*

Proof. Let us define $\mu_0 = (\omega_0, \ldots, \omega_{i-2})$.

$$k \geq 1 \text{ so } k \leq 2^{2^{k-1}-1}, \text{ then}$$

$$k \cdot \omega_i \leq \omega_i \cdot 2^{2^{k-1}-1} \leq (\omega_i)^{2^{k-1}} \text{ since } w_i \geq 2. \text{ So,}$$

$$\alpha \cdot k \cdot \omega_i \leq (\alpha \cdot \omega_i)^{2^{(\alpha\cdot(k-1)\cdot\omega_{i-1})^{2^{t_\alpha(\mu_0)}}}}.$$

$$(\alpha \cdot k \cdot \omega_i)^{2^{(\alpha\cdot\omega_{i-1})^{2^{t_\alpha(\mu_0)}}}} \leq (\alpha \cdot \omega_i)^{2^{(\alpha\cdot(k-1)\cdot\omega_{i-1})^{2^{t_\alpha(\mu_0)}}} \cdot 2^{(\alpha\cdot\omega_{i-1})^{2^{t_\alpha(\mu_0)}}}} \text{ and so,}$$

$$t_\alpha(\mu_0, (\omega_{i-1}, k \cdot \omega_i)) \leq (\alpha \cdot \omega_i)^{2^{(\alpha\cdot k\cdot\omega_{i-1})^{2^{t_\alpha(\mu_0)}}}} = t_\alpha(\mu_0, (k \cdot \omega_{i-1}, \omega_i)).$$

We can now obtain $t_\alpha(\mu') \leq t_\alpha(\mu)$ by monotonicity of exponential. □

Lemma 2.2.7. *If $\widetilde{2} \leq \mu' < \mu$ then $\mathcal{R}ed(\mu, \mu')$.*

Proof. Take $d = 1$ and the proof is simple. □

Lemma 2.2.8. *If $\mathcal{R}ed(\mu, \mu')$ then for all μ_0, we have $\mathcal{R}ed(\mu{+}\mu_0, \mu'{+}\mu_0)$.*

Proof. The conditions 1 and 2 for $\mathcal{R}ed(\mu{+}\mu_0, \mu'{+}\mu_0)$ are given by the hypothesis $\mathcal{R}ed(\mu, \mu')$. We keep the notations $\omega_j, \omega'_j, i_0, d$.

$$1 \leq d \leq \omega_{i_0} - \omega'_{i_0} \text{ so } 1 \leq d \leq (\omega_{i_0}{+}\mu_0(i_0)) - (\omega'_{i_0}{+}\mu_0(i_0)).$$

Let $j > i_0$, we have:

$$\omega'_j{+}\mu_0(j) \leq \omega_j \cdot (\omega_{i_0+1})^{d-1}{+}\mu_0(j) \leq (\omega_j{+}\mu_0(j)) \cdot (\omega_{i_0+1}{+}\mu_0(i_0{+}1))^{d-1}$$

since $\omega_{i_0+1} \geq 1$. □

We now want to prove Theorem 2.2.1.

Proof. Suppose $\mathcal{R}ed(\mu, \mu')$. Using the notations from the definition of $\mathcal{R}ed$, we have:

$$\mu \geq (\omega_0, \ldots, \omega'_{i_0}{+}d, \omega_{i_0+1}, \ldots, \omega_n) \text{ and we have}$$

$$\mu' \leq (\omega_0, \ldots, \omega_{i_0-1}, \omega'_{i_0}, \omega_{i_0+1} \cdot (\omega_{i_0+1})^{d-1}, \ldots, \omega_n \cdot (\omega_{i_0+1})^{d-1}).$$

Let us call $\mu_0 = (\omega_0, \ldots, \omega_{i_0-1})$.

$$\alpha \cdot d \geq 1 \text{ so } \alpha \cdot d < 2^{\alpha\cdot d} \text{ then,}$$

$$\text{as } \omega_{i_0+1} \geq 2, \text{ we have } (\omega_{i_0+1})^{\alpha\cdot d} < (\omega_{i_0+1})^{2^{\alpha\cdot d}} \text{ so,}$$

$$\alpha \cdot (\omega_{i_0+1})^{\alpha\cdot d} < (\alpha \cdot \omega_{i_0+1})^{2^{(\alpha\cdot d)^{2^{t_\alpha(\mu_0)}}}} \text{ and so}$$

$$(\alpha \cdot (\omega_{i_0+1})^{\alpha\cdot d})^{2^{(\alpha\cdot\omega'_{i_0})^{2^{t_\alpha(\mu_0)}}}} < (\alpha \cdot \omega_{i_0+1})^{2^{(\alpha\cdot(d+\omega'_{i_0}))^{2^{t_\alpha(\mu_0)}}}}.$$

$$t_\alpha(\omega_0,\dots,\omega_{i_0-1},\omega'_{i_0},(\omega_{i_0+1})^{\alpha\cdot d}) < t_\alpha(\omega_0,\dots,\omega_{i_0-1},\omega'_{i_0}+d,\omega_{i_0+1}).$$

By Lemma 2.2.5, since $\omega_{i_0+1}\cdot(\omega_{i_0+1})^{(n-i_0)(d-1)} \leq (\omega_{i_0+1})^{\alpha\cdot d}$, and by monotonicity of the exponential, we obtain:

$$t_\alpha(\omega_0,\dots,\omega_{i_0-1},\omega'_{i_0},\omega_{i_0+1}\cdot(\omega_{i_0+1})^{(n-i_0)(d-1)},\dots,\omega_n) < t_\alpha(\omega_0,\dots,\omega'_{i_0}+d,\omega_{i_0+1},\dots,\omega_n).$$

Using several times the shift lemma (Lemma 2.2.6), we obtain:

$$t_\alpha(\omega_0,\dots,\omega_{i_0-1},\omega'_{i_0},\omega_{i_0+1}\cdot(\omega_{i_0+1})^{d-1},\dots,\omega_n\cdot(\omega_{i_0+1})^{d-1}) < t_\alpha(\omega_0,\dots,\omega'_{i_0}+d,\omega_{i_0+1},\dots,\omega_n).$$

Again by Lemma 2.2.5, we obtain $t_\alpha(\mu') < t_\alpha(\mu)$. □

To sum up, this theorem shows that if we want to ensure that a certain integer defined with t_α strictly decreases for a reduction, it is sufficient to work with the relation $\mathcal{R}ed$.

Finally, we need to consider rules for polymorphism. Because of the $\forall$ elimination and introduction rules, our type system is not syntax directed. However, we can prove that for some terms, namely introduction terms, we can "recover" syntax directed rules.

Definition 2.2.12. *A term M_{intro} is said to be an* introduction term *if it has one of those form:*

$$\lambda x.M \mid !M \mid M\otimes N \mid \underline{0} \mid \mathtt{s}(M) \mid \mathtt{ifn}(M,N) \mid \mathtt{iter}^!_{\mathsf{N}}(M,N) \mid \mathtt{tt} \mid \mathtt{ff}$$
$$\mid \mathtt{if}(M,N) \mid \epsilon \mid \mathtt{s}_i(M) \mid \mathtt{ifw}(M_0,M_1,N) \mid \mathtt{iter}^!_{\mathsf{W}}(M_0,M_1,N).$$

To each such term we can associate a typing rule, for instance to $\lambda x.M$ the rule $(\multimap)$, to $!M$ the rule $(!)$ etc. Note that these rules correspond to introduction rules and rules for base type constructors.

For the sake of simplicity, we introduce a notation for list of objects. Let us write $\overline{T}$ to denote a sequence $T_1,\dots,T_n$ of types, and $\forall\overline{\alpha}.T$ to denote the type $\forall\alpha_1\dots\forall\alpha_n.T$ when T does not begin with a quantifier. We can now present the generation lemma.

Lemma 2.2.9 (Generation lemma)**.** *Let M_{intro} be an introduction term. Let π be a typing $\pi \lhd \Gamma \mid \Delta \vdash M_{intro} : \forall\widetilde{\alpha}.T$, where type variables in $\widetilde{\alpha}$ are fresh in Γ and Δ. Let $\widetilde{T'}$ be an instantiation of $\widetilde{\alpha}$. Let $(\boldsymbol{R})$ denote the rule associated to M_{intro} and n its number of premises. Then there exist type derivations $\pi'_1,\dots,\pi'_n$ such that if π' is the derivation obtained by applying the rule $(\boldsymbol{R})$ to $\pi'_1,\dots,\pi'_n$ we have:*

- *π' is a typing of conclusion $\pi' \lhd \Gamma \mid \Delta \vdash M_{intro} : T\{\widetilde{T'}/\widetilde{\alpha}\}$.*
- *For all integer k, $\mu_k(\pi') = \mu_k(\pi)$.*

For example, if $\pi \lhd \Gamma \mid \Delta \vdash \lambda x.M : \forall\widetilde{\alpha}.(T \multimap U)$, then for any sequence of type $\widetilde{T'}$ with same length as $\widetilde{\alpha}$, such that variables in $\widetilde{\alpha}$ are not free in $\widetilde{T'}$, we have a proof $\pi'_1 \lhd \Gamma, x : T\{\widetilde{T'}/\widetilde{\alpha}\} \mid \Delta \vdash M : U\{\widetilde{T'}/\widetilde{\alpha}\}$ with for all k, $\mu_k(\pi) = \mu_k(\pi'_1)+\mathbb{1}_0$.

So for each possible shape for M_{intro}, we can state this lemma more formally just by looking at the associated typing rule. Observe that for terms like $\underline{0}$, this lemma only states that the measure of the proof is exactly $\mathbb{1}_1$.

Proof. This is proved by induction on $\pi \lhd \Gamma \mid \Delta \vdash M_{intro} : \forall\widetilde{\alpha}.T$. Observe that for a given M_{intro} there are only 3 possibles rules: introduction and elimination of $\forall$ and the rule $(\mathbf{R})$ associated to the form of M_{intro}.

- **Rule (R).** This case is trivial, this is exactly the definition of the generation lemma, with $\widetilde{\alpha} = \emptyset$. Observe that for this case, it is important to consider only introduction terms, otherwise there is no reason that $\widetilde{\alpha} = \emptyset$.

- **Elimination of $\forall$.** Suppose we have the proof

$$\pi\lhd \frac{\tau \lhd \Gamma \mid \Delta \vdash M_{intro} : \forall\alpha_0.\forall\widetilde{\alpha}.T}{\Gamma \mid \Delta \vdash M_{intro} : \forall\widetilde{\alpha}.T\{T_0'/\alpha_0\}}$$

  By definition, we have $\forall k, \mu_k(\pi) = \underline{\mu}_k(\tau)$. By renaming, α_0 and variables in $\widetilde{\alpha}$ are not free in T_0'. Take a sequence of type $\widetilde{T'}$ with same length as $\widetilde{\alpha}$, then $(T_0', \widetilde{T'})$ has the same length as $(\alpha_0, \widetilde{\alpha})$, thus we can conclude directly by induction hypothesis and using the fact that $\forall k, \mu_k(\pi) = \mu_k(\tau)$.

- **Introduction of $\forall$.** Suppose we have the proof

$$\pi\lhd \frac{\tau \lhd \Gamma \mid \Delta \vdash M_{intro} : \forall\widetilde{\alpha}.T \qquad \alpha_0 \text{ fresh in } \Gamma \text{ and } \Delta}{\Gamma \mid \Delta \vdash M_{intro} : \forall(\alpha_0, \widetilde{\alpha}).T}$$

  By definition, we have $\forall k, \mu_k(\pi) = \mu_k(\tau)$. Take a sequence of type $(T_0', \widetilde{T'})$. Let (**R**) denote the rule associated to M_{intro} and n its number of premises. By induction hypothesis, there exist type derivations $\pi_1', \ldots, \pi_n'$ such that if π' is the derivation obtained by applying the rule (**R**) to $\pi_1', \ldots, \pi_n'$, we have $\pi' \lhd \Gamma \mid \Delta \vdash M_{intro} : T\{\widetilde{T'}/\widetilde{\alpha}\}$ and $\forall k, \mu_k(\pi') = \mu_k(\pi)$. By Lemma 2.2.1, we can instantiate α_0 by T_0' in $\pi_1', \ldots, \pi_n'$ and we obtain proofs $\pi_1'', \ldots, \pi_n''$ such that, if we denote π'' the derivation obtained by applying the rule (**R**) to $\pi_1'', \ldots, \pi_n''$, we have $\pi'' \lhd \Gamma \mid \Delta \vdash M_{intro} : T\{(T_0', \widetilde{T'})/(\alpha_0, \widetilde{\alpha})\}$. Moreover, for all k, $\mu_k(\pi'') = \mu_k(\pi') = \mu_k(\pi)$.

□

We can now state the subject reduction of sEAL and we show that the measure allows us to construct a bound on the number of reductions.

Theorem 2.2.2. *Let $\tau \lhd \Gamma \mid \Delta \vdash M_0 : T$ and $M_0 \to M_1$. Let α be an integer equal or greater than the depth of τ. Then there is a proof $\tau' \lhd \Gamma \mid \Delta \vdash M_1 : T$ such that $\mathcal{R}ed(\mu_\alpha(\tau)+\widetilde{2}, \mu_\alpha(\tau')+\widetilde{2})$. Moreover, the depth of τ' is smaller than the depth of τ.*

Intuitively, with Lemma 2.2.9, for some terms, we can do as if the typing rules were syntax directed. Then, the proof uses the substitution lemmas (Lemma 2.2.2 and Lemma 2.2.4) for reductions in which substitution appears. For the others constructors, one can see that the measure given in the type system for sEAL is following this idea of the relation $\mathcal{R}ed$. For example, in $[\lambda x_n \ldots x_1.t](M_1, \ldots, M_{n-1}, \underline{v}) \to [\lambda x_{n-1} \ldots x_1.t\{\underline{v}/x_n\}](M_1, \ldots, M_{n-1})$, the degree that appears at position 0 is here to compensate the growing of the measure at position 1. Formally, the proof proceeds as follows.

Proof. To begin with, we show that we can consider that the first rule of τ is not an elimination or an introduction of quantification.

Lemma 2.2.10. *Let $\tau \lhd \Gamma \mid \Delta \vdash M_0 : T$ be a proof that does not start with an introduction or elimination of quantifier, and $M_0 \to M_1$. Let α be an integer equal or greater than the depth of τ. Then there is a proof $\tau' \lhd \Gamma \mid \Delta \vdash M_1 : T$ such that $\mathcal{R}ed(\mu_\alpha(\tau)+\widetilde{2}, \mu_\alpha(\tau')+\widetilde{2})$. Moreover, the depth of τ' is smaller than the depth of τ.*

Suppose that we proved Lemma 2.2.10. We can then prove Theorem 2.2.2 by induction on τ.

- If τ does not start with an introduction or elimination of quantification, by Lemma 2.2.10 we can conclude this case.
- If τ is:

$$\frac{\sigma \lhd \Gamma \mid \Delta \vdash M_0 : T \qquad \alpha \text{ fresh in } \Gamma, \Delta}{\Gamma \mid \Delta \vdash M_0 : \forall \alpha.T}$$

  with $\mu(\tau) = \mu(\sigma)$. By induction hypothesis with σ, there is a proof $\sigma' \lhd \Gamma \mid \Delta \vdash M_1 : T$ such that $\mathcal{R}ed(\mu_\alpha(\sigma)+\widetilde{2}, \mu_\alpha(\sigma')+\widetilde{2})$. Moreover, the depth of σ' is smaller than the depth of σ.

  We can construct the proof τ':

$$\frac{\sigma' \lhd \Gamma \mid \Delta \vdash M_1 : T \qquad \alpha \text{ fresh in } \Gamma, \Delta}{\Gamma \mid \Delta \vdash M_1 : \forall \alpha.T}$$

  And we have $\mu(\tau') = \mu(\sigma')$. We can conclude this case. The case of elimination of $\forall$ is similar.

Now, we prove Lemma 2.2.10. We first consider base reductions without contexts. With the generation lemma (Lemma 2.2.9), the case for the `if`-constructors are straightforward, it is a simple consequence of Lemma 2.2.7. We detail the other cases:

- If $M_0 = (\lambda x.M)M'$ and $M_1 = M\{M'/x\}$, we have a proof:

$$\tau \lhd \frac{\dfrac{\dfrac{\pi \lhd \Gamma_1, x : T' \mid \Delta \vdash M : T}{\Gamma_1 \mid \Delta \vdash \lambda x.M : T' \multimap T} \qquad \sigma \lhd \Gamma_2 \mid \Delta \vdash M' : T'}{\Gamma_1, \Gamma_2 \mid \Delta \vdash (\lambda x.M)M' : T}}{}$$

  The double line corresponds to the generation lemma (Lemma 2.2.9). We will use this notation everywhere in the proof.

$$\forall n \in \mathbb{N}, \mu_n(\tau) = \mu_n(\sigma)+\mu_n(\pi)+2 \cdot \mathbb{1}_0.$$

  The proof $\tau' \lhd \Gamma_1, \Gamma_2 \mid \Delta \vdash M\{M'/x\} : T$ is given by Lemma 2.2.2. As a consequence, we have:

$$\forall n \in \mathbb{N}, \mu_n(\tau') \leq \mu_n(\pi)+\mu_n(\sigma) \text{ so, } \forall n \in N, \mu_n(\tau') < \mu_n(\tau).$$

  Then, it is still true for $n = \alpha \geq \mathit{depth}(\tau)$ and the depth of τ' is smaller than the depth of τ. Moreover, by Lemma 2.2.7, we obtain directly that $\mathcal{R}ed(\mu_\alpha(\tau)+\widetilde{2}, \mu_\alpha(\tau')+\widetilde{2})$.

- If $M_0 = \texttt{let } !x = !M' \texttt{ in } M$ and $M_1 = M\{M'/x\}$ then we have a proof:

$$\tau \lhd \frac{\dfrac{\dfrac{\sigma \lhd \cdot \mid \Delta \vdash M' : T'}{\Gamma_1 \mid \Delta', [\Delta] \vdash !M' : !T'} \qquad \pi \lhd \Gamma_2 \mid \Delta', [\Delta], x : [T'] \vdash M : T}{\Gamma_1, \Gamma_2 \mid \Delta', [\Delta] \vdash \texttt{let } !x = !M' \texttt{ in } M : T}}{}$$

$$\forall n \in \mathbb{N}, \mu_n(\tau) = \mu_n(\pi)+(2, \mu_{n-1}(\sigma)).$$

By Lemma 2.2.4, we obtain a proof $\pi' \lhd \Gamma_2 \mid \Delta', [\Delta] \vdash M\{M'/x\} : T$, with:

$$\forall n \in \mathbb{N}, \mu_n(\pi') \leq (\omega_0(\pi), (\mu^1_n(\pi)+\omega_1(\pi) \cdot \mu_{n-1}(\sigma))).$$

By weakening we have $\tau' \lhd \Gamma_1, \Gamma_2 \mid \Delta', [\Delta] \vdash M\{M'/x\} : T$. By the precedent upper-bound, we obtain $depth(\tau') \leq depth(\tau)$. Moreover, $\omega_0(\tau) - \omega_0(\tau') \geq 2$, and so for $\alpha \geq depth(\tau) \geq 0$, we have $\mu_\alpha(\tau') <_{lex} \mu_\alpha(\tau)$. Finally, for $\alpha \geq j > 0$, we have:

$$\omega_j(\tau')+2 \leq \omega_j(\pi)+\omega_1(\pi) \cdot \omega_{j-1}(\sigma)+2.$$

$$\omega_j(\tau')+2 \leq (\omega_j(\pi)+\omega_{j-1}(\sigma)+2) \cdot (\omega_1(\pi)+\omega_0(\sigma)+2).$$

$$\omega_j(\tau')+2 \leq (\omega_j(\tau)+2) \cdot (\omega_1(\tau)+2).$$

And so we have indeed $\mathcal{R}ed(\mu_\alpha(\tau)+\widetilde{2}, \mu_\alpha(\tau')+\widetilde{2})$.

- If $M_0 = \texttt{let}\ x \otimes y = M \otimes M'\ \texttt{in}\ N$ and $M_1 = N\{M/x\}\{M'/y\}$, we have a proof:

$$\dfrac{\dfrac{\sigma \lhd \Gamma \mid \Delta \vdash M : T \qquad \sigma' \lhd \Gamma' \mid \Delta \vdash M' : T'}{\Gamma, \Gamma' \mid \Delta \vdash M \otimes M' : T \otimes T'}\tau\lhd \qquad \pi \lhd \Gamma'', x : T, y : T' \mid \Delta \vdash N : T''}{\Gamma, \Gamma', \Gamma'' \mid \Delta \vdash \texttt{let}\ x \otimes y = M \otimes M'\ \texttt{in}\ N : T''}$$

$$\forall n \in \mathbb{N}, \mu_n(\tau) = \mu_n(\pi)+\mu_n(\sigma)+\mu_n(\sigma')+2 \cdot \mathbb{1}_0.$$

Using twice Lemma 2.2.2, we obtain a proof $\tau' \lhd \Gamma, \Gamma', \Gamma'' \mid \Delta \vdash N\{M/x\}\{M'/y\} : T''$ with:

$$\forall n \in \mathbb{N}, \mu_n(\tau') \leq \mu_n(\pi)+\mu_n(\sigma)+\mu_n(\sigma') < \mu_n(\tau).$$

So $depth(\tau') \leq depth(\tau)$ and for $\alpha \geq depth(\tau)$, $\mu_\alpha(\tau') < \mu_\alpha(\tau)$. By Lemma 2.2.7, we have $\mathcal{R}ed(\mu_\alpha(\tau)+\widetilde{2}, \mu_\alpha(\tau')+\widetilde{2})$.

- If $M_0 = \texttt{iter}^!_{\mathsf{N}}(!M, !M')\ \underline{k}$ and $M_1 = !(M^k M')$, then we have a proof:

$$\dfrac{\dfrac{\dfrac{\sigma_1 \lhd \cdot \mid \Delta \vdash M : T \multimap T}{\Gamma_1 \mid \Delta', [\Delta] \vdash !M : !(T \multimap T)} \qquad \dfrac{\sigma_2 \lhd \cdot \mid \Delta \vdash M' : T}{\Gamma_2 \mid \Delta', [\Delta] \vdash !M' : !T}}{\Gamma_1, \Gamma_2 \mid \Delta', [\Delta] \vdash \texttt{iter}^!_{\mathsf{N}}(!M, !M') : \mathsf{N} \multimap !T}\tau\lhd \qquad \sigma \lhd \Gamma_3 \mid \Delta', [\Delta] \vdash \underline{k} : \mathsf{N}}{\Gamma_1, \Gamma_2, \Gamma_3 \mid \Delta', [\Delta] \vdash \texttt{iter}^!_{\mathsf{N}}(!M, !M')\underline{k} : !T}$$

$$\forall n \in \mathbb{N}, \mu_n(\tau) = \mu_n(\sigma)+(4, \mu_{n-1}(\sigma_1)+\mu_{n-1}(\sigma_2)).$$

Also note that $\forall n \in \mathbb{N}, \mu_n(\sigma) = (k+1) \cdot \mathbb{1}_1$. We can construct τ':

$$\dfrac{\dfrac{\sigma_1 \lhd \cdot \mid \Delta \vdash M : T \multimap T \qquad \cdot \mid \Delta \vdash M^{k-1} M' : T}{\cdot \mid \Delta \vdash M^k M' : T}\tau'\lhd}{\Gamma_1, \Gamma_2, \Gamma_3 \mid \Delta', [\Delta] \vdash !(M^k M') : !T}$$

$$\dfrac{\sigma_1 \lhd \cdot \mid \Delta \vdash M : T \multimap T \qquad \sigma_2 \lhd \cdot \mid \Delta \vdash M' : T}{\vdots}$$

$$\forall n \in \mathbb{N}, \mu_n(\tau') = k \cdot \mathbb{1}_1 + (1, k \cdot \mu_{n-1}(\sigma_1) + \mu_{n-1}(\sigma_2)).$$

We can see that $depth(\tau') \leq depth(\tau)$. Furthermore, $\omega_0(\tau) - \omega_0(\tau') \geq 2$, so for $\alpha \geq depth(\tau) \geq 0$, we have $\mu_\alpha(\tau') <_{lex} \mu_\alpha(\tau)$. We have $\omega_1(\tau')+2 \leq (\omega_1(\tau)+2)^2$, indeed:

$$k \cdot (1+\omega_0(\sigma_1))+\omega_0(\sigma_2)+2 \leq (k+1+\omega_0(\sigma_1)+\omega_0(\sigma_2)+2)^2.$$

For $1 < j \leq \alpha$, $\omega_j(\tau')+2 \leq (\omega_j(\tau)+2)(\omega_1(\tau)+2)$. Indeed:

$$k \cdot \omega_{j-1}(\sigma_1)+\omega_{j-1}(\sigma_2)+2 \leq (\omega_{j-1}(\sigma_1)+\omega_{j-1}(\sigma_2)+2)(k+1+\omega_0(\sigma_1)+\omega_0(\sigma_2)+2).$$

We can conclude $\mathcal{R}ed(\mu_\alpha(\tau)+\widetilde{2}, \mu_\alpha(\tau')+\widetilde{2})$. The proof for the rule $\mathtt{iter}^!_{\mathsf{W}}$ follows the same pattern.

- If $M_0 = [\lambda x_k \dots x_1.t](M'_1, \dots, M'_{k-1}, \underline{v})$ and $M_1 = [\lambda x_{k-1} \dots x_1.t\{\underline{v}/x_k\}](M'_1, \dots, M'_{k-1})$, then we have the following proof.

$$\tau\lhd \frac{\forall 1 \leq \ell \leq (k-1), \sigma_\ell \lhd \Gamma_\ell \mid \Delta \vdash M'_\ell : A_\ell \qquad \sigma \lhd \Gamma_k \mid \Delta \vdash \underline{v} : A_k \qquad \pi \lhd x_k : A_k^{(i_k)}, \dots, x_1 : A_1^{(i_1)} \vdash_{\mathsf{s}\ell\mathsf{T}} t : U}{\Gamma, \Gamma_1, \dots \Gamma_k \mid \Delta \vdash [\lambda x_k \dots x_1.t](M'_1, \dots, M'_{k-1}, \underline{v}) : \mathtt{type}(U)}$$

Note that, with the generation lemma (Lemma 2.2.9), the proof σ induces that $\underline{v}$ is either an actual integer $\underline{m}$, an actual word $\underline{w}$ or an actual boolean $\mathtt{tt}$ or $\mathtt{ff}$. Moreover, $\forall n \in \mathbb{N}, \mu_n(\sigma) = |\underline{v}| \cdot \mathbb{1}_1$ and

$$\forall n \in \mathbb{N}, \mu_n(\tau) = \sum_{\ell=1}^{k-1} \mu_n(\sigma_\ell)+|v| \cdot \mathbb{1}_1+k(d(\omega(\pi)+I)+1) \cdot \mathbb{1}_0+((\omega(\pi)+I)\{1/j_1\} \cdots \{1/j_l\}+1) \cdot \mathbb{1}_1$$

where $\mathtt{ind}(U) = I$ and $\{j_1, \dots, j_l\} = \mathtt{Var}(I) \cup \mathtt{Var}(\omega(\pi))$. From the proof π, we can construct by Lemma 2.1.3 a proof

$$\pi\{|\underline{v}|/i_k\} \lhd x_k : A_k^{(|\underline{v}|)}, x_{k-1} : A_{k-1}^{(i_{k-1})}, \dots, x_1 : A_1^{(i_1)} \vdash t : U\{|\underline{v}|/i_k\}.$$

Furthermore, we can construct a proof $\sigma' \lhd \cdot \vdash_{\mathsf{s}\ell\mathsf{T}} \underline{v} : A_k^{|\underline{v}|}$. By Lemma 2.1.8,

$$\pi' \lhd x_{k-1} : A_{k-1}^{(i_{k-1})}, \dots, x_1 : A_1^{(i_1)} \vdash t\{\underline{v}/x_k\} : U\{|\underline{v}|/i_k\}$$

and $\omega(\pi') \leq \omega(\pi)\{|\underline{v}|/i_k\}$. We can now construct the proof τ':

$$\tau'\lhd \frac{\forall 1 \leq \ell \leq (k-1), \sigma_\ell \lhd \Gamma_\ell \mid \Delta \vdash M'_\ell : A_\ell \qquad \pi' \lhd x_{k-1} : A_{k-1}^{(i_{k-1})}, \dots, x_1 : A_1^{(i_1)} \vdash_{\mathsf{s}\ell\mathsf{T}} t\{\underline{v}/x_k\} : U\{|\underline{v}|/i_k\}}{\Gamma, \Gamma_k, \Gamma_1, \dots \Gamma_{k-1} \mid \Delta \vdash [\lambda x_{k-1} \dots x_1.t\{\underline{v}/x_k\}](M'_1, \dots, M'_{k-1}) : \mathtt{type}(U)}$$

Let us denote $\{j'_1, \dots, j'_{l'}\} = \mathtt{Var}(I) \cup \mathtt{Var}(\omega(\pi)) \cup \mathtt{Var}(\omega(\pi'))$.

$$\forall n \in \mathbb{N}, \mu_n(\tau') = \sum_{\ell=1}^{k-1} \mu_n(\sigma_\ell)+(k-1)(d(\omega(\pi')+I\{|\underline{v}|/i_k\})+1) \cdot \mathbb{1}_0+((\omega(\pi')+I\{|\underline{v}|/i_k\})\{1/j'_1\} \cdots \{1/j'_{l'}\}+1) \cdot \mathbb{1}_1.$$

With this, we can first see that $depth(\tau') \leq depth(\tau)$. Moreover, by Theorem 2.1.2, from the inequality $\omega(\pi')+I\{|\underline{v}|/i_k\} \leq (I+\omega(\pi))\{|\underline{v}|/i_k\}$, we have:

$$d(\omega(\pi')+I\{|\underline{v}|/i_k\}) \leq d((I+\omega(\pi))\{|\underline{v}|/i_k\}) \leq d(I+\omega(\pi)).$$

$$\text{By Theorem 2.1.2, } (I{+}\omega(\pi))\{|\underline{v}|/i_k\} \leq |\underline{v}|^{d(I+\omega(\pi))} \cdot (I{+}\omega(\pi))\{1/i_k\}.$$

$$\text{So, } (I{+}\omega(\pi))\{|\underline{v}|/i_k\}\{1/j'_1,\ldots,j'_{l'}\} \leq |\underline{v}|^{d(I+\omega(\pi))} \cdot (I{+}\omega(\pi))\{1/j'_1,\ldots,j'_{l'}\}$$

by Lemma 2.1.3 (the substitution for i_k is either one of the j' by definition, or irrelevant if i_k does not appear in the indices). Now from those results, we have, $\forall n \in \mathbb{N}$:

$$\mu_n(\tau') \leq \sum_{\ell=1}^{k-1} \mu_n(\sigma_\ell){+}(k-1)(d(\omega(\pi){+}I){+}1)\cdot \mathbb{1}_0{+}(|\underline{v}|^{d(I+\omega(\pi))} \cdot (I{+}\omega(\pi))\{1/j'_1,\ldots,j'_{l'}\}{+}1)\mathbb{1}_1.$$

Now we can prove $\mathcal{R}ed(\mu_\alpha(\tau){+}\widetilde{2}, \mu_\alpha(\tau'){+}\widetilde{2})$:
By the precedent bound, $\omega_0(\tau) - \omega_0(\tau') \geq d(\omega(\pi){+}I){+}1$. Then:

$$\omega_1(\tau'){+}2 \leq \sum_{\ell=1}^{k-1} \omega_1(\sigma_\ell){+}|\underline{v}|^{d(I+\omega(\pi))} \cdot (I{+}\omega(\pi))\{1/j'_1,\ldots,j'_{l'}\}{+}3.$$

$$\omega_1(\tau'){+}2 \leq (\sum_{\ell=1}^{k-1} \omega_1(\sigma_\ell){+}|\underline{v}|{+}(\omega(\pi){+}I)\{1/j'_1,\ldots,j'_{l'}\}{+}3)^{d(\omega(\pi)+I)+1}.$$

$$\omega_1(\tau'){+}2 \leq (\omega_1(\tau){+}2)\cdot(\omega_1(\tau){+}2)^{d(\omega(\pi)+I)}.$$

And for $1 < j \leq \alpha$,

$$\omega_j(\tau'){+}2 \leq \sum_{\ell=1}^{k-1} \omega_j(\sigma_i){+}2 = \omega_j(\tau){+}2 \leq (\omega_j(\tau){+}2)(\omega_1(\tau){+}2)^{d(\omega(\pi)+I)}.$$

This proves $\mathcal{R}ed(\mu_\alpha(\tau){+}\widetilde{2}, \mu_\alpha(\tau'){+}\widetilde{2})$.

- If $M_0 = [t_0]()$ and $M_1 = [t_1]()$ with $t_0 \to t_1$ in $\mathsf{s}\ell\mathsf{T}$. We have a proof:

$$\tau\lhd \frac{\pi \lhd \cdot \vdash_{\mathsf{s}\ell\mathsf{T}} t_0 : U}{\Gamma \mid \Delta \vdash [t_0]() : \mathtt{type}(U)}$$

$$\forall n \in \mathbb{N}, \mu_n(\tau) = (1{+}(\omega(\pi){+}I)\{1/j_1,\ldots,j_l\})\cdot \mathbb{1}_1$$

where $\mathtt{ind}(U) = I$ and $\{j_1,\ldots,j_l\} = \mathtt{Var}(I) \cup \mathtt{Var}(\omega(\pi))$. By Theorem 2.1.1, the main theorem of $\mathsf{s}\ell\mathsf{T}$, we have a proof $\pi' \lhd \cdot \vdash_{\mathsf{s}\ell\mathsf{T}} t_1 : U$ with $\omega(\pi') < \omega(\pi)$. So we can construct the following proof.

$$\tau'\lhd \frac{\pi' \lhd \cdot \vdash_{\mathsf{s}\ell\mathsf{T}} t_1 : U}{\Gamma \mid \Delta \vdash [t_1]() : \mathtt{type}(U)}$$

Let us denote $\{j'_1,\ldots,j'_{l'}\}$ all the index variables in I, $\omega(\pi)$ and $\omega(\pi')$.

$\forall n \in \mathbb{N}, \mu_n(\tau') = (1{+}(\omega(\pi'){+}I)\{1/j'_1,\ldots,j'_{l'}\})\cdot \mathbb{1}_1$.

We directly see that the depth does not increase. Remark that the depth of τ is greater than 1 in this case.

We have by Lemma 2.1.3, $(\omega(\pi'){+}I)\{1/j'_1,\ldots,j'_{l'}\} < (\omega(\pi){+}I)\{1/j'_1,\ldots,j'_{l'}\}$.

And so, for $\alpha \geq \mathit{depth}(\tau) \geq 1, \mu_\alpha(\tau') < \mu_\alpha(\tau)$, and so we have $\mathcal{R}ed(\mu_\alpha(\tau){+}\widetilde{2}, \mu_\alpha(\tau'){+}\widetilde{2})$.

Remark that as opposed to all the precedent cases, $\mu_0(\tau)$ and $\mu_0(\tau')$ are equal, and so we need to look at position 1 to see that the measure strictly decreases. This remark is essential in the proof of Lemma 2.4.1.

- If $M_0 = [\underline{v}]()$ and $M_1 = \underline{v}$. The fact that M_0 can be typed by τ indicates that $\underline{v}$ is either an actual integer, a word or a boolean. With this remark, the typing τ' of M_1 is just the usual typing for those values. Moreover, we know the weight in $\mathsf{s}\ell\mathsf{T}$ and the measure in sEAL for the typing proof of a value, in $\mathsf{s}\ell\mathsf{T}$ the weight is 0 and in sEAL the measure is $|\underline{v}| \cdot \mathbb{1}_1$. Furthermore, if $\pi \lhd \cdot \vdash_{\mathsf{s}\ell\mathsf{T}} \underline{v} : U$, then we know that $|\underline{v}| \leq \mathtt{ind}(U)$. With this, we have $\mu_n(\tau) = (1{+}I\{1/\mathtt{Var}(I)\}) \cdot \mathbb{1}_1$ and $\mu_n(\tau') = |\underline{v}| \cdot \mathbb{1}_1$. By Lemma 2.1.3, we have $|\underline{v}| \leq I\{1/\mathtt{Var}(I)\}$ and so for $n \geq 1$, $\mu_n(\tau') < \mu_n(\tau)$. This gives us $\mathcal{R}ed(\mu_\alpha(\tau){+}\widetilde{2}, \mu_\alpha(\tau'){+}\widetilde{2})$. And the fact that the depth does not increase is direct.

  Remark that as the precedent case, we need to look at position 1 to see that the measure strictly decreases.

Now we need to work on the reductions under a context. For this we work by induction on contexts, and what we have done previously is the base case. For any inductive case of context except the ! case, the proof is straightforward, it is a direct application of the induction hypothesis.

When the context has the form $C = !C'$, the notion of depth is crucial. Indeed, suppose $M \to M'$, $M_0 = !M$ and $M_1 = !M'$. With the proof τ for M_0, we obtain a proof π for M, this gives us by induction hypothesis a proof π' for M', and this gives us a proof τ' for M_1. Moreover,

$$\forall n \in \mathbb{N}, \mu_n(\tau) = (1, \mu_{n-1}(\pi)) \text{ and } \mu_n(\tau') = (1, \mu_{n-1}(\pi')).$$

As $depth(\pi') \leq depth(\pi)$ we have:

$$depth(\tau') = depth(\pi'){+}1 \leq depth(\pi){+}1 = depth(\tau).$$

And for $\alpha \geq depth(\tau)$, then $(\alpha - 1) \geq depth(\pi)$. By induction hypothesis, we have

$$\mathcal{R}ed(\mu_{\alpha-1}(\pi){+}\widetilde{2}, \mu_{\alpha-1}(\pi'){+}\widetilde{2})$$

From this, we can easily deduce

$$\mathcal{R}ed(\mu_\alpha(\tau){+}\widetilde{2}, \mu_\alpha(\tau'){+}\widetilde{2})$$

Remark that this proof shows that if we had

$$\mathcal{R}ed(\mu_n(\pi){+}\widetilde{2}, \mu_n(\pi'){+}\widetilde{2})$$

we obtain

$$\mathcal{R}ed(\mu_{n+1}(\tau){+}\widetilde{2}, \mu_{n+1}(\tau'){+}\widetilde{2})$$

This remark is important for the proof of Lemma 2.4.1. □

From Theorem 2.2.2 and Theorem 2.2.1, we can easily deduce:

Theorem 2.2.3. *Let $\pi \lhd \Gamma \mid \Delta \vdash M : T$. Denote $\alpha = max(depth(\pi), 1)$, then $t_\alpha(\mu_\alpha(\pi){+}\widetilde{2})$ is a bound on the number of reductions from M.*

Remark 2.2.1. *As explained in the beginning of Section 2.2, we show informally that the proof of correctness in [83] is robust enough to support the addition of polynomial time functions in the type $\mathsf{N} \multimap \mathsf{N}$. This is a generic enrichment of EAL that does not describe the layer computing polynomial time function.*

We work in the classical EAL calculus described in Section 2.2. For any function f from integers to integers, we define a new constructor f in the classical EAL-calculus, and a new

reduction rule $f\ \underline{n} \to \underline{f(n)}$, saying that f applied to the Church encoding of the integer n is reduced to the Church encoding of the integer $f(n)$. We consider a cost to this reduction, depending on the integer n and the function f, that we call $C_f(n)$. We consider that this constructor f has type $\mathsf{N} \multimap \mathsf{N}$.

If this function f is a polynomial time computable function, we can bound the cost function $C_f(n)$ by a polynomial function $(n{+}2)^d$ for a certain d, and we can also bound the size of $f(n)$ by the cost, and so $f(n) \leq (n{+}2)^d$. The proof of correctness relies on a measure μ on terms, and as in the present work, this measure yields a bound by computing $t_\alpha(\mu)$ (see Section 2.2 or [83]).

Now if we look at the reduction rule, if we call $\mu(f)$ the measure for f, we go from the measure $\mu(f){+}(1, n{+}1)$ to $(0, (n{+}2)^d)$. In order to take in consideration the cost of the reduction, we add it in the measure. Thus, we consider that in the right part of the reduction, we have the measure $(0, 2(n{+}2)^d)$. If we define $\mu(f) = (d, 1)$, this reduction follows the relation $\mathcal{R}ed$ defined in Section 2.2. Thus, as in the present work, we can prove that EAL enriched with constructors for polynomial functions characterizes $2k$-EXP.

2.3 Examples of Programs for sEAL

Simple Examples in sEAL

We give some examples of terms in sEAL, first some terms we can usually see for the elementary affine logic, and then we give the term for computing tower of exponentials.

Some general results and notations on sEAL

- For base types A we have the coercion $A \multimap !A$. For example, for words, we have $coerc_w\ \underline{w} \to^*\ !\underline{w}$, with:

$$coerc_w = \mathtt{iter}^!_{\mathsf{W}}(!(\lambda w'.\mathtt{s}_0(w')), !(\lambda w'.\mathtt{s}_1(w')), !\epsilon).$$

- We write $\lambda x \otimes y.M$ for the term $\lambda z.\mathtt{let}\ x \otimes y = z\ \mathtt{in}\ M$.

Polynomials and Tower of Exponentials in sEAL Recall that we defined polynomials in $\mathsf{s}\ell\mathsf{T}$. With this we can define polynomials in sEAL with type $\mathsf{N} \multimap \mathsf{N}$ using the $\mathsf{s}\ell\mathsf{T}$ call. Moreover, using the iteration in sEAL, we can define a tower of exponentials.

We can compute the function $k \mapsto 2^{2^k}$ in sEAL with type $\mathsf{N} \multimap !\mathsf{N}$.

$$\dfrac{\dfrac{\dfrac{\dfrac{n : \mathsf{N} \mid \cdot \vdash n : \mathsf{N} \qquad \sigma \lhd x_1 : \mathsf{N}^{a_1} \vdash_{\mathsf{s}\ell\mathsf{T}} \mathtt{mult}\ x_1\ x_1 : \mathsf{N}^{a_1 \cdot a_1}}{n : \mathsf{N} \mid \cdot \vdash [\lambda x_1.\mathtt{mult}\ x_1\ x_1](n) : \mathsf{N}}}{\pi \lhd \cdot \mid \cdot \vdash \lambda n.[\lambda x_1.\mathtt{mult}\ x_1\ x_1](n) : \mathsf{N} \multimap \mathsf{N}}}{\cdot \mid \cdot \vdash\ !(\lambda n.[\lambda x_1.\mathtt{mult}\ x_1\ x_1](n)) :\ !(\mathsf{N} \multimap \mathsf{N})} \qquad \cdot \mid \cdot \vdash\ !\underline{2} :\ !\mathsf{N}}{\cdot \mid \cdot \vdash \mathtt{exp} = \mathtt{iter}^!_{\mathsf{N}}(!\lambda n.[\lambda x_1.\mathtt{mult}\ x_1\ x_1](n), !\underline{2}) : \mathsf{N} \multimap !\mathsf{N}}$$

$$\mathtt{iter}^!_{\mathsf{N}}(!\lambda n.[\lambda x_1.\mathtt{mult}\ x_1\ x_1](n), !\underline{2})\ \underline{k} \to^* !((\lambda n.[\lambda x_1.\mathtt{mult}\ x_1\ x_1](n))^k\ \underline{2}) \to^* !(2^{2^k}).$$

For an example of measure, for the subproof π, we have $depth(\pi) = 1$. From Section 2.1.1, we can deduce the weight for σ: $\omega(\sigma) = 4{+}6a_1{+}3a_1^3$. We can then deduce:

$$\mu(\pi) = (1{+}1{+}1 \cdot (d(\omega(\sigma){+}a_1 \cdot a_1){+}1), 1{+}(\omega(\sigma){+}a_1 \cdot a_1)\{1/a_1\}) = (6, 15).$$

If we define $2^x_0 = x$ and $2^x_{k+1} = 2^{2^x_k}$, with the use of polynomials, we can represent the function $n \mapsto 2^{P(n)}_{2k}$ for all $k \geq 0$ and polynomial P with a term of type $\mathsf{N} \multimap !^k\mathsf{N}$.

2.3.1 Testing Satisfiability of a Propositional Formula

We sketch here the construction of a term for deciding the SAT problem.

The term for SAT has type $\mathsf{N} \otimes \mathsf{W} \multimap !\mathsf{B}$ and given a formula in conjunctive normal form encoded in the type $\mathsf{N} \otimes \mathsf{W}$, it checks its satisfiability. The modality in front of the output $!\mathsf{B}$ shows that we used a non-polynomial computation, or more precisely an iteration in sEAL, as expected of a term for satisfiability.

We encode formulas in conjunctive normal form in the type $\mathsf{N} \otimes \mathsf{W}$, representing the number of distinct variables in the formula and the encoding of the formula by a word on the alphabet $\Sigma = \{0, 1, \#, |\}$. A literal is represented by the number of the corresponding variable written in binary and the first bit determines if the literal is positive or negative. Then # and | are used as separator for literals and clauses.

For example, the formula $(x_1 \vee x_0 \vee x_2) \wedge (x_3 \vee \overline{x_0} \vee \overline{x_1}) \wedge (\overline{x_2} \vee x_0 \vee \overline{x_3})$ could be represented by $\underline{4} \otimes \underline{|\#101\#100\#110|\#111\#000\#001|\#010\#100\#011}$.

Intermediate terms in sℓT

For the sake of simplicity, we sometimes omit to describe all terms in `ifw` or `iterw`, especially for the letters # and |, when they are not important. First, we can easily define a term $\mathtt{occ_a} : \mathsf{W}^I \multimap \mathsf{N}^I$ that gives the number of occurrences of $a \in \Sigma$ in a word. We can also describe a term that gives the n^{th} bit (from the right) of a binary word as a boolean $\mathtt{n^{th}} : \mathsf{W}^I \multimap \mathsf{N}^I \multimap \mathsf{B}$. And finally, we have a term $\mathtt{Extract_a} : \mathsf{W}^I \multimap \mathsf{W}^I \otimes \mathsf{W}^I$ that separates a word $w = w_0 a w_1$ in $w_0 \otimes w_1$ such that w_1 does not contain any a. This function will allow us to extract the last clause/literal of a word representing a formula.

A valuation is represented by a binary word with a length equal to the number of variables, such that the n^{th} bit of the word represents the boolean associated to the n^{th} variable. To begin with, we define the term $\mathtt{LittoBool} : \mathsf{N}^I \multimap \mathsf{W}^J \multimap \mathsf{W}^K \multimap \mathsf{B}$ such that given the number of variables, a valuation and the encoding of a literal, this term yields the boolean value that the valuation assigns to the literal. This term is described in Figure 2.10.

We then define a term $\mathtt{ClausetoBool} : \mathsf{N}^I \multimap \mathsf{W}^J \multimap \mathsf{W}^K \multimap \mathsf{B}$ such that, given the number of variables, a valuation and a word representing a clause, this term outputs the truth value of this clause using the valuation. The definition is given in Figure 2.10. With this we can check if a clause is true given a certain valuation. We can define in the same way a term for the truth value of a formula $\mathtt{FormulatoBool} : \mathsf{N}^I \multimap \mathsf{W}^J \multimap \mathsf{W}^K \multimap \mathsf{B}$. It is the same definition as `ClausetoBool`, where we replace "`or`" by "`and`" and "`LitttoBool`" by "`ClausetoBool`".

Testing all different valuations

Now that we have `FormulatoBool`, all we have to do is to test this term with all possible valuations. If n is the number of variables, all possible valuations are described by all the binary integers from 0 to $2^n - 1$. Thus, intuitively, given the number of variables n and the formula w_f, we want to compute:

$$\bigvee_{v=0}^{2^n-1} \mathtt{FormulatoBool}\ n\ v\ w_f.$$

In order to do this, we use the constructor for iteration defined in Section 2.1.1:

$$\mathtt{rec}(V, t)\ \underline{n} \to^*\ V\ \underline{n-1}\ (V\ \underline{n-2}\ (\dots\ (V\ \underline{0}\ t)\dots))$$

. We can then give the term for SAT as described in Figure 2.10.

$\text{LittoBool} \equiv \lambda n, v, l.\ \texttt{if}w(\ \lambda l'.\ \text{nth}\ v\ (\text{CBiToUn}\ n\ l')\ ,\lambda l'.\ \text{not}\ (\text{nth}\ v\ (\text{CBiToUn}\ n\ l'))\ ,\text{ff})\ l$

$\text{ClausetoBool} \equiv \lambda n, v, c.\ \texttt{let}\ w \otimes b = \text{itern}($
$\quad \lambda w' \otimes b'.\ \texttt{let}\ w_0 \otimes w_1 = \texttt{Extract}_{\#}\ w'\ \texttt{in}\ w_0 \otimes (\text{or}\ b'\ (\text{LitttoBool}\ n\ v\ w_1))$
$\quad , c \otimes \text{ff}$
$)\ (\texttt{occ}_{\#}\ c)\ \texttt{in}\ b$

$\text{SAT} \equiv \lambda n \otimes w.\ \texttt{let}\ !r = \ \text{iter}\,!n\ ($
$\quad !(\lambda n_0 \otimes n_1.\ \text{succ}(n_0) \otimes [\text{double}](n_1))$
$\quad , !(\texttt{0} \otimes 1)$
$)\ n\ \texttt{in let}\ !w_f = \text{coerc}\ w\ \texttt{in}\ !(\ \texttt{let}\ n \otimes exp = r\ \texttt{in}$
$\quad [\lambda n, exp, w_f.\ \texttt{rec}(\lambda v, b.\ \text{or}\ b\ (\text{FormulatoBool}\ n\ (\text{CUnToBi}\ n\ v)\ w_f), \text{ff})\ exp](n, exp, w_f))$

$\texttt{QBF}_0 \equiv \lambda v \otimes q \otimes f.\ \text{FormulatoBool}\ (\text{length}\ v)\ v\ f$

$\texttt{QBF}_1 \equiv \lambda v \otimes n \otimes exp \otimes q \otimes f.\ \texttt{rec}($
$\quad \lambda v', b.\ (\text{andor}\ q)\ b\ (\texttt{QBF}_0\ (\text{conc}\ v\ (\text{CUnToBi}\ n\ v'))) \otimes (\text{not}\ q) \otimes f)\ , q$
$)\ exp$

$\text{SubSum} \equiv \lambda w_S.\ \text{Riterw}($
$\quad \lambda w \otimes r.\ \texttt{let}\ w_0 \otimes w_1 = \texttt{Extract}_{|}\ w\ \texttt{in}\ w_0 \otimes r$
$\quad , \lambda w \otimes r.\ \texttt{let}\ w_0 \otimes w_1 = \texttt{Extract}_{|}\ w\ \texttt{in}\ w_0 \otimes (\texttt{Binaryadd}\ r\ w_1)$
$\quad , w_S \otimes \texttt{s}_0(\text{epsilon})$
$)$
$\text{SolvSubSum} \equiv \lambda k \otimes w_S.\ \texttt{let}\ !r = (exp\ [\texttt{occ}_{|}] \otimes \text{coerc})(w_S) \otimes (\text{coerc}\ k)\ \texttt{in}$
$\quad ![\lambda n, w, k.\ \texttt{rec}(\lambda v, b\ \text{or}\ b\ (\text{equal}\ k\ (\text{SubSum}\ w\ (\text{CUnToBi}\ (\texttt{occ}_{|}\ w)\ v))), \text{ff})\ n](r)$

Figure 2.10: Examples in sEAL.

The first iteration computes both 2^n and a copy of n. This technique is important as it shows that the linearity of sEAL for base variables is not too constraining for the iteration. Then, the second iteration in $\mathsf{s}\ell\mathsf{T}$ computes the big "or" given previously. Note that we need to be cautious about how to write integers, the number of variables n and exp (the integer representing 2^n) are given in unary, but we need the valuation in binary. And with that we have $\texttt{SAT} : \mathsf{N} \otimes \mathsf{W} \multimap !\mathsf{B}$.

Defining a $\mathsf{s}\ell\mathsf{T}$ term for QBF_k

Now we consider the following QBF_k problem, with k being a fixed non-negative integer. Take a formula:

$$Q_k x_n, x_{n-1}, \ldots, x_{i_{k-1}+1}.\ Q_{k-1} x_{i_{k-1}}, x_{i_{k-1}-1}, \ldots, x_{i_{k-2}+1} \ldots, Q_1 x_{i_1}, x_{i_1-1}, \ldots x_0.\phi.$$

The formula ϕ is a propositional formula in conjunctive normal form on the variables from x_0 to x_n, and $Q_i \in \{\forall, \exists\}$ are alternating quantifiers. That means that if Q_1 is $\forall$ then Q_2 must be $\exists$ and then Q_3 must be $\forall$ and so on. Here the variables are ordered for simplification. It can always be done by renaming. And now we have to answer if this formula is true. This can be solved in our enriched EAL calculus.

First, let us talk about the encoding of such a formula. With those ordered variables, a representation of such a formula can be a term of type $\mathsf{N}_k \otimes \mathsf{N}_{k-1} \otimes \cdots \otimes \mathsf{N}_1 \otimes \mathsf{B} \otimes \mathsf{W}$. For all

i with $1 \leq i \leq k$, N_i represents the number of variables between the quantifiers Q_i and Q_{i-1}. The boolean represents the quantifier Q_k, with the convention $\forall = \mathtt{tt}$. And finally, the formula ϕ is encoded in a word as previously. This is not a canonical representation of a formula, but for any good encoding of a QBF_k formula we should be able to extract this information with a $\mathsf{s}\ell\mathsf{T}$ term, so for simplification, we directly take this encoding. For example, with $k = 2$, the formula:

$$\forall x_3, x_2.\exists x_1, x_0.(x_1 \vee x_0 \vee x_2) \wedge (x_3 \vee \overline{x_0} \vee \overline{x_1}) \wedge (\overline{x_2} \vee x_0 \vee \overline{x_3})$$

is represented by:

$$\underline{2} \otimes \underline{2} \otimes \mathtt{tt} \otimes \underline{|\#101\#100\#110|\#111\#000\#001|\#010\#100\#011}.$$

Now we define by induction on k a $\mathsf{s}\ell\mathsf{T}$ term called $\mathtt{QBF}_k$ for k a non-negative integer. We give to this term the type:

$$\mathtt{QBF}_k : \mathsf{W}^{K_1} \otimes \mathsf{N}^{I_k} \otimes \mathsf{N}^{J_k} \otimes \cdots \otimes \mathsf{N}^{I_1} \otimes \mathsf{N}^{J_1} \otimes \mathsf{B} \otimes \mathsf{W}^{K_2} \multimap \mathsf{B}.$$

One can see a similitude with the representation of a SAT formula. But we add some arguments. First, the argument w_v of type W^{K_1} is a valuation on free variables of the QBF_k formula. Then we are given for each quantifier two integers n_i and exp_i of type N^{I_i} and N^{J_i}, with n_i being the number of variables between the quantifiers Q_i and Q_{i-1}, and $exp_i = 2^{n_i}$. Finally, the boolean represents the quantifier Q_k and W^{K_2} is a formula on variables from x_0 to $x_{n_1+\ldots+n_k+length(w_v)-1}$.

$\mathtt{QBF}_0$ has almost already been defined. See Figure 2.10 for the exact term.

Now, let us describe the term for $\mathtt{QBF}_1$. One can observe that it is close to the $\mathsf{s}\ell\mathsf{T}$ term $\mathtt{SAT}$. First, we have $\mathtt{andor} : \mathsf{B} \multimap \mathsf{B} \multimap \mathsf{B} = \mathtt{if}(\mathtt{and}, \mathtt{or})$. We also write $\mathtt{conc} : \mathsf{W}^I \multimap \mathsf{W}^J \multimap \mathsf{W}^{I+J}$ the term for concatenation of words. With that we can define $\mathtt{QBF}_1$ as explained in Figure 2.10. So, contrary to $\mathtt{SAT}$, we do not always do a big "or" on the results of $\mathtt{QBF}_0$ but we do either a big "and" if the quantifier Q_k is $\forall$, or a big "or" if the quantifier is $\exists$, as one can observe with the use of $\mathtt{andor}$. And when we call $\mathtt{QBF}_0$, we have to update the current valuation w_v and we have to alternate the quantifier. Now with this intuition, we can deduce the general term for $\mathtt{QBF}_{k+1}$ using $\mathtt{QBF}_k$, and then we can also deduce the sEAL term that just computes the arguments of the $\mathsf{s}\ell\mathsf{T}$ term $\mathtt{QBF}_k$ (with only one "!" as we only need to compute exponentials) and uses this function. And so, we obtain a term solving QBF_k with type $\mathsf{N}_k \otimes \mathsf{N}_{k-1} \otimes \cdots \otimes \mathsf{N}_1 \otimes \mathsf{B} \otimes \mathsf{W} \multimap !\mathsf{B}$.

2.3.2 Solving the Subset Sum Problem

We give here another example of solving an NP-Complete problem. Given a goal integer $k \in \mathbb{N}$ and a set S of integers, is there a subset $S' \subset S$ such that $\sum_{n \in S'} n = k$? We explain how we could solve this problem in our calculus. We represent the $SUBSET\ SUM$ problem by two words, k written as a binary integer and a word of the form $|\underline{n_1}|\underline{n_2}|\ldots|\underline{n_m}$, with the integers written in binary, representing the set S. In order to solve this problem, we can first define a $\mathsf{s}\ell\mathsf{T}$ term $\mathtt{equal} : \mathsf{W}^I \multimap \mathsf{W}^J \multimap \mathsf{B}$ that verifies if two binary integers are equal. Note that this is not exactly the equality on words because of the possible extra zeros at the beginning. Then, we can define a term $\mathtt{SubSum} : \mathsf{W}^I \multimap \mathsf{W}^J \multimap \mathsf{W}^{I \cdot J}$ such that, given the word w_S representing the set S and a binary word w_{sub} with a length equal to the cardinality of S, this term computes the sum of all the elements of the subset represented by w_{sub}, since this word can be seen as a function from S to $\{0, 1\}$. See Figure 2.10 for the term. We obtain a type $\mathsf{W}^{I \cdot J}$ for the output because we iterate at most J times a function for binary addition which can be given a type $\mathsf{W}^{a \cdot I} \multimap \mathsf{W}^I \multimap \mathsf{W}^{(a+1) \cdot I}$. Note that to define this function, we use $\mathtt{Extract}_|$ defined previously. Then, we can solve the $SUBSET\ SUM$ problem in the same way as SAT in the term $\mathtt{SolvSubSum}$. The notation $(f \otimes g)(x)$, when f and g are functions defined by iterators,

stands for the function defined by iteration on a couple, where the first projection will compute f and the second one g, such as before in $\mathtt{SAT}$. And so, we obtain a term of type $\mathsf{W} \otimes \mathsf{W} \multimap !\mathsf{B}$. We could also construct a term that gives us the subset corresponding to the goal, by changing the type in the iteration $\mathtt{rec}$ from $\mathsf{N}^a \multimap \mathsf{B} \multimap \mathsf{B}$ to $\mathsf{N}^a \multimap (\mathsf{B} \otimes \mathsf{W}^I) \multimap (\mathsf{B} \otimes \mathsf{W}^I)$, W^I being the type of the argument w.

2.4 Complexity Results: Characterization of 2k-EXP and 2k-FEXP

Now that we have proved Theorem 2.2.2, we have obtained a bound on the number of reduction steps from a term in sEAL. More precisely, this bound shows that between two consecutive weights ω_{i+1} and ω_i, there is a difference of 2 in the height of the tower of exponentials. This will allow us to give a characterization of the classes $2k$-EXP for $k \geq 0$, and each modality "!" in the type of a term will induce a difference of 2 in the height of the tower of exponentials. With exactly the same method, we also have a characterization of the classes $2k$-FEXP for $k \geq 0$.

Restricted Reductions and Values

First, we show that the previous bound on the number of reductions steps in Theorem 2.2.3 can be improved. Indeed, if we restrict the possible reductions, we obtain a more precise bound.

Definition 2.4.1 (Reductions up to a Certain Depth)**.** *For $i \in \mathbb{N}$, we define the i-reductions, that we note $\to_i$:*

- $\forall i \geq 1, [t]() \to_i [t']()$ *if* $t \to t'$ *in* $\mathsf{s}\ell\mathsf{T}$. *Moreover,* $[\underline{v}]() \to_i \underline{v}$.
- *For the other base reductions $M \to M'$, we have $\forall i \in \mathbb{N}, M \to_i M'$.*
- *For all $i \in \mathbb{N}$, if $M \to_i M'$ then $!M \to_{i+1} !M'$.*
- *For all others constructors for contexts, the index i stays the same. For example with the application, we have for all $i \in \mathbb{N}$, if $M \to_i M'$ then $M\ N \to_i M'\ N$.*

Now, we can find a more precise measure to bound the number of i-reductions.

Lemma 2.4.1. *Let $i \in \mathbb{N}$, $\tau \lhd \Gamma \mid \Delta \vdash M_0 : T$ and $M_0 \to_i M_1$. Then there is a proof $\tau' \lhd \Gamma \mid \Delta \vdash M_1 : T$ such that $\mathcal{R}ed(\mu_i(\tau)+\widetilde{2}, \mu_i(\tau')+\widetilde{2})$.*

The proof of this lemma is very similar to the proof of Theorem 2.2.2, the details are expressed in the proof of Theorem 2.2.2. We can then deduce the following theorem using previous results on the relation $\mathcal{R}ed$.

Theorem 2.4.1. *Let $\pi \lhd \Gamma \mid \Delta \vdash M : T$ and $\alpha = max(i, 1)$. Then $t_\alpha(\mu_i(\pi)+\widetilde{2})$ is a bound on the number of i-reductions from M.*

Let us now give an over approximation of the set of closed normal terms for i-reductions, that we call i-values.

Definition 2.4.2 (Values Associated to Restricted Reductions)**.** *We define for all $i \in \mathbb{N}$,* closed i-values V^i *by the following grammar.*

$$
\begin{aligned}
V^0 &:= M \\
\forall i \geq 1, V^i &:= \lambda x.M \mid !V^{i-1} \mid V_0^i \otimes V_1^i \mid \underline{0} \mid \mathtt{s}(V^i) \mid \mathtt{ifn}(V_0^i, V_1^i) \mid \mathtt{iter}^!_{\mathsf{N}}(V_0^i, V_1^i) \\
&\quad \mid \mathtt{tt} \mid \mathtt{ff} \mid \mathtt{if}(V_0^i, V_1^i) \mid \epsilon \mid \mathtt{s}_i(V^i) \mid \mathtt{ifw}(V_0^i, V_1^i, V_2^i) \mid \mathtt{iter}^!_{\mathsf{W}}(V_0^i, V_1^i, V_2^i)
\end{aligned}
$$

We can then prove the following lemma:

Lemma 2.4.2. *Let M be a term. If M is closed and has a typing derivation then, for all $i \in \mathbb{N}$, if M is normal for i-reductions then M is an i-value V^i.*

The proof is very similar to the one of Lemma 2.1.1. Note that we do not have the converse, an i-value is not a normal form for i-reductions. However, we do not need the converse to obtain the complexity results, we are only interested in base type closed i-values with $i \geq 1$. That is why 0-values and non-closed term, such as M in $\lambda x.M$, are so generic.

From the previous results, we now have that, from a typed term M, we can reach the normal form for i-reductions for M in less than $t_i(\mu_i(\pi)+\widetilde{2})$ reductions, and this normal form is an i-value.

A Characterization of $2k$-EXP

Now, we sketch how the type $!\mathsf{W} \multimap !^{k+1}\mathsf{B}$ can characterize the class $2k$-EXP for $k \geq 0$. Recall that 2^x_k is defined by $2^x_0 = x$ and $2^x_{k+1} = 2^{2^x_k}$. The class k-EXP is the class of problems solvable by a Turing machine that works in time $2^{p(n)}_k$ on an entry of size n, where p is a polynomial. First we show that the number of reductions for such a term is bounded by a tower of exponentials of height $2k$.

Lemma 2.4.3. *Let $\pi \lhd \cdot \mid \cdot \vdash t : !\mathsf{W} \multimap !^{k+1}\mathsf{B}$. Let w be a word of size $|w|$. We can compute the result of $t\ !\underline{w}$ in less than a $2k$-exponential tower in $|w|$.*

Proof. Observe that the result of this computation is of type $!^{k+1}\mathsf{B}$, and a $(k+2)$-value of type $!^{k+1}\mathsf{B}$ is exactly of the form $!^{k+1}\mathtt{tt}$ or $!^{k+1}\mathtt{ff}$. So it is enough to only consider $(k+2)$-reductions to compute the result, by Lemma 2.4.2. The measure μ_n of $t\ !\underline{w}$ is $\mu_n = \mu_n(\pi)+2\cdot\mathbb{1}_0+|\underline{w}|\cdot\mathbb{1}_2$. By Theorem 2.4.1, we can bound the number of reductions from $t\ !\underline{w}$ by $t_{k+2}(\mu_{k+2}+\widetilde{2})$. By definition, in $t_{k+2}(\mu_{k+2}+\widetilde{2})$, we can see that the weight at position 2, where the size of w appears, is at height $2k$. □

Now we have to prove that we can simulate a Turing-machine in our calculus. This proof is usual in implicit complexity [10, 8]. Formally, we want to show the following lemma.

Lemma 2.4.4. *Let k be an integer. Let TM be a Turing machine on binary words such that, for an input word w, TM works in time $2^{P(|w|)}_{2k}$, where P is a polynomial function. Then, TM can be simulated in sEAL by a term of type $!\mathsf{W} \multimap !^{k+1}\mathsf{B}$.*

The first thing we prove is the existence of a term in $\mathsf{s}\ell\mathsf{T}$ to simulate n steps of a deterministic Turing-machine on a word w. Suppose given two variables $w : \mathsf{W}^{i_w}$ and $n : \mathsf{N}^{i_n}$, we note Conf_j the type $\mathsf{W}^{i_w+j} \otimes \mathsf{B} \otimes \mathsf{W}^{i_w+j} \otimes \mathsf{B}^q$, with q an integer and B^q being q tensors of booleans. This type represents a configuration on a Turing machine after j steps, with B^q coding the state, and then $\underline{w_0} \otimes x \otimes \underline{w_1}$ represents the tape, with x being the head, w_0 represents the reverse of the word before j, and w_1 represents the word after x. We then define some terms in $\mathsf{s}\ell\mathsf{T}$ that work with this encoding. First we have a term $\mathtt{init}$ such that $w : \mathsf{W}^{i_w}, n : \mathsf{N}^{i_n} \vdash \mathtt{init} : \mathsf{Conf}_1$ and $\mathtt{init}$ computes the initial configuration of the Turing machine. Then, we have a term $\mathtt{step}$ with $\cdot \vdash \mathtt{step} : \mathsf{Conf}_\mathsf{j} \multimap \mathsf{Conf}_{\mathsf{j}+1}$ that computes the result of the transition function from a configuration to the next one, and finally we have a term $\mathtt{final}$ with $\cdot \vdash \mathtt{final} : \mathsf{Conf}_\mathsf{j} \multimap \mathsf{B}$ verifying if the final configuration is accepted or not. More precisely, this is given by:

- Given an initial state s of size q, we can code this state in a term $s : B^q$. Then we pose:

$$\mathtt{init} = \epsilon \otimes (\mathtt{ifw}(\lambda w'.\mathtt{ff} \otimes w', \lambda w'.\mathtt{tt} \otimes w', \mathtt{ff} \otimes \epsilon)\ w) \otimes s.$$

- Given for each state s of size q and boolean b a transition function of the shape $\delta(b,s) \subset \{left, right, stay\} \times \{0,1\} \times \{0,1\}^q$, we can construct the term `step`:

$$\texttt{step} = \lambda c.\texttt{let } x \otimes b \otimes y \otimes s = c \texttt{ in } (\texttt{case}_{q+1}(t_{0^{q+1}}, \ldots, t_{1^{q+1}}))\ (b \otimes s).$$

  The term `case` is a notation for a sequence of conditionals on the tensor of booleans of type $\mathsf{B}^{(q+1)}$, in this case $b \otimes s$. For a given boolean b and a state s, we define $t_{b\otimes s}$ according to $\delta(b,s)$. For example, if $\delta(b,s) = (left, b', s')$, we define:

$$t_{b\otimes s} = (\texttt{ifw}(\lambda w.w \otimes \texttt{ff}, \lambda w.w \otimes \texttt{tt}, \epsilon \otimes \texttt{ff})\ x) \otimes s_{b'}(y) \otimes s'.$$

- Given for states of size q a function $\texttt{accept} : \mathsf{B}^q \to \mathsf{B}$ (constructed with $\texttt{case}_q$), we can construct the term `final`:

$$\texttt{final} = \lambda c.\texttt{let } x \otimes b \otimes y \otimes s = c \texttt{ in } \texttt{accept}(s).$$

Now, suppose given a one-tape deterministic Turing machine TM on binary words such that for words w, TM works in time $2_{2k}^{P(|w|)}$. As usual, we suppose that TM has an infinite tape, this means that on an input w, the Turing-machine can read outside the bound of w and in this case, it reads a 0. We can compute a term in sEAL t_{TM} such that $\cdot \mid \cdot \vdash t_{TM} : !\mathsf{W} \multimap !^{k+1}\mathsf{B}$ and on an input $!w$, the term reduces to the term $!^{k+1}b$ with $b = \texttt{tt}$ if w is accepted by TM, and $b = \texttt{ff}$ otherwise. For this, we show how to decompose the work in order to construct this term.

1. We duplicate the word given in input.
2. With one of those words, we compute the length of the word, and we keep the other one as a copy.
3. Now that we have the length, we can compute $2_{2k}^{P(|w|)}$. So we obtain $!w \otimes !^{k+1}n$ with n representing $2_{2k}^{P(|w|)}$. By using the coercion, we obtain $!^{k+1}w \otimes !^{k+1}n$. We can then give this word and this integer as an input for a $\mathsf{s\ell T}$ program using the $\mathsf{s\ell T}$-call of sEAL.
4. In $\mathsf{s\ell T}$, by using the previously defined term `init` with the word $w : \mathsf{W}^{i_w}$ and the integer $n : \mathsf{N}^{i_n}$, we obtain a configuration C_1 of type $\mathsf{Conf_1}$ representing the initial tape of the Turing machine.
5. By iterating n times (using the constructor `itern`) the term $\cdot \vdash \texttt{step} : \mathsf{Conf_j} \multimap \mathsf{Conf_{j+1}}$, from C_1, we obtain a term of type $\mathsf{Conf_{i_n}}$. By definition, this term is a representation of the tape of the Turing machine after n steps, that is to say at the end of the computation.
6. Finally, with the term `final` we can extract the result of the computation as a boolean in $\mathsf{s\ell T}$.
7. As the word and the integer we used in the $\mathsf{s\ell T}$-call had the type $!^{k+1}\mathsf{W} \otimes !^{k+1}\mathsf{N}$, we obtain in sEAL the result of the computation as a boolean of type $!^{k+1}\mathsf{B}$.

In conclusion, we can simulate TM by a term of type $!\mathsf{W} \multimap !^{k+1}\mathsf{B}$.
With this, using Lemma 2.4.3, we obtain the following theorem.

Theorem 2.4.2. *Terms of type* $!\mathsf{W} \multimap !^{k+1}\mathsf{B}$ *characterize the class* $2k$*-EXP.*

As explained previously, this theorem can be expanded for the classes $2k$-FEXP, that is the class of functions from words to words that can be computed by a one-tape Turing machine with a running time bounded by $2_{2k}^{p(|w|)}$ on a word w. For more precise definitions of such classes, see [10]. This characterization uses the same proof, replacing $!\mathsf{W} \multimap !^{k+1}\mathsf{B}$ with $!\mathsf{W} \multimap !^{k+1}\mathsf{W}$.

Let us now briefly compare this result for sEAL with the situation for EAL. Recall that in EAL with recursive types, we can characterize k-EXP with the type $!\mathsf{W} \multimap !^{k+2}\mathsf{B}$ [8]. The difference can be explained by the fact that in EAL, in the type $\mathsf{N} \multimap \mathsf{N}$ we only have polynomials of degree 1 (polynomials in general have the type $!\mathsf{N} \multimap !\mathsf{N}$), whereas in our case, polynomials have the type $\mathsf{N} \multimap \mathsf{N}$.

To sum up, we showed in this chapter that from a type $W \multimap W$ with enough polynomial time functions, the bang modality from elementary linear logic leads to a characterization of $2k$-FEXPTIME in the type $!W \multimap !^{k+1}W$. We believe there should be more elegant presentation of this result, but the idea in itself of combining a rather expressive language with linear logic modalities can be of interest, since being able to characterizing a specific complexity class can lead to characterizations of other classes.

In the following, we will focus on complexity analysis of programs, instead of characterizations of complexity classes. So, not only we want some complexity bounds on programs, but we want those bounds to be as precise as possible. We thus define a framework, relying on sized types, where sizes are far more expressive that the one of $\mathsf{s}\ell\mathsf{T}$, in order to carry this complexity analysis.

Chapter 3

Complexity in a Functional Language with Sized Types

In this chapter, we present a simple functional language, with data-types and fixpoints, for which we give a sized type system for complexity analysis. This sized type system is inspired both by [6] and [29], and it relies mainly on polymorphic use of integer variables and explicit annotations for complexity and sizes in types. Moreover, the language we define uses an explicit `tick` constructor, which will be our abstraction of complexity: a `tick` accounts for one unit of time, and all other constructors are considered to take zero unit of time. This allows for a more flexible notion of complexity, and we can recover usual notions as a special case. For example, the number of reduction steps could be recovered by adding a `tick` after a function call or a pattern matching.

The goal of this chapter is to present this sized-type system first in the usual context of a functional language, with all the notations we use to write sizes. This type system will later be adapted for parallel complexity. Then, we present a proof of soundness: the type system gives a bound that is indeed greater than the number of `tick` that we can see during a reduction. The methodology for this proof will also be used throughout the thesis. First, we prove some structure lemmas on the type system, then some usual substitution lemmas, showing that variables are well-behaved. Finally, the main theorem is subject reduction, showing that typing is preserved by reduction, and that the complexity derived from a typing behaves accordingly.

3.1 A simple functional language with sizes

3.1.1 Syntax and Semantics

The language we consider for this chapter is a λ-calculus with data-types, pattern matching and fix point for functions. In order to account for complexity, we will also use a `tick` constructor to indicate which parts of the term should count for the reduction cost in a program. By a slight abuse of language, we call this language PCF, as the features are globally the same, even if the syntax does not respect the usual syntax of PCF.

Definition 3.1.1. *The sets of* terms *and* values *are given by the following grammars:*

$$\begin{aligned} M, N &:= x \mid \lambda x.M \mid M\ N \mid \langle M, N\rangle \mid \pi_1\ M \mid \pi_2\ M \mid \texttt{fix}\ f\ x\ =\ M \mid \texttt{tick}.M \\ &\quad \mid \underline{0} \mid \texttt{s}(M) \mid \texttt{match}\ M\ \{\underline{0} \mapsto N_1;;\ \texttt{s}(x) \mapsto N_2\} \\ V, W &:= x \mid \lambda x.M \mid \langle V, W\rangle \mid \underline{0} \mid \texttt{s}(V) \end{aligned}$$

$$\frac{}{(\lambda x.M)\ V \to_0 M[x := V]} \qquad \frac{}{\pi_i\langle V_1, V_2\rangle \to_0 V_i}$$

$$\frac{}{(\texttt{fix}\ f\ x\ =\ M)\ V \to_0 M[x := V][f := (\texttt{fix}\ f\ x\ =\ M)]} \qquad \frac{}{\texttt{tick}.M \to_1 M}$$

$$\frac{}{\texttt{match}\ \underline{0}\ \{\underline{0} \mapsto N_1;;\ \texttt{s}(x) \mapsto N_2\} \to_0 N_1}$$

$$\frac{}{\texttt{match}\ \texttt{s}(V)\ \{\underline{0} \mapsto N_1;;\ \texttt{s}(x) \mapsto N_2\} \to_0 N_2[x := V]} \qquad \frac{M \to_k N}{C[M] \to_k C[N]}$$

Figure 3.1: Semantics for PCF

So, we have standard lambda calculus, constructors for integers with a pattern-matching, pairs and projections, the `tick` constructor as the only source of complexity, and $\texttt{fix}\ f\ x\ =\ M$ which is a function defined recursively, so in M both x and f are free variables. We only have integers as data-types, but this is merely for the sake of simplicity, the type system could be extended to other data-types, as long as the pattern-matching rule is well-defined. See Remark 3.1.3 for more details.

As usual in a functional language, we may use the notation $\texttt{let}\ x = M\ \texttt{in}\ N$ to denote the term $(\lambda x.N)\ M$. We also use $(\lambda\langle x, y\rangle.N)$ to denote $(\lambda z.\texttt{let}\ x = \pi_1\ z\ \texttt{in let}\ y = \pi_2\ z\ \texttt{in}\ N)$, and similarly for $\texttt{fix}\ f\ \langle x, y\rangle\ =\ M$. Finally, we use $\underline{n}$ when n is an integer to denote $\texttt{s}(\texttt{s}(\cdots\texttt{s}(\underline{0})))$ with n calls to successor.

We then define a call-by-value semantics for this language. It is described by a set of base rules and a contextual rule. We also annotate rules by their complexity, where any standard rule has complexity 0 and removing a `tick` has complexity one. The rules are given in Figure 3.1. We use the notation $M[x := N]$ to denote the term M in which the free variable x has been replaced by N.

Definition 3.1.2. *The* contexts *are defined by the following grammar:*

$$C := [] \mid M\ C \mid C\ V \mid \langle C, M\rangle \mid \langle M, C\rangle \mid \pi_i\ C \mid \texttt{s}(C) \mid \texttt{match}\ C\ \{\underline{0} \mapsto N_1;;\ \texttt{s}(x) \mapsto N_2\}$$

The important points of this semantics is that first it is call-by-value, so for the complexity it means that if we have a `tick` in an argument, then it is reduced before being used as an argument. This avoids in particular duplication and erasure of a `tick` by a function call. For the function defined by fixpoint, we only unfold this fixpoint in an actual call to this function. This avoids unnecessary unfolding. In this sense, the fixpoint constructor in this language differs from the usual fixpoint operator in PCF. Still, it behaves as expected, as shown in the following example:

Example 3.1.1 (Reduction with Fixpoint)**.** *Consider the term* M :

$$\texttt{fix}\ f\ x\ =\ \texttt{tick.match}\ x\ \{\underline{0} \mapsto \underline{0};;\ \texttt{s}(y) \mapsto f\ y\}$$

Then:

$$\begin{aligned}&(M\ \underline{1}) \to_0 \texttt{tick.match}\ \underline{1}\ \{\underline{0} \mapsto \underline{0};;\ \texttt{s}(y) \mapsto M\ y\} \to_1 \texttt{match}\ \underline{1}\ \{\underline{0} \mapsto \underline{0};;\ \texttt{s}(y) \mapsto M\ y\}\\ \to_0&(M\ \underline{0}) \to_0 \texttt{tick.match}\ \underline{0}\ \{\underline{0} \mapsto \underline{0};;\ \texttt{s}(y) \mapsto M\ y\} \to_1 \texttt{match}\ \underline{0}\ \{\underline{0} \mapsto \underline{0};;\ \texttt{s}(y) \mapsto M\ y\}\\ \to_0&\underline{0}\end{aligned}$$

Then, we easily define the complexity of a term.

Definition 3.1.3 (Complexity of a Term)**.** *The* complexity *of a reduction $M \to_k^* N$, where $\to_k^*$ is the reflexive and transitive closure of $(\to_0 \cup \to_1)$ is defined as the sum of the cost through this reduction. Then, the complexity of a term M is defined as the maximal complexity over all reductions from M. This complexity could potentially be infinite.*

So on the previous example, the complexity is then 2, as expected. Let us look at another example.

Example 3.1.2 (Fibonacci)**.** *We can define the Fibonnaci function, where we count the number of calls, with the following terms:*

$$ADD := \texttt{fix}\ add\ \langle x, y\rangle\ =\ \texttt{match}\ x\ \{\underline{0} \mapsto y;;\ \texttt{s}(x') \mapsto \texttt{s}(add\ \langle x', y\rangle)\}$$

$$FIB := \texttt{fix}\ fib\ x\ =\ \texttt{tick.match}\ x\ \{\underline{0} \mapsto \underline{0};;\ \texttt{s}(x') \mapsto \texttt{match}\ x'\ \{\underline{0} \mapsto 1;;\ \texttt{s}(x'') \mapsto add\ \langle fib\ x', fib\ x''\rangle\}\}$$

Then, the complexity of $(FIB\ \underline{n})$ is given by $F(n)$ where

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

3.1.2 Sized Types

We now focus on a typing system for this functional language, such that we can extract a bound on the complexity of a program from a typing derivation. We give a presentation of sized types inspired by [6]. This type system will be close to the base we use for the π-calculus, and we present it to illustrate the interest of a sized type system. With regard to [6], there are some differences to notice. We choose a more generic form of arrow types, not well-suited for type inference but that generalizes the canonical types of [6]. Moreover, we annotate explicitly types with complexity, but we explain in Remark 3.1.2 that this is in fact not a big difference with regard to [6]. And finally, we choose a presentation of sizes adapted to this thesis, and inspired by [29].

As the previous type system described in Section 2.1, this system relies on the definition of indices to keep track of the size of values in a process.

Definition 3.1.4. *The set of* indices *for natural numbers is given by the following grammar.*

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

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

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

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

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

Definition 3.1.5 (Constraints on Indices)**.** *Let $\varphi \subset \mathcal{V}$ be a set of index variables. A* constraint *C on φ is an expression with the shape $I \bowtie J$ where I and J are indices with free variables in φ and $\bowtie$ denotes a binary relation on integers. Usually, we take $\bowtie \in \{\leq, <, =, \neq\}$. Finite set of constraints are denoted Φ.*

For a set $\varphi \subset \mathcal{V}$, we say that a valuation $\rho : \varphi \to \mathbb{N}$ *satisfies* a constraint $I \bowtie J$ on φ, noted $\rho \vDash I \bowtie J$ when $[\![I]\!]_\rho \bowtie [\![J]\!]_\rho$ holds. Similarly, $\rho \vDash \Phi$ holds when $\rho \vDash C$ for all $C \in \Phi$. Likewise, we note $\varphi; \Phi \vDash C$ when for all valuations ρ on φ such that $\rho \vDash \Phi$ we have $\rho \vDash C$. Notice that the order $\leq$ in a context $\varphi; \Phi$ is not total in general, for example $(i, j); \cdot \nvDash i \leq ij$ and $(i, j); \cdot \nvDash ij \leq i$.

Remark 3.1.1 (More Complex Indices)**.** *This definition of indices allows any usual function to be defined, but it does not allow some more complex indices that are used for example in [29]. For example, we cannot define*

$$\sum_{0 \leq i \leq I} J$$

where, if we call φ the set of index variables in I and J without i, and we have $\rho : \varphi \to \mathbb{N}$,

$$[\![\sum_{0 \leq i \leq I} J]\!]_\rho = \sum_{n=0}^{[\![I]\!]_\rho} [\![J]\!]_{\rho[i \mapsto n]}$$

A usual way to add this kind of indices is to explicitly put them in the definitions, or we could alternatively consider higher-order interpretation of indices, where for example the pair of this index J and the index variable i would be interpreted as the higher order function of type $\mathbb{N} \to \mathbb{N}$ defined by

$$[\![(J, i)]\!]_\rho(n) = [\![J]\!]_{\rho[i \mapsto n]}$$

For the sake of simplicity, we keep here the simple interpretation of functions.

We now define the sized type system by enriching usual base types with sizes, and by adding some polymorphism and complexity information in an arrow type.

Definition 3.1.6 (Types)**.** *The set of types is given by the following grammar:*

$$T, S := \mathsf{Nat}[I, J] \;\big|\; T \times S \;\big|\; \widetilde{\forall i}.T \Rightarrow_K S$$

Intuitively, an integer n of type $\mathsf{Nat}[I, J]$ satisfies $I \leq n \leq J$. A function of type $\widetilde{\forall i}.T \Rightarrow_K S$ is of type $T\{\widetilde{I}/\widetilde{i}\} \Rightarrow_{K\{\widetilde{I}/\widetilde{i}\}} S\{\widetilde{I}/\widetilde{i}\}$ for any sequence $\widetilde{I}$ of indices, and K is the complexity of a call to this function. The important point is that it can express a complexity which depends on the size, for example a linear time function from integer to integers that doubles the size of its input could be typed with $\forall i.\mathsf{Nat}[0, i] \Rightarrow_i \mathsf{Nat}[0, 2i]$. We will use $\mathsf{Nat}[I]$ to denote the singleton type $\mathsf{Nat}[I, I]$. We only consider well-defined intervals, so in order to use the interval $[I, J]$ in a context $\varphi; \Phi$ we must have $\varphi; \Phi \vDash I \leq J$.

Let us give some notations that will be used throughout the thesis before giving formal rules.

Definition 3.1.7 (Notations for Contexts)**.** *From now on, when we write (φ, i), it is always assumed that $i \notin \varphi$. So, for example in Figure 3.2, in the rule for arrow, it is implicit that $\widetilde{i}$ are fresh index variables. Note that this can always be done by renaming.*

Similarly, for a context $\Gamma = x_1 : T_1, \ldots, x_n : T_n$, when we write $\Gamma, x : T$, then it is assumed that x is not already in Γ.

However, for the set of constraints, we write $\Phi, I \bowtie J$ to denote the set $\Phi \cup \{I \bowtie J\}$, without any assumption on whether this union is disjoint.

With those types comes a simple notion of subtyping. Intuitively, subtyping allows to take a larger bound on the size of an input, and a smaller bound on an input. This is described by Figure 3.2. A subtyping judgement has the shape $\varphi; \Phi \vdash T \sqsubseteq T'$, where the free index variables

$$\frac{\varphi;\Phi \vDash I \geq I' \qquad \varphi;\Phi \vDash J \leq J'}{\varphi;\Phi \vdash \mathsf{Nat}[I,J] \sqsubseteq \mathsf{Nat}[I',J']} \qquad \frac{\varphi;\Phi \vdash T \sqsubseteq T' \qquad \varphi;\Phi \vdash S \sqsubseteq S'}{\varphi;\Phi \vdash T \times S \sqsubseteq T' \times S'}$$

$$\frac{(\varphi,\widetilde{i});\Phi \vdash T' \sqsubseteq T \qquad (\varphi,\widetilde{i});\Phi \vDash K \leq K' \qquad (\varphi,\widetilde{i});\Phi \vdash S \sqsubseteq S'}{\varphi;\Phi \vdash \forall\widetilde{i}.T \Rightarrow_K S \sqsubseteq \forall\widetilde{i}.T' \Rightarrow_{K'} S'}$$

Figure 3.2: Subtyping Rules for PCF with Sizes

of T, T', Φ should be included in φ. In this thesis, almost all relations depend on a context $\varphi;\Phi$, and φ always describes the set of usable free index variables, and Φ a set of constraints on φ. A way to understand the rule for subtyping is just to reason about set inclusion. The set of integers between I and J is included in the set of integers between I' and J' if $I \leq I'$ and $J \leq J'$. A function from T to S with complexity bounded by K is included in the set of function from a smaller set T', to a greater set S'. And if $K \leq K'$, then the complexity of this function is indeed bounded by K'.

We can extend this subtyping relation to contexts, so given $\Gamma = x_1 : T_1, \dots x_n : T_n$ and $\Gamma' = x_1 : T'_1, \dots, x_n : T'_n$, we say that $\varphi;\Phi \vdash \Gamma \sqsubseteq \Gamma'$ if $\varphi;\Phi \vdash T_i \sqsubseteq T'_i$ for all $1 \leq i \leq n$.

A typing judgement has the shape $\varphi;\Phi;\Gamma \vdash M : T \triangleleft K$, with the meaning that under the constraints Φ, then M has type T in the typing context Γ, and K is a bound on the complexity of M. The typing rules are detailed in Figure 3.3.

$$\frac{}{\varphi;\Phi;\Gamma, x : T \vdash x : T \triangleleft 0} \qquad \frac{(\varphi,\widetilde{i});\Phi;\Gamma, x : T \vdash M : S \triangleleft K}{\varphi;\Phi;\Gamma \vdash \lambda x.M : \forall\widetilde{i}.T \Rightarrow_K S \triangleleft 0}$$

$$\frac{\varphi;\Phi;\Gamma \vdash M : \forall\widetilde{i}.T \Rightarrow_K S \triangleleft K_1 \qquad \varphi;\Phi;\Gamma \vdash N : T\{\widetilde{I}/\widetilde{i}\} \triangleleft K_2}{\varphi;\Phi;\Gamma \vdash M\ N : S\{\widetilde{I}/\widetilde{i}\} \triangleleft K_1{+}K_2{+}K\{\widetilde{I}/\widetilde{i}\}}$$

$$\frac{\varphi;\Phi;\Gamma \vdash M : T \triangleleft K_1 \qquad \varphi;\Phi;\Gamma \vdash N : S \triangleleft K_2}{\varphi;\Phi;\Gamma \vdash \langle M, N\rangle : T \times S \triangleleft K_1 + K_2} \qquad \frac{\varphi;\Phi;\Gamma \vdash M : T_1 \times T_2 \triangleleft K}{\varphi;\Phi;\Gamma \vdash \pi_i\ M : T_i \triangleleft K}$$

$$\frac{(\varphi,\widetilde{i});\Phi;\Gamma, x : T, f : \forall\widetilde{i}.T \Rightarrow_K S \vdash M : S \triangleleft K}{\varphi;\Phi;\Gamma \vdash \texttt{fix}\ f\ x\ =\ M : \forall\widetilde{i}.T \Rightarrow_K S \triangleleft 0} \qquad \frac{\varphi;\Phi;\Gamma \vdash M : T \triangleleft K}{\varphi;\Phi;\Gamma \vdash \texttt{tick}.M : T \triangleleft K{+}1}$$

$$\frac{}{\varphi;\Phi;\Gamma \vdash \underline{0} : \mathsf{Nat}[0,0] \triangleleft 0} \qquad \frac{\varphi;\Phi;\Gamma \vdash M : \mathsf{Nat}[I,J] \triangleleft K}{\varphi;\Phi;\Gamma \vdash \mathtt{s}(M) : \mathsf{Nat}[I{+}1, J{+}1] \triangleleft K}$$

$$\frac{\varphi;\Phi;\Gamma \vdash M : \mathsf{Nat}[I,J] \triangleleft K' \qquad \varphi;(\Phi, I \leq 0);\Gamma \vdash N_1 : T \triangleleft K \qquad \varphi;(\Phi, J \geq 1);\Gamma, x : \mathsf{Nat}[I{-}1, J{-}1] \vdash N_2 : T \triangleleft K}{\varphi;\Phi;\Gamma \vdash \texttt{match}\ M\ \{\underline{0} \mapsto N_1;;\ \mathtt{s}(x) \mapsto N_2\} : T \triangleleft K'{+}K}$$

$$\frac{\varphi;\Phi;\Gamma' \vdash M : T' \triangleleft K' \qquad \varphi;\Phi \vdash \Gamma \sqsubseteq \Gamma' \qquad \varphi;\Phi \vdash T' \sqsubseteq T \qquad \varphi;\Phi \vDash K' \leq K}{\varphi;\Phi;\Gamma \vdash M : T \triangleleft K}$$

Figure 3.3: Sized Typing Rules for PCF

In this typing system, we observe that all values are given complexity 0, this is expected as a value is in normal form so has indeed complexity 0. The rule we should explain is the one for application. In order to call the function M, we first need to instantiate the variables in $\widetilde{i}$ by actual indices. Once this is done, the complexity of $M\ N$ is the complexity of reducing N to a value V_N (K_2), then reducing M to a value V_M (K_1) and finally the complexity of reducing V_M applied to V_N, which is given by the complexity of the arrow type.

Let us give an example of typing, with the Fibonacci function described in Example 3.1.2.

Example 3.1.3 (Typing of Fibonacci Function)**.** *The typing for ADD is given in Figure 3.4, and the one for FIB is described in Figure 3.5. In this example, and in the following ones, we*

$$\dfrac{\dfrac{(i,j); i \leq 0 \vDash j = i{+}j}{\varphi; i \leq 0; \Gamma \vdash y : \mathsf{Nat}[i{+}j] \lhd 0} \qquad \dfrac{\dfrac{\dfrac{\varphi; i \geq 1; \Gamma, x' : \mathsf{Nat}[i{-}1] \vdash \langle x', y\rangle : \mathsf{Nat}[i{-}1] \times \mathsf{Nat}[j] \lhd 0}{\varphi; i \geq 1; \Gamma, x' : \mathsf{Nat}[i{-}1] \vdash add\ \langle x', y\rangle : \mathsf{Nat}[(i{-}1){+}j] \lhd 0}}{\varphi; i \geq 1; \Gamma, x' : \mathsf{Nat}[i{-}1] \vdash add\ \langle x', y\rangle : \mathsf{Nat}[(i{+}j){-}1] \lhd 0}}{\varphi; i \geq 1; \Gamma, x' : \mathsf{Nat}[i{-}1] \vdash \mathtt{s}(add\ \langle x', y\rangle) : \mathsf{Nat}[i{+}j] \lhd 0}}{\dfrac{(i,j); \cdot; x : \mathsf{Nat}[i], y : \mathsf{Nat}[j], add : T_+ \vdash \mathtt{match}\ x\ \{\underline{0} \mapsto y; ;\ \mathtt{s}(x') \mapsto \mathtt{s}(add\ \langle x', y\rangle)\} : \mathsf{Nat}[i{+}j] \lhd 0}{\cdot; \cdot; \cdot \vdash \mathtt{fix}\ add\ \langle x, y\rangle\ =\ \mathtt{match}\ x\ \{\underline{0} \mapsto y; ;\ \mathtt{s}(x') \mapsto \mathtt{s}(add\ \langle x', y\rangle)\} : T_+ \lhd 0}}$$

Figure 3.4: Typing for ADD in PCF

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\varphi; \Phi; \Gamma \vdash fib\ x' : \mathsf{Nat}[Fib(i{-}1)] \lhd F(i{-}1) \qquad \varphi; \Phi; \Gamma \vdash fib\ x'' : \mathsf{Nat}[Fib(i{-}2)] \lhd F(i{-}2)}{\varphi; \Phi; \Gamma \vdash \langle fib\ x', fib\ x''\rangle : \mathsf{Nat}[Fib(i{-}1)] \times \mathsf{Nat}[Fib(i{-}2)] \lhd F(i{-}1){+}F(i{-}2)}}{\varphi; \Phi; x : \mathsf{Nat}[i], x' : \mathsf{Nat}[i{-}1], x'' : \mathsf{Nat}[i{-}2], fib : T_f \vdash ADD\ \langle fib\ x', fib\ x''\rangle : \mathsf{Nat}[Fib(i{-}1){+}Fib(i{-}2)] \lhd F(i{-}1){+}F(i)}}{i; (i \geq 1, i{-}1 \geq 1); x : \mathsf{Nat}[i], x' : \mathsf{Nat}[i{-}1], x'' : \mathsf{Nat}[i{-}2], fib : T_f \vdash ADD\ \langle fib\ x', fib\ x''\rangle : \mathsf{Nat}[Fib(i)] \lhd F(i){-}1}}{i; i \geq 1; x : \mathsf{Nat}[i], x' : \mathsf{Nat}[i{-}1], fib : T_f \vdash \mathtt{match}\ x'\ \{\underline{0} \mapsto 1; ;\ \mathtt{s}(x'') \mapsto \cdots\} : \mathsf{Nat}[Fib(i)] \lhd F(i){-}1}}{i; \cdot; x : \mathsf{Nat}[i], fib : T_f \vdash \mathtt{match}\ x\ \{\underline{0} \mapsto \underline{0}; ;\ \mathtt{s}(x') \mapsto \mathtt{match}\ x'\ \{\underline{0} \mapsto 1; ;\ \mathtt{s}(x'') \mapsto \cdots\}\} : \mathsf{Nat}[Fib(i)] \lhd F(i){-}1}}{i; \cdot; x : \mathsf{Nat}[i], fib : T_f \vdash \mathtt{tick.match}\ x\ \{\underline{0} \mapsto \underline{0}; ;\ \mathtt{s}(x') \mapsto \mathtt{match}\ x'\ \{\underline{0} \mapsto 1; ;\ \mathtt{s}(x'') \mapsto \cdots\}\} : \mathsf{Nat}[Fib(i)] \lhd F(i)}}{\cdot; \cdot; \cdot \vdash \mathtt{fix}\ fib\ x\ =\ \mathtt{tick.match}\ x\ \{\underline{0} \mapsto \underline{0}; ;\ \mathtt{s}(x') \mapsto \mathtt{match}\ x'\ \{\underline{0} \mapsto 1; ;\ \mathtt{s}(x'') \mapsto ADD\ \langle fib\ x', fib\ x''\rangle\}\} : T_f \lhd 0}$$

Figure 3.5: Typing for FIB in PCF

may omit some easy premises, we represent a subtyping rule by a double line, without precising the constraints when they are obvious, and we may automatically rename contexts in order to gain some space. For this example, we denote by T_+ and T_f the types

$$T_+ := \forall (i,j).\mathsf{Nat}[i] \times \mathsf{Nat}[j] \Rightarrow_0 \mathsf{Nat}[i{+}j] \qquad T_f := \forall i.\mathsf{Nat}[i] \Rightarrow_{F(i)} \mathsf{Nat}[Fib(i)]$$

where F is the function described in Example 3.1.2, and Fib is the index function representing the Fibonacci function.

Remark 3.1.2 (About Explicit Complexity in Types)**.** *Contrary to the sized types literature, we add explicitly complexity in types. One may think that this makes the type system too complicated, but we show informally that this is in fact equivalent to types with only sizes. Indeed, it is well-known that for an imperative language, complexity can just be computed as the size of a global variable that we increment through the computation. For a functional language, we can do the same thing with a simple state monad, where we choose* Nat *for the state space. This is in fact what is done in [6]. In our case, the* `tick` *constructor would then correspond to incrementing this state, and the complexity K in an arrow type corresponds to the difference on the size of this state between output and input, and so in this sense, complexity is just the size of one particular state that we follow through the computation.*

Remark 3.1.3 (Pattern Matching for Other Data-Types)**.** *We only presented the pattern matching for integers in this section. The pattern matching for words can easily be found as an extension of the one for integers, and the one for lists will be given formally in a later section, in Figure 4.11. If we want to consider algebraic-data types, then pattern matching will be more complex. For example, consider the case of trees. For a tree, we may want two informations about both size and depth, and so a type of the shape $\mathsf{Tr}[I_d, J_d][I_s, J_s]$, where intuitively, a tree of this type had a depth between I_d and J_d and a size between I_s and J_s. Then, the pattern matching rule would be, with $\varphi' = \varphi, i, i', j, j'$:*

$$\dfrac{\varphi'; (\Phi, i \leq j, i' \leq j', J_d \geq 1, Is \leq 1{+}i{+}i', 1{+}j{+}j' \leq J_s); \Gamma, x : \mathsf{Tr}[I_d{-}1, J_d{-}1][i,j], y : \mathsf{Tr}[I_d{-}1, J_d{-}1][i',j'] \vdash N_2 : T \lhd K}{\varphi; \Phi; \Gamma \vdash \mathtt{match}\ M\ \{\mathtt{Leaf} \mapsto N_1; ; \mathtt{Node}(x,y) \mapsto N_2\} : T \lhd K'{+}K}$$

with the two additional premises:

$$(1) \qquad \varphi; \Phi; \Gamma \vdash M : \mathsf{Tr}[I_d, J_d][I_s, J_s] \triangleleft K'$$

$$(2) \qquad \varphi; (\Phi, I_d \leq 0, Is \leq 0); \Gamma \vdash N_1 : T \triangleleft K$$

Even if the rule seems complex, the reasoning is natural: we do not know the size of the subtree, but we do know a bound on their sum, as we have a bound on the size of the initial tree. So, we take arbitrary bounds for the sizes of the subtree, and we add a constraint on the sum. Note that in the case where we impose $i' = j' = 0$, we have in the constraints $I_s \leq 1{+}i \leq 1{+}j \leq J_s$, which can be written equivalently with $(I_s{-}1 \leq i \leq j \leq J_s{-}1, J_s \geq 1)$, and we have the same constraints as depth. Also, we emphasize the fact that, by definition of φ, the index variables i, i', j, j' must not appear in T nor K, and so the type and the complexity cannot depend on the actual sizes of the subtrees as they are unknown.

3.1.3 Soundness by Subject Reduction

The fact that K is indeed a bound on the complexity of a term in a judgement should not be surprising given the type system. In this section, we give a proof method to show formally this complexity bound, and we will see that this methodology is reused in this thesis for all the complexity type systems. The idea is to prove soundness of the type system (K is indeed a bound on the complexity) with a subject reduction theorem, stating that typing is somewhat preserved through a reduction step.

Structure Lemmas

The first lemmas we usually show are the structure lemmas. Their proofs are often direct, but they are useful to simplify the soundness proof.

Lemma 3.1.1 (Weakening)**.** *Let φ, φ' be disjoint set of index variables, Φ be a set of constraints on φ, Φ' a set of constraints on (φ, φ'), and Γ and Γ' be contexts on disjoint set of variables. Then, we have:*

1. *If $\varphi; \Phi \vDash C$ then $(\varphi, \varphi'); (\Phi, \Phi') \vDash C$*
2. *If $\varphi; \Phi \vdash T \sqsubseteq S$ then $(\varphi, \varphi'); (\Phi, \Phi') \vdash T \sqsubseteq S$*
3. *If $\varphi; \Phi; \Gamma \vdash M : T \triangleleft K$ then $(\varphi, \varphi'); (\Phi, \Phi'); \Gamma, \Gamma' \vdash M : T \triangleleft K$*

The proof proceeds by induction, Point 1 is a direct consequence of the definition of $\varphi; \Phi \vDash C$, and then Point 2 is by induction on the subtyping rules, and Point 3 is by induction on the typing rules, using Point 2 when needed. Overall, there is nothing difficult with this proof. Then, we have the opposite of weakening.

Lemma 3.1.2 (Strengthening)**.** *Let φ be a set of index variables, Φ a set of constraints on φ and C a constraint on ϕ such that $\varphi; \Phi \vDash C$. Then, we have:*

1. *If $\varphi; (\Phi, C) \vDash C'$ then $\varphi; \Phi \vDash C'$*
2. *If $\varphi; (\Phi, C) \vdash T \sqsubseteq S$ then $\varphi; \Phi \vdash T \sqsubseteq S$*
3. *If $\varphi; (\Phi, C); \Gamma, \Gamma' \vdash M : T \triangleleft K$ and the variables in Γ' are not free in M, then $\varphi; \Phi; \Gamma \vdash M : T \triangleleft K$*

Again, the proofs are straightforward.

Substitution Lemmas

Then, we are interested in substitution lemmas. In this type system, we can substitute both index variables and terms variables, so this gives two substitution lemmas.

Lemma 3.1.3 (Index Substitution)**.** *Let φ be a set of index variables and $i \notin \varphi$. Let J be an index with free variables in φ. Then,*

1. $[\![I\{J/i\}]\!]_\rho = [\![I]\!]_{\rho[i \mapsto [\![J]\!]_\rho]}$.
2. *If* $(\varphi, i); \Phi \vDash C$ *then* $\varphi; \Phi\{J/i\} \vDash C\{J/i\}$.
3. *If* $(\varphi, i); \Phi \vdash T \sqsubseteq U$ *then* $\varphi; \Phi\{J/i\} \vdash T\{J/i\} \sqsubseteq U\{J/i\}$.
4. *If* $(\varphi, i); \Phi; \Gamma \vdash M : T \lhd K$ *then* $\varphi; \Phi\{J/i\}; \Gamma\{J/i\} \vdash M : T\{J/i\} \lhd K\{J/i\}$.

Again, this lemma is proved by successive inductions. Note that from the previous lemmas, we can easily deduce the following lemma, that may be useful:

Lemma 3.1.4 (Name Preserving Index Substitution)**.** *Let φ be a set of index variables with $i \in \varphi$, and J an index with free variables in φ. If $\varphi; \Phi; \Gamma \vdash M : T \lhd K$ then $\varphi; \Phi\{J/i\}; \Gamma\{J/i\} \vdash M : T\{J/i\} \lhd K\{J/i\}$.*

Proof. The proof uses renaming and weakening. Let us call $\varphi' = \varphi/\{i\}$. So, from $(\varphi', i); \Phi; \Gamma \vdash M : T \lhd K$, we have, by weakening (Lemma 3.1.1), $(\varphi', i, i'); \Phi; \Gamma \vdash M : T \lhd K$. Let us call J' equal to J but the index variable i is renamed with i'. Then, we have, by index substitution, $(\varphi', i'); \Phi\{J'/i\}; \Gamma\{J'/i\} \vdash M : T\{J'/i\} \lhd K\{J'/i\}$. And then, again by renaming i' by i, we obtain $(\varphi', i); \Phi\{J/i\}; \Gamma\{J/i\} \vdash M : T\{J/i\} \lhd K\{J/i\}$. □

Then, we have substitution for terms, or more precisely for values since the reduction rules only substitute terms variables by values:

Lemma 3.1.5 (Value Substitution)**.** *If $\varphi; \Phi; \Gamma, x : S \vdash M : T \lhd K$ and $\varphi; \Phi; \Gamma \vdash V : S \lhd K'$, then $\varphi; \Phi; \Gamma \vdash M[x := V] : T \lhd K$.*

Proof. The important point for this lemma, is that K' is not used in the final derivation, this is because we can prove the following lemma:

Lemma 3.1.6 (Complexity of Values)**.** *If $\varphi; \Phi; \Gamma \vdash V : S \lhd K'$ then there is a typing $\varphi; \Phi; \Gamma \vdash V : S \lhd 0$*

This can easily be proved by induction on the typing derivation: the only rule that can increase the complexity of a value is the subtyping rule.

Once this lemma is proved, the proof of Lemma 3.1.5 is done by induction on the typing derivation for M. The main point is that the axiom rule

$$\frac{}{\varphi; \Phi; \Gamma, x : S \vdash x : S \lhd 0}$$

can then directly be replaced by the derivation for $\varphi; \Phi; \Gamma \vdash V : S \lhd 0$. □

Subject Reduction

Finally, we prove subject reduction, and then we will directly obtain the soundness theorem.

Theorem 3.1.1 (Subject Reduction)**.** *If $\varphi; \Phi; \Gamma \vdash M : T \lhd K$ then:*

1. *If $M \to_0 N$, we have $\varphi; \Phi; \Gamma \vdash N : T \lhd K$*
2. *If $M \to_1 N$, we have $\varphi; \Phi; \Gamma \vdash N : T \lhd K'$ with $\varphi; \Phi \vDash K'+1 \leq K$.*

Proof. We proceed by induction on $M \to_k N$. The first thing to notice is that the only non syntax-directed rule of the type system is the subtyping rule, and we can always suppose that there is exactly one subtyping rule between each other rules, by transitivity and reflexivity of subtyping. Moreover, if the typing $\varphi; \Phi; \Gamma \vdash M : T \lhd K$ starts with a subtyping rule and $M \to_0 N$, then we can mimic exactly the same rule for N, thus we can ignore this first subtyping rule in this case, as in Lemma 2.2.10. So, formally, we will prove the following lemma:

Lemma 3.1.7 (Subject Reduction)**.** *If π is a derivation of $\varphi; \Phi; \Gamma \vdash M : T \lhd K$ then:*

1. *If $M \to_0 N$, and the bottom rule of π is not a subtyping rule, then, we have $\varphi; \Phi; \Gamma \vdash N : T \lhd K$*
2. *If $M \to_1 N$, we have $\varphi; \Phi; \Gamma \vdash N : T \lhd K'$ with $\varphi; \Phi \vDash K'+1 \leq K$.*

And, from this lemma, we can easily obtain Theorem 3.1.1. Indeed, given $M \to_0 N$ and a derivation π of $\varphi; \Phi; \Gamma \vdash M : T \lhd K$, with the shape

$$\frac{\varphi; \Phi; \Gamma' \vdash M : T' \lhd K' \qquad \varphi; \Phi \vdash \Gamma \sqsubseteq \Gamma' \qquad \varphi; \Phi \vdash T' \sqsubseteq T \qquad \varphi; \Phi \vDash K' \leq K}{\varphi; \Phi; \Gamma \vdash M : T \lhd K}$$

Then, as $\varphi; \Phi; \Gamma' \vdash M : T' \lhd K'$ does not start with a subtyping rule, we can apply Lemma 3.1.7 and obtain $\varphi; \Phi; \Gamma' \vdash N : T' \lhd K'$. Thus, we can give the proof:

$$\frac{\varphi; \Phi; \Gamma' \vdash N : T' \lhd K' \qquad \varphi; \Phi \vdash \Gamma \sqsubseteq \Gamma' \qquad \varphi; \Phi \vdash T' \sqsubseteq T \qquad \varphi; \Phi \vDash K' \leq K}{\varphi; \Phi; \Gamma \vdash N : T \lhd K}$$

And this derivation has the same conclusion as the original one, so this concludes Theorem 3.1.1. So, let us know show Lemma 3.1.7.

- **Case** $(\lambda x.M)\ V \to_0 M[x := V]$. Then, the typing has the shape:

$$\dfrac{\dfrac{\dfrac{\dfrac{\Pi_M}{(\varphi, \widetilde{i}); \Phi; \Delta, x : T' \vdash M : S' \lhd K'}}{\varphi; \Phi; \Delta \vdash (\lambda x.M) : \forall \widetilde{i}.T' \Rightarrow_{K'} S' \lhd 0} \quad (1)}{\varphi; \Phi; \Gamma \vdash (\lambda x.M) : \forall \widetilde{i}.T \Rightarrow_K S \lhd K_1} \qquad \dfrac{\Pi_V}{\varphi; \Phi; \Gamma \vdash V : T\{\widetilde{I}/\widetilde{i}\} \lhd K_2}}{\varphi; \Phi; \Gamma \vdash (\lambda x.M)\ V : S\{\widetilde{I}/\widetilde{i}\} \lhd K_1 + K_2 + K\{\widetilde{I}/\widetilde{i}\}}$$

  where (1) is

$$\varphi; \Phi \vdash \Gamma \sqsubseteq \Delta \qquad \varphi; \Phi \vdash \forall \widetilde{i}.T' \Rightarrow_{K'} S' \sqsubseteq \forall \widetilde{i}.T \Rightarrow_K S$$

  The subtyping condition on the arrow type gives us

$$(\varphi; \widetilde{i}); \Phi \vdash T \sqsubseteq T' \qquad (\varphi; \widetilde{i}); \Phi \vDash K' \leq K \qquad (\varphi, \widetilde{i}); \Phi \vdash S' \sqsubseteq S$$

so, using the subtyping rule, we can obtain from Π_M a derivation of $(\varphi, \widetilde{i}); \Phi; \Gamma, x : T \vdash M : S \triangleleft K$. Then, by the index substitution lemma (Lemma 3.1.3), we obtain a derivation of

$$\varphi; \Phi\{\widetilde{I}/\widetilde{i}\}; \Gamma\{\widetilde{I}/\widetilde{i}\}, x : T\{\widetilde{I}/\widetilde{i}\} \vdash M : S\{\widetilde{I}/\widetilde{i}\} \triangleleft K\{\widetilde{I}/\widetilde{i}\}$$

If we look where the index variables in $\widetilde{i}$ can be free, we can refine this judgement by

$$\varphi; \Phi; \Gamma, x : T\{\widetilde{I}/\widetilde{i}\} \vdash M : S\{\widetilde{I}/\widetilde{i}\} \triangleleft K\{\widetilde{I}/\widetilde{i}\}$$

We can then use the value substitution lemma (Lemma 3.1.5) with Π_V to obtain

$$\varphi; \Phi; \Gamma \vdash M[x := V] : S\{\widetilde{I}/\widetilde{i}\} \triangleleft K\{\widetilde{I}/\widetilde{i}\}$$

We can then conclude this case by subtyping.

- **Case** $\pi_i\ \langle V_1, V_2\rangle \to_0 V_i$. Then, the typing has the shape:

$$\dfrac{\dfrac{\dfrac{\dfrac{\Pi_1}{\varphi; \Phi; \Delta \vdash V_1 : T_1' \triangleleft K_1} \quad \dfrac{\Pi_2}{\varphi; \Phi; \Delta \vdash V_2 : T_2' \triangleleft K_2}}{\varphi; \Phi; \Delta \vdash \langle V_1, V_2\rangle : T_1' \times T_2' \triangleleft K_1{+}K_2} \quad (1)}{\varphi; \Phi; \Gamma \vdash \langle V_1, V_2\rangle : T_1 \times T_2 \triangleleft K}}{\varphi; \Phi; \Gamma \vdash \pi_i\ \langle V_1, V_2\rangle : T_i \triangleleft K}$$

where (1) is

$$\varphi; \Phi \vdash \Gamma \sqsubseteq \Delta \qquad \varphi; \Phi \vdash T_1' \times T_2' \sqsubseteq T_2 \times T_2 \qquad \varphi; \Phi \vDash K_1{+}K_2 \leq K$$

With a subtyping rule, we obtain immediately for both $i = 1$ and $i = 2$ that $\varphi; \Phi; \Gamma \vdash V_i : T_i \triangleleft K$, as $K_i \leq K_1{+}K_2 \leq K$. This concludes this case.

- **Case** $(\texttt{fix}\ f\ x\ =\ M)\ V \to_0 M[x := V][f := (\texttt{fix}\ f\ x\ =\ M)]$. Then the typing has the shape:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Pi_M}{(\varphi, \widetilde{i}); \Phi; \Delta, x : T', f : \forall\widetilde{i}.T' \Rightarrow_{K'} S' \vdash M : S' \triangleleft K'}}{\varphi; \Phi; \Delta \vdash \texttt{fix}\ f\ x\ =\ M : \forall\widetilde{i}.T' \Rightarrow_{K'} S' \triangleleft 0} \quad (1)}{\varphi; \Phi; \Gamma \vdash \texttt{fix}\ f\ x\ =\ M : \forall\widetilde{i}.T \Rightarrow_K S \triangleleft K_1} \quad \dfrac{\Pi_V}{\varphi; \Phi; \Gamma \vdash V : T\{\widetilde{I}/\widetilde{i}\} \triangleleft K_2}}{\varphi; \Phi; \Gamma \vdash (\texttt{fix}\ f\ x\ =\ M)\ V : S\{\widetilde{I}/\widetilde{i}\} \triangleleft K_1 + K_2 + K\{\widetilde{I}/\widetilde{i}\}}$$

where (1) is

$$\varphi; \Phi \vdash \Gamma \sqsubseteq \Delta \qquad \varphi; \Phi \vdash \forall\widetilde{i}.T' \Rightarrow_{K'} S' \sqsubseteq \forall\widetilde{i}.T \Rightarrow_K S$$

.

First, we have a derivation of $(\varphi, \widetilde{i}); \Phi; \Delta, x : T', f : \forall\widetilde{i}.T' \Rightarrow_{K'} S' \vdash M : S' \triangleleft K'$ and a derivation of $\varphi; \Phi; \Delta \vdash \texttt{fix}\ f\ x\ =\ M : \forall\widetilde{i}.T' \Rightarrow_{K'} S' \triangleleft 0$, so by Lemma 3.1.5, we obtain a derivation of

$$(\varphi, \widetilde{i}); \Phi; \Delta, x : T' \vdash M[f := (\texttt{fix}\ f\ x\ =\ M)] : S' \triangleleft K'$$

The subtyping condition on the arrow type gives us

$$(\varphi; \widetilde{i}); \Phi \vdash T \sqsubseteq T' \qquad (\varphi; \widetilde{i}); \Phi \vDash K' \leq K \qquad (\varphi; \widetilde{i}); \Phi \vdash S' \sqsubseteq S$$

so, using the subtyping rule, we can obtain a derivation of

$$(\varphi, \widetilde{i}); \Phi; \Gamma, x : T \vdash M[f := (\texttt{fix}\ f\ x\ =\ M)] : S \lhd K$$

Then, by the index substitution lemma (Lemma 3.1.3), we obtain a derivation of

$$\varphi; \Phi; \Gamma, x : T\{\widetilde{I}/\widetilde{i}\} \vdash M[f := (\texttt{fix}\ f\ x\ =\ M)] : S\{\widetilde{I}/\widetilde{i}\} \lhd K\{\widetilde{I}/\widetilde{i}\}$$

We then use the value substitution lemma (Lemma 3.1.5) with Π_V to obtain

$$\varphi; \Phi; \Gamma \vdash M[f := (\texttt{fix}\ f\ x\ =\ M)][x := V] : S\{\widetilde{I}/\widetilde{i}\} \lhd K\{\widetilde{I}/\widetilde{i}\}$$

We can then conclude this case by subtyping.

- **Case** $\texttt{tick}.M \to_1 M$. Then, the typing has the shape:

$$\dfrac{\dfrac{\dfrac{\Pi_M}{\varphi; \Phi; \Delta \vdash M : T' \lhd K'}}{\varphi; \Phi; \Delta \vdash \texttt{tick}.M : T' \lhd K'{+}1} \qquad \varphi; \Phi \vdash \Gamma \sqsubseteq \Delta \qquad \varphi; \Phi \vdash T' \sqsubseteq T \qquad \varphi; \Phi \vDash K'{+}1 \leq K}{\varphi; \Phi; \Gamma \vdash \texttt{tick}.M : T \lhd K}$$

Then, with subtyping from Π_M we can obtain $\varphi; \Phi; \Gamma \vdash M : T \lhd K'$ and this concludes this case.

- **Case** $\texttt{match}\ \underline{0}\ \{\underline{0} \mapsto N_1;;\ \texttt{s}(x) \mapsto N_2\} \to_0 N_1$. Then, the typing has the shape:

$$\dfrac{\dfrac{\dfrac{}{\varphi; \Phi; \Delta \vdash \underline{0} : \mathsf{Nat}[0,0] \lhd 0} \qquad \varphi; \Phi \vdash \Gamma \sqsubseteq \Delta; \mathsf{Nat}[0,0] \sqsubseteq \mathsf{Nat}[I,J]}{\varphi; \Phi; \Gamma \vdash \underline{0} : \mathsf{Nat}[I,J] \lhd K'} \qquad \dfrac{\Pi_1}{\varphi; (\Phi, I \leq 0); \Gamma \vdash N_1 : T \lhd K} \qquad \cdots}{\varphi; \Phi; \Gamma \vdash \texttt{match}\ \underline{0}\ \{\underline{0} \mapsto N_1;;\ \texttt{s}(x) \mapsto N_2\} : T \lhd K'{+}K}$$

By definition of subtyping, we obtain directly that $\varphi; \Phi \vDash I \leq 0$. So, by Lemma 4.5.9 on Π_1, we obtain $\varphi; \Phi; \Gamma \vdash N_1 : T \lhd K$ and this concludes this case.

- **Case** $\texttt{match}\ \texttt{s}(V)\ \{\underline{0} \mapsto N_1;;\ \texttt{s}(x) \mapsto N_2\} \to_0 N_2[x := V]$. Then, the typing has the shape:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Pi_V}{\varphi; \Phi; \Delta \vdash V : \mathsf{Nat}[I', J'] \lhd K''}}{\varphi; \Phi; \Delta \vdash \texttt{s}(V) : \mathsf{Nat}[I', J'] \lhd K''} \qquad (1)}{\varphi; \Phi; \Gamma \vdash \texttt{s}(V) : \mathsf{Nat}[I,J] \lhd K'} \qquad \cdots \qquad \dfrac{\Pi_2}{\varphi; (\Phi, J \geq 1); \Gamma, x : \mathsf{Nat}[I{-}1, J{-}1] \vdash N_2 : T \lhd K}}{\varphi; \Phi; \Gamma \vdash \texttt{match}\ \texttt{s}(V)\ \{\underline{0} \mapsto N_1;;\ \texttt{s}(x) \mapsto N_2\} : T \lhd K'{+}K}$$

where (1) is

$$\varphi; \Phi \vdash \Gamma \sqsubseteq \Delta \qquad \varphi; \Phi \vdash \mathsf{Nat}[I'{+}1, J'{+}1] \sqsubseteq \mathsf{Nat}[I,J] \qquad \varphi; \Phi \vDash K'' \leq K'$$

In particular, we have $\varphi; \Phi \vDash I \leq I'{+}1$ and $\varphi; \Phi \vDash J'{+}1 \leq J$. From this, we obtain $\varphi; \Phi \vDash I{-}1 \leq I'$, $\varphi; \Phi \vDash J \geq 1$ and $\varphi; \Phi \vDash J' \leq J{-}1$. So, we have by subtyping from Π_V:

$$\varphi; \Phi; \Gamma \vdash V : \mathsf{Nat}[I{-}1, J{-}1] \lhd K'$$

so by Lemma 3.1.5: $\varphi; (\Phi, J \geq 1); \Gamma \vdash N_2[x := V] : T \triangleleft K$, and finally by strengthening we obtain $\varphi; \Phi; \Gamma \vdash N_2[x := V] : T \triangleleft K$ and this concludes this case.

The methodology for pattern-matching is very simple, we only need to show that the actual predecessor indeed satisfies the assumption we took on predecessors, and that the constraints are satisfied. With this methodology, we can adapt the soundness proof for other data-types, and it also works for the rule described in Remark 3.1.3, using index substitution to replace i, i', j, j' by the actual bounds given in the typing.

- **Case** $C[M] \to_k C[N]$ when $M \to_k N$. This case can be done by induction on the grammar for C, all cases are straightforward.

□

Finally, from Theorem 3.1.1, we deduce the following theorem.

Theorem 3.1.2 (Soundness)**.** *If $\varphi; \Phi; \Gamma \vdash M : T \triangleleft K$, and M has complexity n, then we have $\varphi; \Phi \vDash K \geq n$. In particular, for any $\rho : \varphi \to \mathbb{N}$ such that $\rho \vDash \Phi$, we have $[\![K]\!]_\rho \geq n$.*

Notice that this theorem talks about open terms (with variables). However, our notion of complexity does not behave well with open terms. For example $\texttt{match}\ x\ \{\underline{0} \mapsto N_1; ;\ \mathtt{s}(x) \mapsto N_2\}$ is in normal form for a variable x. However, a typing of this term would give a non-zero complexity, as it would assume that x is an actual integer. So, the complexity bound should be understood as a bound on all possible closed terms obtainable from a well-typed substitution of the open term. Formally, we have the following corollary:

Corollary 3.1.1 (Complexity and Open Terms)**.** *If $\varphi; \Phi; \Gamma, \widetilde{x} : \widetilde{T} \vdash M : T \triangleleft K$, then for any sequence of values $\widetilde{V}$ such that $\varphi; \Phi; \Gamma \vdash \widetilde{V} : \widetilde{T} \triangleleft 0$, K is a bound on the complexity of $M[\widetilde{x} := \widetilde{V}]$.*

This is easily proved using Theorem 3.1.2 and Lemma 3.1.5.

So, we showed that the type system defined in this chapter indeed gives a bound on the complexity, which should have been expected with the way this type system is defined. Moreover, as stated before, the soundness result could be extended to other data-types, such as lists, words or trees. An other possible extension would be to consider *refinement types* [49], in which base types have the shape $\mathsf{Nat}\{\alpha \mid \phi(\alpha)\}$, where $\phi(\alpha)$ is a formula on α. In this case, sized types could be recovered by considering formulas of the shape $I \leq \alpha \leq J$, but depending on the set of formulas one could express more interesting properties. In particular, such a type system is used in [7] for probabilistic complexity analysis, and the similitude with our type system can be seen especially on the pattern matching rule.

In the following, we adapt the sized type system for parallel complexity analysis.

Chapter 4

Complexity in Pi-calculus

In this chapter, we focus on parallel complexity analysis in the π-calculus, with sized types. A first observation is that for the *work* analysis of the π-calculus, corresponding to the sequential computation time of a process, the constructors for parallelism should not have important consequences for the way complexity is managed, and indeed we show that an adaptation of the previous type system can directly give interesting results for the work. This adaptation uses in particular the usual input/output type system of the π-calculus, in order to manage subtyping. As the type system for work will be close to the usual sized type systems for functional languages, methods developed for functional languages on those type systems can be adapted for work-analysis of π-calculus, especially the type inference procedure of [6].

As for parallel complexity analysis, we define a notion of *span* for π-calculus, corresponding to complexity under maximal parallelism. This notion of span is defined by a small-step semantics where processes are annotated with the number of `tick` they saw. For this notion, we give two type systems. The first one is adapted from the type system for work, with time annotations that are useful for span analysis, as in [36]. Those results on work and span analysis with sized types have been published in [14]. However, this adaption leads to some limitations for the type system, especially in presence of some concurrent behaviours such as semaphores.

In the last section, we thus present a type system with *usages*, a type system paradigm originally used for deadlock-freedom [77]. This type system comes from joint work with Naoki Kobayashi, and it relies on a new way to manage time annotations in usages, specifically designed for complexity analysis. This type system, although more complex than the previous one, seems more expressive, and it can in particular handle semaphores.

4.1 The Pi-calculus : Preliminaries

In this work, we consider the π-calculus as a model of parallelism and concurrent systems. he main points of π-calculus are that processes can be composed in parallel, communications between processes happen with the use of channels, and channel names can be created dynamically, so the topology can change at runtime.

Several presentations of the π-calculus exist in the literature, so we detail here the presentation we chose for this thesis. Moreover, we give some usual definitions useful for this thesis, and we present the two type systems we rely on for complexity analysis: input/output types and usages.

4.1.1 Syntax and Standard Semantics

We present here a classical syntax for the synchronous π-calculus. More details about π-calculus and variants of the syntax can be found in [90]. The sets of *variables*, *expressions* and *processes* are defined by the following grammar.

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

We use x, y, z as meta-variables for integer variables, and a, b, c as those for channel names. The notation $\widetilde{v}$ stands for a sequence of variables $v_1, v_2, \ldots, v_k$. Similarly, $\widetilde{e}$ denotes a sequence of expressions. We work up to α-renaming, and we write $P[\widetilde{v} := \widetilde{e}]$ to denote the substitution of $\widetilde{e}$ for the free variables $\widetilde{v}$ in P. For the sake of simplicity, we consider only integers and lists as base types in the following, but the results can be generalized to other data-types, as in Section 3.1. Moreover, we consider very simple expressions with only base constructors, but we could enrich the set of expressions with functions (such as addition on integers) and still obtain the soundness. Indeed, the core of our proof lies in the parallel constructors, and the set of expressions has little impact on the theoretical results.

Intuitively, $P \mid Q$ stands for the parallel composition of P and Q. The process $a(\widetilde{v}).P$ represents an input: it stands for the reception on the channel a of a tuple of values identified by the variables $\widetilde{v}$ in the continuation P. The process $!a(\widetilde{v}).P$ is a replicated version of $a(\widetilde{v}).P$, it behaves like an infinite number of $a(\widetilde{v}).P$ in parallel. The process $\overline{a}\langle\widetilde{e}\rangle.P$ represents an output: it sends a sequence of expressions on the channel a, and continues as P. A process $(\nu a)P$ dynamically creates a new channel name a and then proceeds as P. We also have standard pattern matching on data types.

We now describe the standard semantics for this calculus. The first step is to define a congruence relation $\equiv$ on those processes. It is defined as the least congruence containing:

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

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

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

Definition 4.1.1 (Guarded Processes and Canonical Form)**.** *A process G is* guarded *if it has one of the following shapes:*

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

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

Formally, we now show that all processes have a somewhat unique canonical form, as in [76].

Lemma 4.1.1 (Existence of Canonical Form)**.** *For any process P, there is a Q in canonical form such that $P \equiv Q$.*

Proof. Le us suppose that, by renaming, all the introductions of new variables have different names and that they also differ from the free variables already in P. We can then proceed by induction on the structure of P. The only interesting case is for parallel composition. Suppose that

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

With $P_1, \ldots, P_n, Q_1, \ldots, Q_m$ guarded processes. Then, by hypothesis on the name of variables, we have $\widetilde{a}$ and $\widetilde{b}$ disjoint and $\widetilde{a}$ is not free in Q, as well as $\widetilde{b}$ is not free in P. So, we obtain

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

□

Lemma 4.1.2 (Uniqueness of Canonical Form)**.** *If*

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

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

Proof. Recall that α-renaming is not a rule of $\equiv$. Let us define a set $\mathtt{name}$ of channel variables and a multiset $\mathtt{gp}$ of guarded processes.

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

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

Lemma 4.1.3. *If $P \equiv Q$ then $\mathtt{name}(P) = \mathtt{name}(Q)$. Moreover, if $\mathtt{gp}(P) = [P_1, \ldots, P_n]$ and $\mathtt{gp}(Q) = [Q_1, \ldots, Q_m]$, then $m = n$ and for some permutation $Q'_1, \ldots, Q'_n$ of $Q_1, \ldots, Q_n$, we have $P_i \equiv Q'_i$ for all i.*

Finally, Lemma 4.1.2 is a direct consequence of Lemma 4.1.3. □

We now define the usual reduction relation for the π-calculus, that we denote $P \to Q$. It is defined by the rules given in Figure 4.1. Remark that substitution should be well-defined in order to do some reduction steps: channel names must be substituted by other channel names and base type variables can be substituted by any expression except channel names. However, when we will consider typed processes, this will always yield well-defined substitutions.

Let us give some examples of process that should be of interest in the following. As a first example, we show a way to encode a usual functional program in π-calculus. In order to do this, we use replicated input to encode functions, and we use a return channel for the output.

Example 4.1.1 (Map)**.** *Given a channel f representing a function F such that $\overline{f}\langle y, a\rangle$ returns $F(y)$ on the channel a, we can write the "map" function in our calculus as described in Figure 4.2. The main idea for this kind of encoding is to use the dynamic creation of names ν to create the return channel before calling a function, and then to use this channel to obtain back the result of this call.*

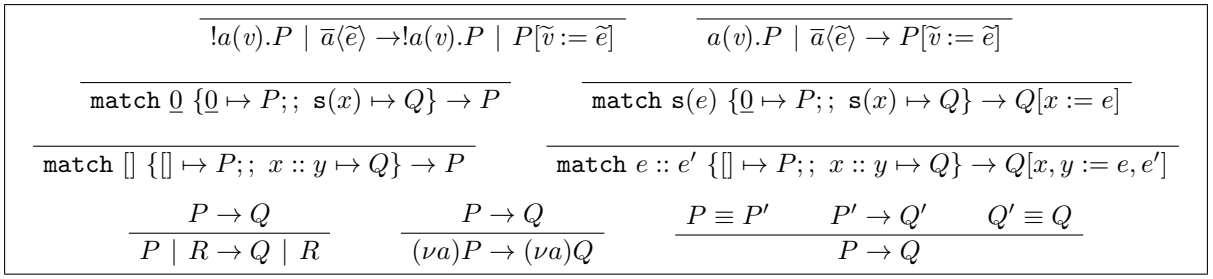

Figure 4.1: Standard Reduction Rules

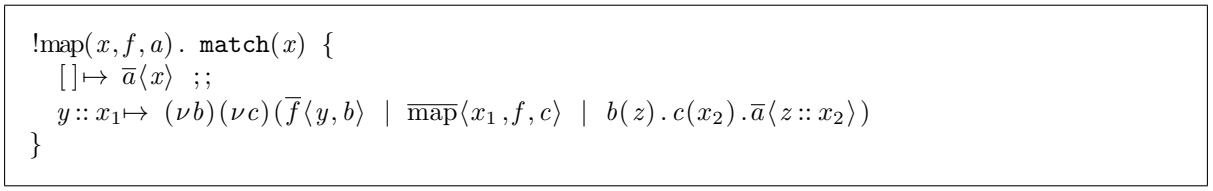

Figure 4.2: The Map Function

Example 4.1.2 (Fibonacci)**.** *As another simple example, we take a representation of the Fibonacci function in π-calculus. This is described by Figure 4.3, where both servers add and fib are put in parallel.*

This example is a simple interesting example to compare the sequential complexity (work) and the complexity under maximal parallelism (span). Indeed, we would like the work to be exponential in n and the span to be linear, since all recursive calls can be done in parallel.

4.1.2 Input/Output Types

Several typing paradigms have been proposed for the π-calculus, see [90] or [73] for a survey. As we saw in Section 3.1, with a sized type system, we want to have a subtyping relation, as this allows for more flexibility. In the π-calculus, the simplest type system that allows subtyping is the input/output type system [88]. The idea is to give to a channel some capabilities. For example, a channel with only the input capability can only be used to receive message. A channel with both input and output capabilities can be used to both receive and send messages. Let us explain intuitively why this distinction between input and output is important for subtyping.

Recall that in Section 3.1, we saw that $T \sqsubseteq U$ means that the set of values typed with T is a subset of the values typed with U. So, if we consider, for the sake of the example, a type Nat for integers, and a type Real for real numbers, then we have $\mathsf{Nat} \sqsubseteq \mathsf{Real}$. Another way to see this is that in any program where a value v is used as a real, then it is safe to feed an integer to this program. So, if $T \sqsubseteq U$, then in a program that uses a value of type U, it is safe to consider that this value has type T instead.

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

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

If we look at the variance of the subtyping relation, for an input channel we have covariance,

$$
\begin{array}{l}
!\mathrm{add}(n,m,a).\ \mathtt{match}(n)\ \{ \\
\quad \mathtt{0} \mapsto \overline{a}\langle m\rangle\ ;; \\
\quad \mathtt{s}(p) \mapsto (\nu b)\ (\overline{\mathrm{add}}\langle p,m,b\rangle \mid b(\mathrm{q}).\ \overline{a}\langle \mathtt{s}(\mathrm{q})\rangle) \\
\} \\
!\mathrm{fib}(n,a).\ \mathtt{match}(n)\ \{ \\
\quad \mathtt{0} \mapsto \overline{a}\langle \mathtt{0}\rangle\ ;; \\
\quad \mathtt{s}(m) \mapsto \mathtt{match}(m)\ \{ \\
\quad\quad \mathtt{0} \mapsto \overline{a}\langle 1\rangle\ ;; \\
\quad\quad \mathtt{s}(p) \mapsto (\nu b)(\nu c)\ (\overline{\mathrm{fib}}\langle m,b\rangle \mid \overline{\mathrm{fib}}\langle p,c\rangle \mid c(x).b(y).\ \overline{\mathrm{add}}\langle x,y,a\rangle) \\
\}\}
\end{array}
$$

Figure 4.3: The Fibonacci Function

$$
\frac{}{\mathcal{B} \sqsubseteq \mathcal{B}} \qquad \frac{\widetilde{T} \sqsubseteq \widetilde{U} \qquad \widetilde{U} \sqsubseteq \widetilde{T}}{\mathtt{ch}(\widetilde{T}) \sqsubseteq \mathtt{ch}(\widetilde{U})} \qquad \frac{}{\mathtt{ch}(\widetilde{T}) \sqsubseteq \mathtt{in}(\widetilde{T})} \qquad \frac{}{\mathtt{ch}(\widetilde{T}) \sqsubseteq \mathtt{out}(\widetilde{T})}
$$

$$
\frac{\widetilde{T} \sqsubseteq \widetilde{U}}{\mathtt{in}(\widetilde{T}) \sqsubseteq \mathtt{in}(\widetilde{U})} \qquad \frac{\widetilde{U} \sqsubseteq \widetilde{T}}{\mathtt{out}(\widetilde{T}) \sqsubseteq \mathtt{out}(\widetilde{U})} \qquad \frac{T \sqsubseteq T' \qquad T' \sqsubseteq T''}{T \sqsubseteq T''}
$$

Figure 4.4: Subtyping Rules

and for an output channel we have contravariance. Then, a channel with both capabilities should have invariance. That is why it is important to make a distinction between input channel and output channel, as they do not behave the same way with subtyping.

We now formally state this in a type system. The sets of *base types* and types are given by the following grammars.

$$
\mathcal{B} := \mathsf{Nat} \mid \mathsf{List}(\mathcal{B}) \qquad T := \mathcal{B} \mid \mathtt{ch}(\widetilde{T}) \mid \mathtt{in}(\widetilde{T}) \mid \mathtt{out}(\widetilde{T})
$$

A channel of type $\mathtt{in}(\widetilde{T})$ is used only as an input, with messages of type $\widetilde{T}$, similarly for $\mathtt{out}(\widetilde{T})$ but for output. And a channel of type $\mathtt{ch}(\widetilde{T})$ can be used both as an input and as an output.

When a type T is not a base type, we call it a *channel type*. Then, we define a subtyping relation on those types, expressed by the rules of Figure 4.4. As stated above, we have indeed covariance for input, contravariance for output and invariance for channels. We also use an explicit transitivity rule, but it is in fact not needed as it could be replaced by an exhaustive set of rules. The transitivity is only used to first remove a capability, and then do subtyping according to the variance of the remaining capability.

We can now define typing for expressions and processes. This is expressed by the rules of Figure 4.5 and Figure 4.6. We use the notation $\Gamma \vdash \widetilde{e} : \widetilde{T}$ for a sequence of typing judgements for expressions in the tuple $\widetilde{e}$.

$$
\frac{v : T \in \Gamma}{\Gamma \vdash v : T} \qquad \frac{}{\Gamma \vdash \underline{0} : \mathsf{Nat}} \qquad \frac{\Gamma \vdash e : \mathsf{Nat}}{\Gamma \vdash \mathtt{s}(e) : \mathsf{Nat}} \qquad \frac{}{\Gamma \vdash [] : \mathsf{List}(\mathcal{B})}
$$

$$
\frac{\Gamma \vdash e : \mathcal{B} \qquad \Gamma \vdash e' : \mathsf{List}(\mathcal{B})}{\Gamma \vdash e :: e' : \mathsf{List}(\mathcal{B})} \qquad \frac{\Delta \vdash e : U \qquad \Gamma \sqsubseteq \Delta \qquad U \sqsubseteq T}{\Gamma \vdash e : T}
$$

Figure 4.5: Typing Rules for Expressions

$$\frac{}{\Gamma \vdash 0} \qquad \frac{\Gamma \vdash P \qquad \Gamma \vdash Q}{\Gamma \vdash P \mid Q} \qquad \frac{\Gamma \vdash a : \mathtt{in}(\widetilde{T}) \qquad \Gamma, \widetilde{v} : \widetilde{T} \vdash P}{\Gamma \vdash !a(\widetilde{v}).P}$$

$$\frac{\Gamma \vdash a : \mathtt{in}(\widetilde{T}) \qquad \Gamma, \widetilde{v} : \widetilde{T} \vdash P}{\Gamma \vdash a(\widetilde{v}).P} \qquad \frac{\Gamma \vdash a : \mathtt{out}(\widetilde{T}) \qquad \Gamma \vdash \widetilde{e} : \widetilde{T} \qquad \Gamma \vdash P}{\Gamma \vdash \overline{a}\langle \widetilde{e} \rangle.P} \qquad \frac{\Gamma, a : T \vdash P}{\Gamma \vdash (\nu a)P}$$

$$\frac{\Delta \vdash P \qquad \Gamma \sqsubseteq \Delta}{\Gamma \vdash P} \qquad \frac{\Gamma \vdash e : \mathsf{Nat} \qquad \Gamma \vdash P \qquad \Gamma, x : \mathsf{Nat} \vdash Q}{\Gamma \vdash \mathtt{match}\ e\ \{\underline{0} \mapsto P; ;\ \mathtt{s}(x) \mapsto Q\}}$$

$$\frac{\Gamma \vdash e : \mathsf{List}(\mathcal{B}) \qquad \Gamma \vdash P \qquad \Gamma, x : \mathcal{B}, y : \mathsf{List}(\mathcal{B}) \vdash Q}{\Gamma \vdash \mathtt{match}\ e\ \{[] \mapsto P; ;\ x :: y \mapsto Q\}}$$

Figure 4.6: Input/Output Typing Rules for Processes

So, in the following, for the first two type systems of Section 4.2 and Section 4.4, we will use input/output types and then define the subtyping relation accordingly.

4.1.3 An Introduction to Usages

In the last typing system of this thesis, of Section 4.5, we will consider *usages*, another typing paradigm of π-calculus usually used for deadlock-freedom analysis [75, 72, 77]. The main idea of usages is to type a channel with a simple abstract program describing the behaviour of this channel independently of other channels, and then to analyse the property of this abstract program to try to obtain a property on the initial process.

In this section, we describe briefly and informally a usage type system, namely a simple version of the one in [72], to analyse deadlock-freedom.

The basic idea is to define *usages* by the following grammar:

$$U := 0 \;\big|\; \mathtt{In}.U \;\big|\; \mathtt{Out}.U \;\big|\; (U_1 \mid U_2)$$

So, basically, a usage is a very simple parallel program, with `In` that can communicate with `Out`. The idea is then to type a channel by a program describing its behaviour. For example, in the process

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

we would then give the respective usages U_a and U_b to a and b:

$$U_a = \mathtt{In} \mid \mathtt{Out} \mid 0 \qquad U_b = 0 \mid \mathtt{In} \mid \mathtt{Out}$$

As the process can be modified with a congruence relation, it seems then natural to also take a congruence relation for usages. We can thus simply take

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

mimicking the congruence relation for the π-calculus.

And then, if we look at subject reduction, we see that

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

so, even if the usage of a stays the same, the usage of b has been changed to $U_b' = 0$. But this is not a problem, since U_b' is in fact obtained by one reduction step from U_b, with the intuition that $\mathtt{In}.U \mid \mathtt{Out}.V \to (U \mid V)$. So, we can define a semantics for usages, and then a reduction step of a process induces a reduction step in one of the usages.

However, such simple usages are not useful in practice. Indeed, consider the process

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

The usages of channels in this process are exactly U_a and U_b defined above. However, the previous process was perfectly fine but this one is deadlocked, so we must be able to differentiate them. In order to do that, time annotations are introduced.

Formally, the set of usages is in fact defined by the following grammar:

$$U := 0 \;\Big|\; \mathtt{In}_{t_c}^{t_o}.U \;\Big|\; \mathtt{Out}_{t_c}^{t_o}.U \;\Big|\; (U \mid V) \qquad t_o, t_c \in \mathbb{N}_\infty = \mathbb{N} \cup \{\infty\}$$

Those time indications on usages should be seen as a kind of *assume-guarantee* reasoning. In [72], the usage $\mathtt{In}_{t_c}^{t_o}.U$ describes a channel that will be used for input, and then used according to U. The two time annotations t_o and t_c are called the *obligation* and the *capacity* respectively. The obligation t_o indicates a *guarantee* that if the channel is indeed used for input, then the input will become ready before t_o time units. The capacity t_c indicates the *assumption* that an output will be provided before t_c time units once the input is ready. We have a similar meaning for $\mathtt{Out}_{t_c}^{t_o}.U$. In [72], a unit of time corresponds to the time of one communication between an input and an output. Informally, we then say that a usage is *reliable* if the associated assume-guarantee reasoning is correct. So, for example, $\mathtt{In}_1^0 \mid \mathtt{Out}_0^1$ is reliable, because if we look at the input, the capacity says that the environment should provide an output within one unit of time, and this is correct since the obligation of the output is 1. However, the usage $\mathtt{In}_1^0 \mid \mathtt{Out}_0^2$ is not reliable, as the assumption on the capacity 1 of the input is incorrect. Note that this reliability should always be true during a computation, so it should also be verified after any number of reduction steps.

Let us come back the example given before:

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

The usage U_b should have the shape:

$$U_b = \mathtt{In}_{t_c}^{t_o} \mid \mathtt{Out}_{t_c'}^{t_o'}$$

Let us look at obligations. Both the input and the output are ready immediately, so we can take $t_o = t_o' = 0$. Then, for the input, we have a guarantee that the output is ready in 0 unit of time, so we can take $t_c = 0$, and similarly, we have $t_c' = 0$. So, we have

$$U_b = \mathtt{In}_0^0 \mid \mathtt{Out}_0^0$$

Then, for U_a, we have also a shape, with unknown t_o, t_o', t_c, t_c':

$$U_a = \mathtt{In}_{t_c}^{t_o} \mid \mathtt{Out}_{t_c'}^{t_o'}$$

Again, the input is ready immediately, so $t_o = 0$. For the output, we need to wait for the input of b, so $t_o' = 1$. Then, we obtain by symmetry $t_c = 1$ and $t_c' = 0$. So,

$$U_a = \mathtt{In}_1^0 \mid \mathtt{Out}_0^1$$

.

Now, let us come back to

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

and consider the usages:

$$U_a = \mathtt{In}_{t_c^1}^{t_o^1} \mid \mathtt{Out}_{t_c^2}^{t_o^2} \qquad U_b = \mathtt{In}_{t_c^3}^{t_o^3} \mid \mathtt{Out}_{t_c^4}^{t_o^4}$$

Let us examine the usage U_a. We have easily $t_o^1 = 0$, because the input is immediately ready. For t_o^2, we observe that in order for the output of a to be ready, the input on b must be reduced. Looking at the usage of b, we can say that $t_o^2 \geq t_c^3 + 1$, since t_c^3 gives a bound on the time b need to synchronize, and this synchronization counts for one time unit. Moreover, in order to have a reliable usage, the capacity should verify $t_c^3 \geq t_o^4$. Then by symmetry, as the output for b needs to wait for the input on a, we have $t_o^4 \geq t_c^1 + 1$. Again, by reliability, it must be the case that $t_c^1 \geq t_o^2$. So, we have:

$$t_o^2 \geq t_c^3 + 1 \geq t_o^4 + 1 \geq t_c^1 + 2 \geq t_o^2 + 2$$

And we obtain $t_o^2 = \infty$. In the end, we therefore obtain the usages:

$$U_a = \mathtt{In}_{\infty}^{0} \mid \mathtt{Out}_{0}^{\infty} \qquad U_b = \mathtt{In}_{\infty}^{0} \mid \mathtt{Out}_{0}^{\infty}$$

stating that, in fact, the output on a and b will never be ready, and thus we have indeed a deadlock.

Finally, in a usage type system, a *subusage* relation is defined: similarly to subtyping, we need some flexibility on usages. As a simple example, if a channel is typed with the usage $\mathtt{In}_{t_c'}^{t_o'}$, then in particular, it must be ready before t_o' unit of time, so if $t_o' \leq t_o$, it is also ready before t_o unit of time. Moreover, if it is typed under the assumption that the context will provide something within t_c' unit of time, then in particular, under the assumption that the context will provide something faster, within t_c unit of time where $t_c \leq t_c'$, the typing should still be safe. So, we obtain:

$$\frac{t_o' \leq t_o \qquad t_c \leq t_c'}{\mathtt{In}_{t_c}^{t_o} \sqsubseteq \mathtt{In}_{t_c'}^{t_o'}}$$

To summarize, if we read this relation from right to left, it is safe for the typing to weaken the guarantee and to strengthen the assumption.

So, to sum up the main points of usages:

1. Usages are defined as simple parallel programs, with time annotations. Those programs come with a congruence relation.

2. A semantics is then defined for usages, allowing input and output to communicate.

3. A notion of reliable usages is defined, where a usage is reliable if the assume-guarantee reasoning is always sound, even after some reduction steps.

4. A subusage relation must also be defined, to have some flexibility. This subusage relation is also very important for soundness.

Note that there is a choice to be made in a usage type system. One can enrich usages with choice, or replication. Then, depending on the meaning one want to give to time annotations, reliability would be defined differently. Similarly, the semantics and the subusage relation can differ depending on the way time annotations are used. In the literature, usages have been mainly used for deadlock-freedom, but we will give in Section 4.5 a usage type-system for complexity analysis.

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

Figure 4.7: Simple Tick Reduction Rules

4.2 Work of a Process

We now proceed to present our contributions to complexity analysis in the π-calculus. As in Section 3.1, we introduce a `tick` constructor, as the only source of complexity of this language. Again, from this several notions of complexity can be derived. For example, counting the number of communication steps consists in adding a `tick` after each input. Similarly, the number of reduction steps can be recovered by also adding a `tick` after each pattern matching. It can also be useful to single out what is a costly operation. For example, in a sorting algorithm, we can use `tick` to count the number of comparisons, ignoring everything else.

4.2.1 Semantics and Type System

Semantics

We first describe a semantics for the work, that is to say the total number of ticks during a reduction without parallelism. The time reduction $\to_1$ is defined in Figure 4.7. Intuitively, this reduction removes exactly one tick at the top-level.

Then from any process P, a sequence of reduction steps to Q is just a sequence of one-step reductions with the standard reduction $\to$ defined in Figure 4.1 or $\to_1$, and the work complexity of this sequence is the number of $\to_1$ steps. In this paper, we always consider the worst-case complexity so the work of a process is defined as the maximal complexity over all such sequences of reduction steps from this process.

Notice that with this semantics for work, adding `tick` in a process does not change its behaviour: we do not create nor erase reduction paths.

Example 4.2.1 (Fibonacci)**.** *We take back the process P described in Example 4.1.2 with a slight modification: after the replicated input $!fib$ we add a* `tick`*. We can see that the work of $(\nu a)(P \mid \overline{fib}\langle 10, a\rangle)$ is $F(10)$ where F is defined by:*

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

Size Input/Output Types

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

Definition 4.2.1. *The set of* base types *is given by the following grammar.*

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

As for Definition 3.1.6, in the context $\varphi; \Phi$, an integer n of type $\mathsf{Nat}[I, J]$ must be such that $\varphi; \Phi \vDash I \leq n \leq J$. Likewise, a list of type $\mathsf{List}[I, J](\mathcal{B})$ must have a length between I and J. We

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

Figure 4.8: Subtyping Rules for Base Sized types

may use $\mathsf{Nat}[I]$ to denote the type $\mathsf{Nat}[I,I]$ in order to gain some space, especially in examples of type derivations.

As in Section 3.1, with those types comes a notion of subtyping, in order to have some flexibility on bounds. This is described by the rules of Figure 4.8.

Then, after base types, we have to give a type to channel names in a process. As we want to generalize subtyping for channel types, we will use input/output types, as explained before in Section 4.1.2.

Unlike in usual input/output types, in this work we also distinguish two kinds of channels: the *simple channels* (that we will often call channels), and replicated channels (called *servers*). All the inputs on a server channel must be replicated (as in $!a(\widetilde{v}).P$), while no input on a simple channel can be replicated.

Definition 4.2.2. *The set of types is given by the following grammar.*

$$T := \mathcal{B} \mid \mathtt{ch}(\widetilde{T}) \mid \mathtt{in}(\widetilde{T}) \mid \mathtt{out}(\widetilde{T}) \mid \forall\widetilde{i}.\mathtt{srv}^K(\widetilde{T}) \mid \forall\widetilde{i}.\mathtt{isrv}^K(\widetilde{T}) \mid \forall\widetilde{i}.\mathtt{osrv}^K(\widetilde{T})$$

The three different types for channels and servers correspond to the three different sets of capabilities. We note srv when the server have both capabilities, isrv when it has only input and osrv when it has only output. Then, for servers, we have additional information: there is a quantification over index variables, and the index K stands for the *complexity* of the process spawned by this server. This type for servers is very similar to the arrow type described in Section 3.1.

A typical example could be a server taking as input a list and a channel, and sending to this channel the sorted list, in time $k \cdot n$ where n is the size of the list: $P = !a(x,b).\cdots\overline{b}\langle e\rangle$ where e represents at the end of the process the list x sorted. Such a server name a could be given the type $\forall i.\mathtt{srv}^{k\cdot i}(\mathsf{List}[0,i](\mathcal{B}), \mathtt{out}(\mathsf{List}[0,i](\mathcal{B})))$. This type means that for all integers i, if given a list of size at most i and an output channel waiting for a list of size at most i, the process spawned by this server will stop at time at most $k \cdot i$. As for functional programs, the quantified index variables are very useful especially for replicated input, in the same way it was useful for recursive functions. Some more details about the similarities will be explained in Example 4.2.2.

Then, we describe subtyping for servers in Figure 4.9. Those rules have flavor similar to the one described in Section 4.1.2. Note that the complexity can be modified by subtyping, with a variance depending on the capabilities. As before, we can understand the variance by looking at what modifications are safe. In an input server, if we can safely guarantee when defining a server that the complexity is no greater than K', then we can safely guarantee that the complexity is no greater than K with $K \geq K'$. Similarly, for an output server, if we can derive a complexity bound under the assumption that the computation will not last longer than K', then we can also derive this same complexity bound under the assumption that the computation will not last longer than K, with $K \leq K'$.

We can now present the type system. Rules for expressions are given in Figure 4.10. The typing for expressions $\varphi;\Phi;\Gamma \vdash e : T$ means that under the constraints Φ, in the context Γ, the expression e can be given the type T.

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

$$\frac{}{\varphi;\Phi\vdash\forall\widetilde{i}.\mathtt{srv}^K(\widetilde{T})\sqsubseteq\forall\widetilde{i}.\mathtt{isrv}^K(\widetilde{T})}\qquad\frac{}{\varphi;\Phi\vdash\forall\widetilde{i}.\mathtt{srv}^K(\widetilde{T})\sqsubseteq\forall\widetilde{i}.\mathtt{osrv}^K(\widetilde{T})}$$

$$\frac{(\varphi,\widetilde{i});\Phi\vdash\widetilde{T}\sqsubseteq\widetilde{U}\qquad(\varphi,\widetilde{i});\Phi\vDash K'\leq K}{\varphi;\Phi\vdash\forall\widetilde{i}.\mathtt{isrv}^K(\widetilde{T})\sqsubseteq\forall\widetilde{i}.\mathtt{isrv}^{K'}(\widetilde{U})}\qquad\frac{(\varphi,\widetilde{i});\Phi\vdash\widetilde{U}\sqsubseteq\widetilde{T}\qquad(\varphi,\widetilde{i});\Phi\vDash K\leq K'}{\varphi;\Phi\vdash\forall\widetilde{i}.\mathtt{osrv}^K(\widetilde{T})\sqsubseteq\forall\widetilde{i}.\mathtt{osrv}^{K'}(\widetilde{U})}$$

$$\frac{\varphi;\Phi\vdash T\sqsubseteq T'\qquad\varphi;\Phi\vdash T'\sqsubseteq T''}{\varphi;\Phi\vdash T\sqsubseteq T''}$$

Figure 4.9: Subtyping Rules for Server Types

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

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

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

Figure 4.10: Typing Rules for Expressions

Then, rules for processes are described in Figure 4.11 and Figure 4.12. Figure 4.12 describes rules specific to work, whereas rules in Figure 4.11 will be reused for span. A typing judgement $\varphi;\Phi;\Gamma\vdash P\triangleleft K$ intuitively means that under the constraints Φ, in a context Γ, a process P is typable and its work complexity is bounded by K.

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

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

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

$$\frac{\varphi;\Phi;\Delta\vdash P\triangleleft K'\qquad\varphi;\Phi\vdash\Gamma\sqsubseteq\Delta\qquad\varphi;\Phi\vDash K'\leq K}{\varphi;\Phi;\Gamma\vdash P\triangleleft K}$$

Figure 4.11: Common Typing Rules for Processes

The rules can be seen as a combination of the input/output typing rules of Section 4.1.2 with rules for the size type system for functional program of Section 3.1. The main differences are that because of the two kinds of channels, we need two rules for an output. Note that a replicated input has complexity zero, and it is a call to this server that generates complexity in the type system, as for functions in Section 3.1. This is because once defined, a replicated input stays during all the reduction, so we do not want them to generate complexity.

Note that there is a dissymmetry between servers and simple channel in this type system: for servers, complexity comes from output and thus an index to keep track of the complexity

$$\frac{\varphi;\Phi;\Gamma\vdash P\triangleleft K_1 \qquad \varphi;\Phi;\Gamma\vdash Q\triangleleft K_2}{\varphi;\Phi;\Gamma\vdash P\mid Q\triangleleft K_1{+}K_2} \qquad \frac{\varphi;\Phi;\Gamma\vdash P\triangleleft K}{\varphi;\Phi;\Gamma\vdash \mathtt{tick}.P\triangleleft K{+}1}$$

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

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

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

Figure 4.12: Work Typing Rules for Processes

is needed in the type, whereas for simple channels, complexity comes from input. However, this dissymmetry is not necessary. Indeed, servers need to have those typing rules, because we want a replicated input to have complexity 0 and we want polymorphism over indices, but for simple channels we have a choice. We can either keep the current type system or modify simple channels types such that they are similar to servers types (with polymorphism over indexes and an index for complexity in the type). We choose to present the current type system first because this way we can show this alternative choice of typing for simple channels instead of mimicking servers. Another interesting type system would be a totally non-syntax directed type-system for which we have the choice between those two typing behaviours for simple channels. For the sake of simplicity, we avoid presenting this type system, but in fact the two choices have advantages and disadvantages and so allowing two different choices of typing can increase expressivity.

Remark 4.2.1 (Advantages and Disadvantages of the Different Typing Behaviours)**.** *Consider this example:*

$$a().P_1 \mid a().P_2 \mid \overline{a}\langle\rangle \mid \overline{a}\langle\rangle \mid \overline{a}\langle\rangle$$

With the typing behaviour for simple channels described in Figure 4.12, we obtain a complexity bound of the shape $K_1 + K_2$ where K_1 is the complexity of P_1 and K_2 is the complexity of P_2. If we use a server-like typing behaviour, as described in Figure 4.15, then we have to give a complexity for the channel a, and this complexity must be $\max(K_1, K_2)$. So, in the end, we obtain a complexity bound equals to $3 * \max(K_1, K_2)$, which is less precise than $K_1 + K_2$. However, if we consider this example:

$$a().P_1 \mid a().P_2 \mid \overline{a}\langle\rangle$$

Then, the typing behaviour presented in Figure 4.12 gives us a complexity bound $K_1 + K_2$, whereas a server-like typing behaviour gives us a complexity bound $\max(K_1, K_2)$.

So, to sum up, informally the behaviour presented in Figure 4.12 gives more precise complexity bound when there are no locked inputs but the server-like typing behaviour is more precise when there are locked inputs. That is why a type system where we have a choice should be better for theoretical analysis.

Example 4.2.2. *Let us take again the process for Fibonacci described in Example 4.1.2. We first give a typing for add. As there is no tick in add, this typing only shows how input/output types and sized types work. A simplified version of this typing is given in Figure 4.13. In order to gain some space, we do not precise the easy premises, and we automatically rename the typing contexts. Also, recall that $\mathsf{Nat}[I]$ denotes the type $\mathsf{Nat}[I, I]$.*

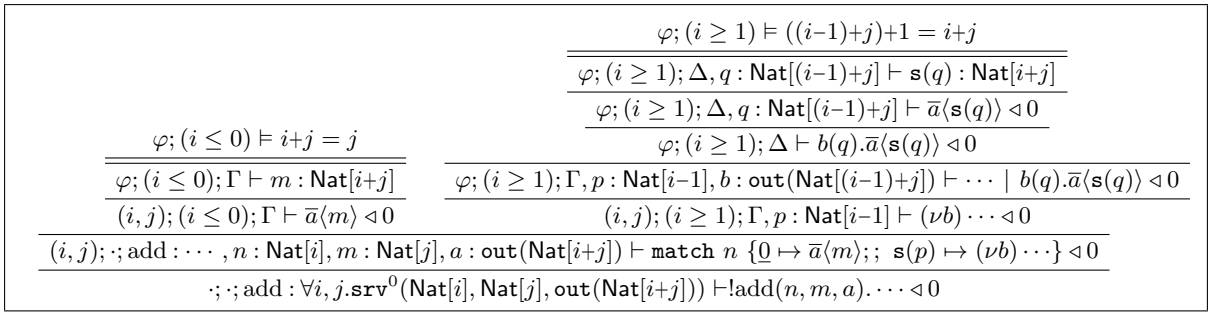

Figure 4.13: A Typing Without Complexity for Addition

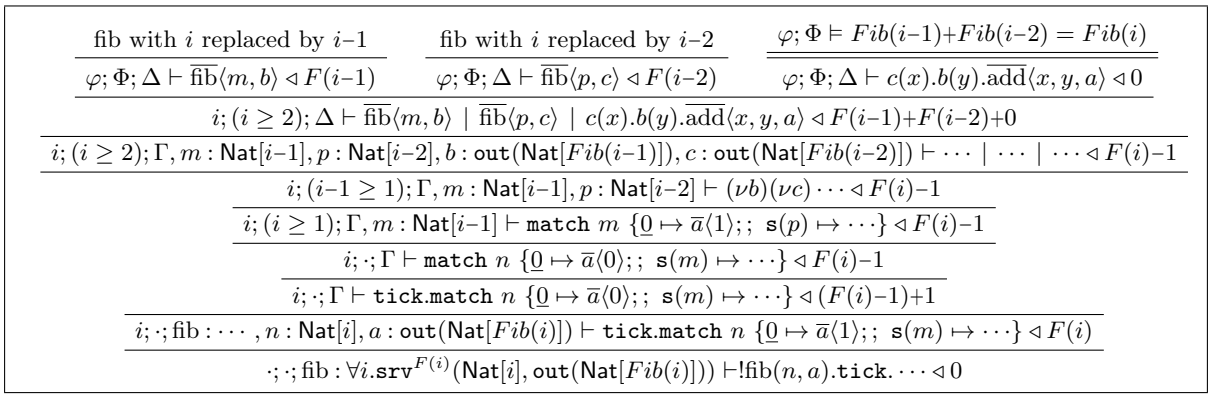

Figure 4.14: A Typing for Fibonacci (Work)

The way sizes and indices are used in this typing is similar to Section 3.1.

We now describe the typing for the Fibonacci function. Again, we will only focus on interesting premises. The type derivation is given in Figure 4.14. We use the Fibonacci function in indices, denoted $Fib(I)$, and we also use the function F defined in Example 4.2.1.

4.2.2 Soundness of the Type System

We now state the properties of this typing system. We do not detail the proofs as all proofs are a simplified version of the one for span that will be described in Section 4.4.2. The proof follows the same methodology as Section 3.1: in this type system for work, we can easily obtain the structural properties: weakening and strengthening. Then, we can prove the substitution lemmas, and with those properties, we obtain the usual subject reduction.

Theorem 4.2.1 (Subject Reduction)**.** *If $\varphi; \Phi; \Gamma \vdash P \triangleleft K$ then:*

- *If $P \to Q$ then $\varphi; \Phi; \Gamma \vdash Q \triangleleft K$.*
- *If $P \to_1 Q$ then $\varphi; \Phi; \Gamma \vdash Q \triangleleft K'$ with $\varphi; \Phi \vDash K'{+}1 \leq K$.*

Proof. The second point can be proved by induction on $P \to_1 Q$. All the cases are direct, since the rule for parallel composition is the sum of the two complexities and the rule for ν does not change the complexity. Finally, the rule for tick gives directly this property. □

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

Theorem 4.2.2 (Work Complexity Bound)**.** *If $\varphi; \Phi; \Gamma \vdash P \triangleleft K$ then, if we call n the work complexity of P, we have $\varphi; \Phi \vDash K \geq n$. In particular, for any $\rho : \varphi \to \mathbb{N}$ such that $\rho \vDash \Phi$, we have $[\![K]\!]_\rho \geq n$.*

So, again, this complexity bound only makes sense if the set of constraints Φ is satisfiable. We emphasize the fact that this soundness result is easily adaptable to similar processes and type systems for work. As stated before, we can enrich processes with other algebraic data-types and the proof can easily be adapted. We can also change the typing behaviour for simple channels, and we still get the soundness. An interesting consequence of this soundness theorem is that it immediately gives soundness for any subsystem. In particular, we now detail a weaker typing system where the shapes of types are restricted in order to have an inference procedure close to the one in [6].

4.2.3 A Hint for Type Inference

We present in this section a type inference procedure based from [6]. This work gives an inference procedure for a sized type system for functional programs, in a similar fashion of the one presented in Section 3.1. In order to define the procedure smoothly, arrow types are restricted (restricted types are called *canonical* types). This paper then describes a sound and complete inference procedure for this canonical type system. We describe how to obtain intuitively a sound and complete type procedure for a subsystem of our work type system by mimicking the inference procedure of [6]. Intuitively, in [6], the canonical form of a type forces the input of functions to have the shape $\mathsf{Nat}[0, i]$ (or $\mathsf{List}[0, i](\mathcal{B})$) for some new fresh variable i. Then, in a type more complex indices, such as i^2+j or other expressions, only appear at positions that correspond to outputs.

So, what we will do is to give a type system inspired by this canonical form. Then, if we can show that this type system is a restriction of the previous one for work, we obtain automatically its soundness (with regard to work complexity). We can then use an inference procedure as in [6], that we will not detail here, but it will be described in the Appendix A.1. There are two steps to this type inference procedure, first we need to be able to generate constraints from a process such that if this set of constraints is satisfiable, then the process is typable (soundness). In order to show expressivity, we would also like that if a process is typable, then the generated set of constraints is indeed satisfiable (completeness). With this, we know that we do not lose expressivity with the reduction to a constraint satisfaction problem. Then, the second step, is to give those constraints to a SMT solver and hope that it can solve it (recall that this is undecidable in general). In this section, we will focus on the first step of this procedure.

We begin by changing the behaviour of channels, in order to have a quantification for channel variables even for channel types, then we merge servers and channels together. This way, we obtain a type for channels very close to the arrow type in [6].

Definition 4.2.3. *The set of types and* base types *are given by the following grammar.*

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

With the associated subtype system described in Figure 4.8 and Figure 4.9 where `srv` is replaced by `ch`. The typing rules are given by Figure 4.10, Figure 4.11 and Figure 4.15. So, the only modification compared to the system we presented before is that channel types and server types are not distinct.

$$\frac{\varphi;\Phi;\Gamma\vdash P\triangleleft K_1 \qquad \varphi;\Phi;\Gamma\vdash Q\triangleleft K_2}{\varphi;\Phi;\Gamma\vdash P\mid Q\triangleleft K_1{+}K_2} \qquad \frac{\varphi;\Phi;\Gamma\vdash P\triangleleft K}{\varphi;\Phi;\Gamma\vdash \texttt{tick}.P\triangleleft K{+}1}$$

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

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

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

Figure 4.15: Typing Rules for Processes for Work Inference

The Restricted Type System

In order to obtain a procedure for our type system, we would first need to begin by a classical algorithm such that, given a process P, this algorithm gives a classical π-calculus type to this process P. This is a standard type inference algorithm. This algorithm outputs a type in the following grammar:

$$\mathcal{B}_A := \mathsf{Nat} \mid \mathsf{List}(\mathcal{B}) \mid \alpha,\beta,\dots \qquad T_A := \mathcal{B}_A \mid \texttt{ch}(\widetilde{T_A}) \mid A,B,\dots$$

Where $\alpha,\beta,\dots$ are variables for base types, and $A,B,\dots$ are variables for channel types. So, similarly, our type inference procedure will use those base types variables $\alpha,\beta,\dots$ and those channel types variables $A,B,\dots$.

For now, we only have a type system that gives a sound bound on the complexity, described in Figure 4.15. So, we need to adapt it in order to obtain a type system very close to the one in [6] so that we can mimic the type inference procedure. We call this type system the *intermediate type system*.

Definition 4.2.4. *For this intermediate type system, we consider the following grammar.*

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

Then, we define types in canonical form, inspired by [6].

Definition 4.2.5 (Canonical Intermediate Types)**.** *A* canonical intermediate type *is a type given by the following grammar:*

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

Where i is an index variable, n an integer and in the channel type, the index variables of K are in $\{i_1,\dots,i_m\}$ and m is equal to the number of base type index occurrences in $\widetilde{T_c}$, noted $\texttt{btocc}(\widetilde{T_c})$, and defined by:

- $\texttt{btocc}(T_c^1,\cdots,T_c^k) = \texttt{btocc}(T_c^1){+}\cdots{+}\texttt{btocc}(T_c^k)$
- $\texttt{btocc}(\mathsf{Nat}[0,i]) = 1$
- $\texttt{btocc}(\mathsf{List}[0,i](\mathcal{B}_c)) = 1{+}\texttt{btocc}(\mathcal{B}_c)$
- $\texttt{btocc}(\alpha) = 0$

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

$$\frac{}{\varphi; \Gamma \vdash [] : \mathsf{List}[0,0]} \qquad \frac{\varphi; \Gamma \vdash e : \mathcal{B}_i^1 \qquad \varphi; \Gamma \vdash e' : \mathsf{List}[0,I](\mathcal{B}_i^2) \qquad \varphi; \cdot \vdash \mathcal{B}_i^1, \mathcal{B}_i^2 \sqsubseteq \mathcal{B}_i}{\varphi; \Gamma \vdash e :: e' : \mathsf{List}[0,I{+}1](\mathcal{B}_i)}$$

Figure 4.16: Intermediate Typing Rules for Expressions

- $\mathtt{btocc}(\forall i_1, \ldots, i_m.\, c\!h^K(\widetilde{T_c})) = \mathtt{btocc}(A) = 0$

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

Example 4.2.3. *The following type is a canonical channel type:*

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

The first quantification is over 3 *index variables because there are exaclty* 3 *positions in this type in which we can put an index variable without crossing another quantifier. Then, those* 3 *variables are indeed ordered from left to right. All the subtypes are also canonical, and notice that base type variables and type variables are not taken in account in the counting of index variables.*

Obviously, a canonical intermediate type is an intermediate type. So, in our intermediate type system, we will ask that all channel names have a canonical type. Moreover, in a context Γ, we will require all types to be canonical. Formally, a typing judgment has the shape $\varphi; \Gamma \vdash P \triangleleft K$ where Γ contains only canonical types, and there is no set of constraints Φ as there are no such constraints in [6]. Subtyping is defined as a restriction of the previous subtyping relation to intermediate types, and the type system is given by Figure 4.16 and Figure 4.17. One can see that the invariants described above on canonical types are respected. Also, there are no subtyping rules in this type system, as canonical types do not need subtyping, and the modifications on the complexity bounds are internalized in the useful rules. Moreover, in this typing, and for the following of this section, we only consider processes with well written pattern matching, meaning that pattern matching can only be done on base type variable (otherwise, the pattern matching is not useful...). Please note that this intermediate type system is not used for its theoretical value, that is why we can ask those kinds of restrictions, since we are not interested in subject reduction for this intermediate type system. Note that thanks to this restriction, we can consider that the base type element in a pattern matching is a canonical type, and this helps us get rid of the previous rule for pattern matching when we needed to add constraints in the branches. The idea is that instead of adding the constraints $i \geq 1$ in the typing, we do a substitution and replace i by an index of the shape $i{+}1$, and so i now becomes the size of the predecessor (or the tail for a list). Without this set of constraint Φ in a typing, we obtain a type system closer to the one in [6].

Now, let us show that if a process is typable with those intermediate types, then it is typable for the type system presented in the beginning of this section.

$$\frac{}{\varphi;\Gamma \vdash 0 \triangleleft 0} \qquad \frac{\varphi;\Gamma \vdash P \triangleleft K_1 \qquad \varphi;\Gamma \vdash Q \triangleleft K_2 \qquad \varphi;\cdot \vDash K_1{+}K_2 \leq K}{\varphi;\Gamma \vdash P \mid Q \triangleleft K} \qquad \frac{\varphi;\Gamma, a : T_c \vdash P \triangleleft K}{\varphi;\Gamma \vdash (\nu a)P \triangleleft K}$$

$$\frac{\varphi;\Gamma \vdash a : \forall \widetilde{i}.\mathtt{ch}^K(\widetilde{T_c}) \qquad (\varphi,\widetilde{i});\Gamma,\widetilde{v} : \widetilde{T_c} \vdash P \triangleleft K' \qquad (\varphi,\widetilde{i});\cdot \vDash K' \leq K}{\varphi;\Gamma \vdash !a(\widetilde{v}).P \triangleleft 0}$$

$$\frac{\varphi;\Gamma \vdash a : \forall \widetilde{i}.\mathtt{ch}^K(\widetilde{T_c}) \qquad (\varphi,\widetilde{i});\Gamma,\widetilde{v} : \widetilde{T_c} \vdash P \triangleleft K' \qquad (\varphi,\widetilde{i});\cdot \vDash K' \leq K}{\varphi;\Gamma \vdash a(\widetilde{v}).P \triangleleft 0}$$

$$\frac{\varphi;\Gamma \vdash a : \forall \widetilde{i}.\mathtt{ch}^K(\widetilde{T_c}) \qquad \varphi;\Gamma \vdash \widetilde{e} : \widetilde{T_c}\{\widetilde{J}/\widetilde{i}\} \qquad \varphi;\cdot \vDash K\{\widetilde{J}/\widetilde{i}\} \leq K'}{\varphi;\Gamma \vdash \overline{a}\langle\widetilde{e}\rangle \triangleleft K'} \qquad \frac{\varphi;\Gamma \vdash P \triangleleft K}{\varphi;\Gamma \vdash \mathtt{tick}.P \triangleleft K{+}1}$$

$$\frac{\varphi;\Gamma \vdash v : \mathsf{Nat}[0,i] \quad \varphi;\Gamma \vdash P \triangleleft K_1 \quad \varphi;\cdot \vDash K_1 \leq K \quad \varphi;\Gamma\{i{+}1/i\}, x : \mathsf{Nat}[0,i] \vdash Q \triangleleft K_2 \quad \varphi;\cdot \vDash K_2 \leq K\{i{+}1/i\}}{\varphi;\Gamma \vdash \mathtt{match}\ v\ \{\underline{0} \mapsto P;;\ \mathtt{s}(x) \mapsto Q\} \triangleleft K}$$

$$\frac{\varphi;\Gamma \vdash v : \mathsf{Nat}[0,i{+}n{+}1] \quad \varphi;\Gamma \vdash P \triangleleft K_1 \quad \varphi;\cdot \vDash K_1 \leq K \quad \varphi;\Gamma, x : \mathsf{Nat}[0,i{+}n] \vdash Q \triangleleft K_2 \quad \varphi;\cdot \vDash K_2 \leq K}{\varphi;\Gamma \vdash \mathtt{match}\ v\ \{\underline{0} \mapsto P;;\ \mathtt{s}(x) \mapsto Q\} \triangleleft K}$$

$$\frac{\varphi;\Gamma \vdash v : \mathsf{List}[0,i](\mathcal{B}_i)\ \varphi;\Gamma \vdash P \triangleleft K_1\ \varphi;\Gamma\{i{+}1/i\}, x : \mathcal{B}_i, y : \mathsf{List}[0,i](\mathcal{B}_i) \vdash Q \triangleleft K_2\ \varphi \vDash K_1 \leq K; K_2 \leq K\{i{+}1/i\}}{\varphi;\Gamma \vdash \mathtt{match}\ v\ \{[] \mapsto P;;\ x :: y \mapsto Q\} \triangleleft K}$$

$$\frac{\varphi;\Gamma \vdash v : \mathsf{List}[0,i{+}n{+}1](\mathcal{B}_i)\ \varphi;\Gamma \vdash P \triangleleft K_1\ \varphi;\cdot \vDash K_1 \leq K\ \varphi;\Gamma, x : \mathcal{B}_i, y : \mathsf{List}[0,i{+}n](\mathcal{B}_i) \vdash Q \triangleleft K_2\ \varphi;\cdot \vDash K_2 \leq K}{\varphi;\Gamma \vdash \mathtt{match}\ v\ \{[] \mapsto P;;\ x :: y \mapsto Q\} \triangleleft K}$$

Figure 4.17: Intermediate Typing Rules for Processes

Theorem 4.2.3. *If a process P is such that $\varphi;\Gamma \vdash P \triangleleft K$ by the rules of Figure 4.16 and Figure 4.17, then, for any Γ_{sub} that is Γ where all type variables have been substituted by actual types, we have $\varphi;\cdot;\Gamma_{sub} \vdash P \triangleleft K$ for the rules of Figure 4.10, Figure 4.11 and Figure 4.15.*

Proof. The proof is done by induction on $\varphi;\Gamma \vdash P \triangleleft K$. The type variables do not cause any problem, since in a intermediate typing, if we can use a type variable then it means that this type is never really inspected. When typing an expression, this is rather direct, using subtyping for expressions. Again, for typing rules for processes, a lot of cases are direct just by using the subtyping rule of Figure 4.11. Thus, we only detail the interesting cases.

- If the typing is

$$\frac{\dfrac{a : \forall \widetilde{i}.\mathtt{ch}^K(\widetilde{T_c}) \in \Gamma}{\varphi;\Gamma \vdash a : \forall \widetilde{i}.\mathtt{ch}^K(\widetilde{T_c})} \qquad (\varphi,\widetilde{i});\Gamma,\widetilde{v} : \widetilde{T_c} \vdash P \triangleleft K' \qquad (\varphi,\widetilde{i});\cdot \vDash K' \leq K}{\varphi;\Gamma !a(\widetilde{v}).P \triangleleft 0}$$

  Note that with the rules of Figure 4.16, the only way to type a channel is by the axiom rule. Then, we can give the following type:

$$\frac{\dfrac{a : \forall \widetilde{i}.\mathtt{ch}^K(\widetilde{T_{csub}}) \in \Gamma_{sub}}{\varphi;\cdot;\Gamma_{sub} \vdash a : \forall \widetilde{i}.\mathtt{in}^K(\widetilde{T_{csub}})} \qquad \dfrac{(\varphi,\widetilde{i});\cdot;\Gamma_{sub},\widetilde{v} : \widetilde{T_{csub}} \vdash P \triangleleft K' \qquad (\varphi,\widetilde{i});\cdot \vdash K' \sqsubseteq K}{(\varphi,\widetilde{i});\cdot;\Gamma_{sub},\widetilde{v} : \widetilde{T_{csub}} \vdash P \triangleleft K}}{\varphi;\cdot;!a(\widetilde{v}).P \triangleleft 0}$$

- If the typing is

$$\frac{\varphi;\Gamma \vdash v : \mathsf{Nat}[0,i] \quad \varphi;\Gamma \vdash P \triangleleft K_1 \quad \varphi;\cdot \vDash K_1 \leq K \quad \varphi;\Gamma\{i{+}1/i\}, x : \mathsf{Nat}[0,i] \vdash Q \triangleleft K_2 \quad \varphi;\cdot \vDash K_2 \leq K\{i{+}1/i\}}{\varphi;\Gamma \vdash \mathtt{match}\ v\ \{\underline{0} \mapsto P;;\ \mathtt{s}(x) \mapsto Q\} \triangleleft K}$$

  Then, we would like to give the following type:

$$\frac{\varphi;\cdot;\Gamma_{sub} \vdash e : \mathsf{Nat}[0,i] \qquad \varphi;(0 \leq 0);\Gamma_{sub} \vdash P \lhd K \qquad \varphi;(i \geq 1);\Gamma_{sub}, x : \mathsf{Nat}[0{-}1, i{-}1] \vdash Q \lhd K}{\varphi;\cdot;\Gamma_{sub} \vdash \mathtt{match}\ v\ \{\underline{0} \mapsto P;;\ \mathtt{s}(x) \mapsto Q\} \lhd K}$$

From the typing $\varphi;\Gamma \vdash P \lhd K_1$, obtaining $\varphi;(0 \leq 0);\Gamma_{sub} \vdash P \lhd K$ is direct by induction hypothesis. Now, by induction hypothesis we also obtain

$$\varphi;\cdot;\Gamma_{sub}\{i{+}1/i\}, x : \mathsf{Nat}[0,i] \vdash Q \lhd K_2 \qquad \varphi;\cdot \vDash K_2 \leq K\{i{+}1/i\}$$

Then, by index substitution, we have

$$\varphi;\cdot;\Gamma_{sub}\{i{+}1/i\}\{i{-}1/i\}, x : \mathsf{Nat}[0,i{-}1] \vdash Q \lhd K_2\{i{-}1/i\} \qquad \varphi;\cdot \vDash K_2\{i{-}1/i\} \leq K\{i{+}1/i\}\{i{-}1/i\}$$

Thus, by weakening we obtain:

$$\varphi;(i \geq 1);\Gamma_{sub}\{i{+}1/i\}\{i{-}1/i\}, x : \mathsf{Nat}[0,i{-}1] \vdash Q \lhd K_2\{i{-}1/i\}$$

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

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

$$\varphi;(i \geq 1);\Gamma_{sub}, x : \mathsf{Nat}[0,i{-}1] \vdash Q \lhd K$$

Thus, we can conclude the proof as the typing above can be obtained. The other case of pattern matching, where e has a type $\mathsf{Nat}[0, i+n+1]$, is easier.

With this, we covered all the important cases.

□

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

Still, with this intermediate type system, we can mimic the work in [6], and we obtain a procedure that is sound and complete. We have not explored yet a procedure that would work directly in our original type system and be complete with regard to this type system. Some details of the procedure and the proof are given in the Appendix A.1.

4.3 Parallel Complexity

4.3.1 Span with Annotated Processes

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

First, we want a parallel complexity that works as if we had an infinite number of processors. So, on the process $\mathtt{tick}.0 \mid \mathtt{tick}.0 \mid \mathtt{tick}.0 \mid \cdots \mid \mathtt{tick}.0$ we want the complexity to be 1, whatever the number of `tick` in parallel.

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

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

Remark 4.3.1. *A notion of complexity defined with* maximal progress*, as in [72], would not satisfy this condition. Indeed, maximal progress is usually defined as "when a communication is available, then we need to do this communication". In this previous example, this would impose the reduction path to P_1. We can also construct a similar example without* `tick`*, with* $b().a().P_0 \mid \overline{b}\langle\rangle \mid a().(c().P_1 \mid \overline{c}\langle\rangle) \mid \overline{a}\langle\rangle$

A possible way to define such a parallel complexity by using the literature would be to adapt causal complexity [45, 43, 42], however we believe there is a simpler presentation in our case. We will show at the end of this section the equivalence between our notion and a kind of causal complexity. The idea for defining span has been proposed by Naoki Kobayashi. It consists in introducing a new construction for processes, $m : P$, where m is an integer. A process using this constructor will be called an *annotated process*. Intuitively, this annotated process has the meaning P *with* m *ticks before*. We can then enrich the congruence relation $\equiv$ with the following rules:

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

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

With this congruence relation and this new constructor, we can give a new shape to the canonical form presented in Definition 4.1.1.

Definition 4.3.1 (Canonical Form for Annotated Processes)**.** *An annotated process is in canonical form if it has the shape:*

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

with $G_1, \ldots, G_m$ guarded annotated processes, defined as in Definition 4.1.1.

Notice that the congruence relation above allows to obtain this canonical form from any annotated processes. With this intuition in mind, we can then define a reduction relation $\Rightarrow$ for annotated processes. The rules are given in Figure 4.18. We do not detail the rules for integers as they are deducible from the ones for lists. Intuitively, this semantics works as the usual semantics for π-calculus, but when doing a synchronization, as we need both the input and the output to be ready, the synchronization can only happen after the maximum of the two annotations.

We then define the parallel complexity of an annotated process.

Definition 4.3.2 (Parallel Complexity)**.** *Let P be an annotated process. We define its* local complexity *$\mathcal{C}_\ell(P)$ by:*

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

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

$$\frac{}{\texttt{match } [] \ \{[] \mapsto P;; \ x :: y \mapsto Q\} \Rightarrow P}$$

$$\frac{}{\texttt{match } e :: e' \ \{[] \mapsto P;; \ x :: y \mapsto Q\} \Rightarrow Q[x, y := e, e']}$$

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

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

Figure 4.18: Reduction Rules

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

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

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

Lemma 4.3.1 (Congruence and Local Complexity)**.** *Let* P, Q *be annotated processes such that* $P \equiv Q$*. Then, we have* $\mathcal{C}_\ell(P) = \mathcal{C}_\ell(Q)$*.*

Lemma 4.3.2 (Reduction and Local Complexity)**.** *Let* P, P' *be annotated processes such that* $P \Rightarrow P'$*. Then, we have* $\mathcal{C}_\ell(P') \geq \mathcal{C}_\ell(P)$*.*

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

4.3.2 Span and Causal Complexity

In this section, we present how our notion of span can be linked with the causal complexity of the literature. This section is not mandatory to understand the typing systems for span, but we believe it can be of interest to show that both annotated processes and causal complexity are well-behaved notions of parallel complexity in the π-calculus.

Presentation of Causal Complexity

We present here a notion of causal complexity inspired by other works [45, 43, 42]. We explained before with the canonical form in Definition 4.1.1 that a process can be described by a set of names and a multiset of guarded processes, when working up to congruence. For causal

complexity, we consider more structure for processes. The idea is to see a process as a set of names and a binary tree where leaves are guarded processes and a node means parallel composition. So formally, instead of using the previous congruence relation, we use the *tree congruence*. This is defined as the least congruence relation $\equiv_t$ closed under:

$$(\nu a)(\nu b)P \equiv_t (\nu b)(\nu a)P$$

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

So, this tree congruence can indeed move names as before, but it preserves the tree-shape of a process. This time, the canonical form, initially described in Definition 4.1.1, has the shape $(\nu\widetilde{a})t$, where t is a tree as defined above. With this intuition, we can redefine the semantics in order to preserve this tree structure. The tree rules are described in Figure 4.19. It is supposed in this reduction rules that for the left-hand side, all the names (νa) are at the beginning of the process, as in the canonical form. So, for example, from a process P seen as a tree with at position p a replicated input $!a(\widetilde{v}).Q$ and at position p' an output $\overline{a}\langle\widetilde{e}\rangle$, we obtain the process P', the same tree as P, with at position p the same replicated input but at position p' we have the tree corresponding to the process $Q[\widetilde{v} := \widetilde{e}]$. This reduction step is annotated by a *location* $\tau\langle p, p'\rangle$ in order to remember at which positions the modification happened, and what was the action performed (here τ means a communication). In the same way, we can define a reduction annotated by a `tick`, or a match. An alternative presentation of this semantics, closer to the one in [45], with a labelled transition system, is also possible. As for the standard reduction in π-calculus, we could show that both semantics (the one with the congruence and the one with the labelled transition system) are equivalent. We work with the tree congruence definition because it is easier to use in proofs.

The goal of this semantics is first to preserve the tree structure in a reduction step, and second to remember when doing a reduction step where the modification occurs exactly in the tree. Then, we can define a causality relation between locations. The idea is that a location ℓ_1 *causes* a location ℓ_2 when the reduction step with location ℓ_2 could not have happened without the reduction step with location ℓ_1.

Formally, we define a location by the following grammar.

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

The intuition is that a reduction $P \to P'$ with location $p;\texttt{tick}$ removed the top `tick` of the guarded process of P at position p. A reduction $P \to P'$ with location $\tau\langle p, p'\rangle$ is a communication between the input process at position p and the output process at position p'. Finally, the action $p;\mathsf{if}$ is for the position of a pattern-matching reduction. Then, we can define a causality relation $\ell \prec_c \ell'$ between locations:

Definition 4.3.3. *The causality relation $\ell \prec_c \ell'$ between locations is defined by:*

- $p;\texttt{tick} \prec_c p';\texttt{tick}$ *when p is a prefix of p'*
- $p;\texttt{tick} \prec_c \tau\langle p_0, p_1\rangle$ *when p is a prefix of p_0 or p_1*
- $\tau\langle p_0, p_1\rangle \prec_c p;\texttt{tick}$ *when p_1 is a prefix of p*
- $\tau\langle p_0, p_1\rangle \prec_c \tau\langle p'_0, p'_1\rangle$ *when p_1 is a prefix of p'_0 or p'_1.*

By extension, we will sometimes say that a location $\ell \prec_c p$ when $\ell \prec_c p;\texttt{tick}$.

And for if locations, they behave as a `tick` location. The main interest of causal complexity is that this notion of causality can be adapted to account for different behaviours. For example, in [45], the causality relation is different. Here, we choose this causality relation to show the

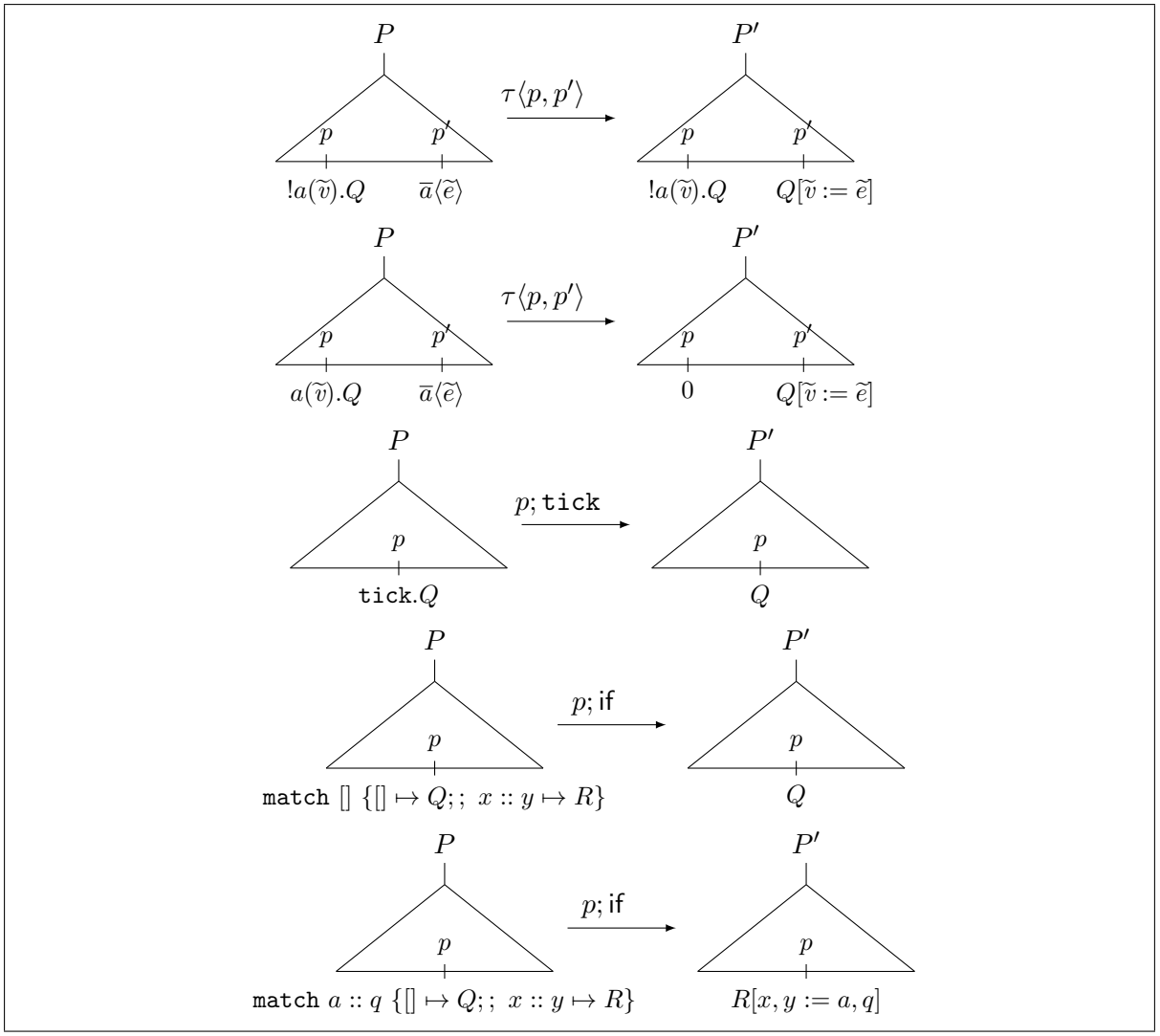

Figure 4.19: Semantics for Causal Complexity

equivalence with annotated processes, and in this sense, our notion of span is a particular case of causal complexity with this choice of causality relation.

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

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

Definition 4.3.4 (Computation)**.** *A* computation *from a process P is a sequence $(P_i, \ell_i)_{i \leq N}$ such that $P_0 = P$ and $P_i \xrightarrow{\ell_i} P_{i+1}$ for $i < N$.*

Definition 4.3.5 (Causal Complexity)**.** *In a computation $(P_i, \ell_i)_{i \leq N}$, we say that ℓ_i* depends *on ℓ_j, noted $\ell_i \prec \ell_j$ when $i < j$ and $\ell_i \prec_c \ell_j$. Then, the causal complexity of this computation is given by the maximal number of tick locations in all the chains of $\prec^\star$, the reflexive and*

transitive closure of $\prec$. Then, the causal complexity of a process P is defined as the maximal causal complexity over all computations from P.

Remark 4.3.2. *In the rules for synchronization in Figure 4.19, the continuation of the input is that an unusual position. Indeed, the continuation is put where the output was, instead of where the input was. We choose this presentation because this way, we obtain a simple statement for an exchange lemma that can be proved easily:*
"If $P_0 \xrightarrow{\ell} P_1 \xrightarrow{\ell'} P_2$ and $\ell \not\frown_c \ell'$, then there exists P_1' and P_2' with $P_0 \xrightarrow{\ell'} P_1' \xrightarrow{\ell} P_2'$ and $P_2 \equiv_t P_2'$ " With an alternative presentation where the continuation is put at the input position, we could have for example

$$!a(v).P \mid (\overline{a}\langle e\rangle \mid \overline{a}\langle e'\rangle) \to (!a(v).P \mid P[v := e]) \mid (0 \mid \overline{a}\langle e'\rangle) \to ((!a(v).P \mid P[v := e']) \mid P[v := e]) \mid (0 \mid 0)$$

And if we exchange the two reduction steps we obtain

$$!a(v).P \mid (\overline{a}\langle e\rangle \mid \overline{a}\langle e'\rangle) \to^2 ((!a(v).P \mid P[v := e]) \mid P[v := e']) \mid (0 \mid 0)$$

And so, the tree has been slightly modified, thus the exchange lemma is harder to state with tree congruence only. Moreover, as the shape of the tree changes, then the locations may also change, so a notion of trace equivalence may be needed.

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

Parallel Complexity $\geq$ Causal Complexity

In this section, we consider causal complexity, and we show that the parallel complexity is always greater than causal complexity. Formally, we prove the following lemma.

Lemma 4.3.3. *Let P be a process. Let $(P_i, \ell_i)_{i \leq N}$ be a computation from P with causal complexity K. Then, the global parallel complexity of P is greater than K.*

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

Definition 4.3.6 (Tree Representation of Annotated Processes)**.** *We consider annotated trees such that a set of names is given at the beginning, nodes represent parallel composition and leaves are processes of the shape $n : G$ with G guarded. Such a tree indeed represents an annotated process.*

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

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

Lemma 4.3.4 (Causal Reduction and Annotation)**.** *Suppose that $P \xrightarrow{\ell} Q$. Then, for any P' annotation of P, we have $P' \Rightarrow Q'$ where Q' is an annotation of Q.*

Moreover, we have:

- *If $\ell = p;\texttt{tick}$ then Q' has the same annotation as P' for all leaves at a position p' such that p is not a prefix of p'. For all the other leaves, Q' has the annotation $n + 1$ where n is the annotation for the leaf at position p in P'*

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

Proof. The proof is done by case analysis on the rules of Figure 4.19. All cases are rather direct. The idea is to always keep the same tree shape in the reduction $\Rightarrow$, and then to make the names go up and the annotations go down after doing this reduction using the congruence rules. Formally, the names can go up with $((\nu a)P) \mid Q \equiv (\nu a)(P \mid Q)$ (always possible by α-renaming) and $n : ((\nu a)P) \equiv (\nu a)(n : P)$, and then the annotations can go down with $n : (P \mid Q) \equiv n : P \mid n : Q$. Then, with the shape of the reduction $\Rightarrow$ we can indeed see that the annotations indeed correspond to the one given in this lemma. □

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

Lemma 4.3.5. *Let $(P'_i, \ell_i)_{i \leq N}$ be the computation given by the previous lemma from an original computation $(P_i, \ell_i)_{i \leq N}$. Then, for all $i \leq N$, the annotation of a guarded process G at position p in P'_i is an upper bound of the maximal number of* `tick` *locations in all chains of $\prec^\star$ for the locations $\ell_0, \ldots, \ell_{i-1}$ such that the last location ℓ of this chain satisfies $\ell \prec_c p$*

We prove this by induction on i.

- This is true for $i = 0$ since P' is P annotated with zeros everywhere.
- Let $i < N$. Suppose that this is true for P'_i, the annotation of P_i. Let us look at the reduction $P_i \xrightarrow{\ell_i} P_{i+1}$.
  - If $\ell_i = p; \mathsf{if}$. Then by induction hypothesis, the annotation n for the pattern matching bounds the maximal number of `tick` locations in all chains of $\prec^\star$ for the locations $\ell_0, \ldots, \ell_{i-1}$ such that the last location ℓ is such that $\ell \prec_c p$. Let us look at P'_{i+1} given by lemma 4.3.4. For all the positions in P'_{i+1} with p not a prefix, dependency did not change since $\ell_i = p; \mathsf{if}$ does not cause those positions. As annotations did not change either, the hypothesis is still correct. For the new positions in the tree with p as a prefix, all the annotations are n. Any chain of causality $\prec^\star$ with the locations $\ell_0, \ldots, \ell_i$ is either a chains that does not contain ℓ_i and that caused p, and so n is a bound by induction hypothesis, or it is a chain that contains ℓ_i and so it is a chain in causal relation with p, and so n is a bound.
  - If $\ell_i = p; \texttt{tick}$. Then by induction hypothesis, the annotation n for the tick corresponds to the maximal number of `tick` locations in all chains of $\prec^\star$ for the locations $\ell_0, \ldots, \ell_{i-1}$ that are also in causality $\prec_c$ with p. Let us look at P'_{i+1} given by lemma 4.3.4. For all the positions in P'_{i+1} with p not a prefix, dependency did not change since $\ell_i = p; \mathsf{if}$ does not cause those positions. As annotations did not change either, the hypothesis is still correct. For the new positions in the tree with p as a prefix, all the annotations are n. All chains of causality $\prec^\star$ with the locations $\ell_0, \ldots, \ell_i$ are either chains that do not contain ℓ_i and that caused p and so $n + 1$ is

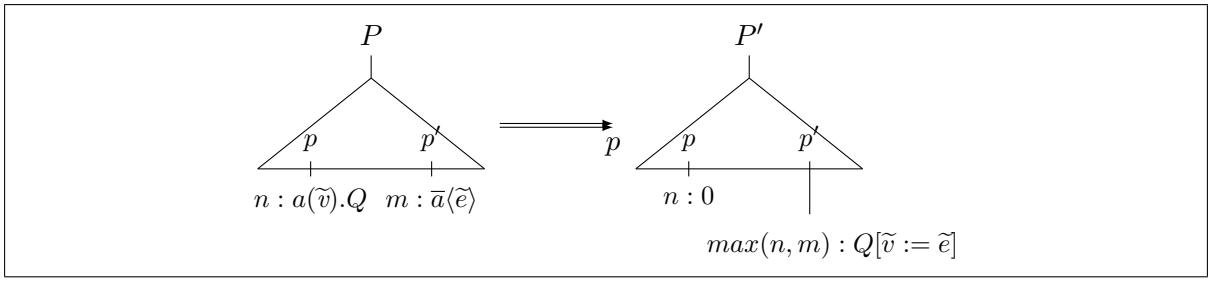

Figure 4.20: Tree Semantics for Span

a bound because n is a bound by induction hypothesis, either they contain ℓ_i and so in this case it was a chain in causal relation with p and so $n+1$ is a bound as n is a bound on the causality between ℓ_0 and ℓ_{i-1} and the last location ℓ_i adds one to the complexity.

– If $\ell_i = \tau\langle p, p'\rangle$. By induction hypothesis, the annotation n for the input and m for the output are bounds on some chains of causality on $\ell_0, \ldots, \ell_{i-1}$. Let us look at P'_{i+1} given by Lemma 4.3.4. For all positions that are not a prefix of p', nothing changed so the hypothesis is still true. Let us look at positions with p' as a prefix. All the annotations for those positions are $\max(n, m)$. Let us look at chains of causality on $\ell_0, \ldots, \ell_i$ that end with a location that causes those positions. The new chains that were not in the previous hypothesis are the ones that finish with ℓ_i. For those chains, either they cause ℓ_i because they cause p or because they cause p'. In both cases, n or m was a bound on the number of ticks by induction hypothesis. So, $\max(n, m)$ is indeed a bound on the number of ticks for all those chains.

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

Causal Complexity $\geq$ Parallel Complexity

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

$$(\nu a)(\nu b)P \equiv_t (\nu b)(\nu a)P$$

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

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

And then we define the semantics $\Rightarrow$ exactly as before but with trees instead of simple processes in parallel. An example is given in Figure 4.20

As before, any annotated process can be written in a tree representation as in Definition 4.3.6 using the tree congruence rule $\equiv_t$ for annotated processes. So, from this, it is rather direct that this semantics defined with tree $\Rightarrow$ is equivalent to the previously defined $\Rightarrow$, in the sense that they give the same complexity. (It relies in particular on the fact that congruence does not change the parallel complexity). And now, we can work on this parallel complexity with tree representation. Formally, we want to prove:

Lemma 4.3.6. *If P is a process without annotation, with $P(\Rightarrow)^*Q$ and $\mathcal{C}_\ell(Q) = K$, then there is a computation $(P_i, \ell_i)_{i\leq N}$ from P with causal complexity greater than K.*

So, by definition of causal complexity and parallel complexity, this will indeed show that the causal complexity is greater than the parallel complexity.

As we only need to prove this lemma for processes without annotation, we can take an additional hypothesis on annotated processes: we consider in the following that annotations appear exclusively at the "top-level" of a process, so there is no annotation in the subprocess P in $a(\widetilde{v}).P$, in $\texttt{tick}.P$, in $!a(\widetilde{v}).P$ or in the subprocesses of a pattern matching. We can take this hypothesis because if a process P satisfies this hypothesis and $P \Rightarrow Q$ then Q also satisfies this hypothesis. And of course, processes without annotation satisfy this hypothesis. Moreover, in this definition, we do not consider $0 : P$ as a real annotation since it can be removed by congruence. We start by the following definition.

Definition 4.3.7 (Removing Annotation)**.** *Let P be a tree representation of an annotated process. We define $\mathsf{forget}(P)$ as the tree P where leaves $n : G$ are replaced by G.*

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

Lemma 4.3.7 (Forget Reductions)**.** *Suppose that $P \Rightarrow Q$. Then, we have $\mathsf{forget}(P) \xrightarrow{\ell} \mathsf{forget}(Q)$ for some location ℓ.*

Proof. The proof is direct because we modified $\Rightarrow$ in order to obtain immediately this. Note that the reduction $\texttt{tick}.P \Rightarrow 1 : P$, or more generally $n : \texttt{tick}.P \Rightarrow (n+1) : P$ here corresponds indeed to removing a tick with the forgetful function. □

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

Lemma 4.3.8. *If P is a process without annotation, and if $P(\Rightarrow)^*Q$, then there is a computation $(P_i, \ell_i)_{i\leq N}$ from $P = \mathsf{forget}(P)$ to $\mathsf{forget}(Q)$. Moreover, for each leaf $n : G$ at position p in Q, there is a chain of causality $\prec^\star$ with at least n ticks that ends with a location ℓ such that $\ell \prec_c p$.*

Note that from this lemma we can deduce immediately Lemma 4.3.6. We prove this by induction on $P(\Rightarrow)^*Q$

- If this relation is the reflexive one, and $P = Q$, this is direct because all annotations are equal to 0.
- Now suppose we have $P(\Rightarrow)^*R \Rightarrow Q$. By induction hypothesis, there is a computation $(P_i, \ell_i)_{i\leq N}$ from $\mathsf{forget}(P)$ to $\mathsf{forget}(R)$, with the expected chains of causality. We now proceed by case analysis on $R \Rightarrow Q$.
  - If this reduction is a pattern matching reduction at position p. Then, we have $\mathsf{forget}(R) \xrightarrow{p;\mathsf{if}} \mathsf{forget}(Q)$. Let us take a leaf $n : G$ at position p' of Q. If p is not a prefix of p', then this leaf was also in R. So, by induction hypothesis, we obtain the desired chain of causality. If p is a prefix of p', then this n was also the annotation for the position p in R. So, by induction hypothesis for R, there is a chain of causality $\prec^\star$ with at least n ticks that ends with a location ℓ such that $\ell \prec_c p$. By definition, it means that this last location $\ell \prec_c p;\mathsf{if}$. So, this gives us a chain of causality with at least n ticks that ends with a location that cause p' in Q. This concludes this case.

– If this reduction is a tick reduction at position p. Then, we have $\mathsf{forget}(R) \xrightarrow{p;\mathtt{tick}} \mathsf{forget}(Q)$. Let us take a leaf $n : G$ at position p' of Q. If p is not a prefix of p', then this leaf was also in R. So, by induction hypothesis, we obtain the desired chain of causality. If p is a prefix of p', then $n-1$ was the annotation for the position p in R. So, by induction hypothesis for R, there is a chain of causality $\prec^\star$ with at least $n-1$ ticks that ends with a location ℓ such that $\ell \prec_c p$. By definition, it means that this last location $\ell \prec_c p;\mathtt{tick}$. So, this gives us a chain of causality with at least n ticks that ends with a location that causes p' in Q. This concludes this case.

– If this reduction is a synchronization with input at position p and output at position p'. Then, we have $\mathsf{forget}(R) \xrightarrow{\tau\langle p,p'\rangle} \mathsf{forget}(Q)$. Let us take a leaf $n : G$ at position q of Q. If p' is not a prefix of q, then this leaf was also in R (except for the position p in the case of non-replicated input where the guarded process changes but not the annotation, still the following reasoning works). So, by induction hypothesis, we obtain the desired chain of causality. If p' is a prefix of q, then, we have $n = \max(n_0, n_1)$ with n_0 the annotation for the input position p in R and n_1 the annotation for the output position p' in R. Let us say, by symmetry, that n_0 is the maximum between those two. So, by induction hypothesis for R, there is a chain of causality $\prec^\star$ with at least n_0 ticks that ends with a location ℓ such that $\ell \prec_c p$. By definition, it means that this last location $\ell \prec_c \tau\langle p, p'\rangle$. So, this gives us a chain of causality with at least n_0 ticks that ends with a location that causes q in Q. This concludes this case.

This concludes the proof.

So, we have indeed that causal complexity is greater than parallel complexity.

From this, we have the equivalence between causal complexity and our definition of parallel complexity.

4.4 Types for Span

We present here a type system for span, so we want as previously a type system such that typing a process gives us a bound on its span. Formally, we will prove the following theorem:

Theorem 4.4.1 (Typing and Complexity)**.** *Let P be a process and m be its global parallel complexity. If we have $\varphi; \Phi; \Gamma \vdash P \triangleleft K$, then $\varphi; \Phi \vDash K \geq m$.*

Notice that this theorem talks about open processes. However, our notion of complexity does not behave well with open processes, similarly to what happened for Theorem 3.1.2. For example the process $\mathtt{match}\ v\ \{\underline{0} \mapsto P;;\ \mathtt{s}(x) \mapsto Q\}$ is in normal form for a variable v, so this process has global complexity 0. Still, we will also obtain the following corollary:

Corollary 4.4.1 (Complexity and Open Processes)**.** *We have:*

- *If $\varphi; \Phi; \Gamma, \widetilde{v} : \widetilde{T} \vdash P \triangleleft K$, then for any sequence of expressions $\widetilde{e}$ such that $\varphi; \Phi; \Gamma \vdash \widetilde{e} : \widetilde{T}$, K is a bound on the global complexity of $P[\widetilde{v} := \widetilde{e}]$*
- *If $\varphi; \Phi; \Gamma \vdash P \triangleleft K$, then for any other annotated process Q such that $\varphi; \Phi; \Gamma \vdash Q \triangleleft K'$, $max(K, K')$ is a bound on the global complexity of $P \mid Q$.*

So, when we have a typing $\varphi; \Phi; \Gamma \vdash P \triangleleft K$ for an open process, one should not see K as a bound on the actual complexity on P, but it should be seen as a bound on the complexity of

this particular process in an environment respecting the type of Γ. So, in $\varphi; \Phi; v : \mathsf{Nat}[2, 10] \vdash \mathtt{match}\ v\ \{\underline{0} \mapsto P; ;\ \mathtt{s}(x) \mapsto Q\} \triangleleft K$, this index K is a bound on the complexity of this pattern matching under the assumption that the environment gives to v an integer value between 2 and 10.

4.4.1 Sized Types with Time

The type system is an extension of the previous one for work. In order to take into account parallelism, we need a way to synchronize the time between processes in parallel, thus we will add some time information in types, as in [72] or [36].

Definition 4.4.1. *The set of types and* base types *are given by the grammar:*

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

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

As before, we define a notion of subtyping on those types. The rules are essentially the same as the ones in Figures 4.8 and 4.9. The only difference is that we force the time of a type to be invariant in subtyping.

In order to write the typing rules, we need some other definitions to work with time in types. The first thing we need is a way to advance time.

Definition 4.4.2 (Advancing Time in Types)**.** *Given a set of index variables φ, a set of constraints Φ, a type T and an index I, we define T after I time units, denoted $\langle T\rangle_{-I}^{\varphi;\Phi}$ by:*

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

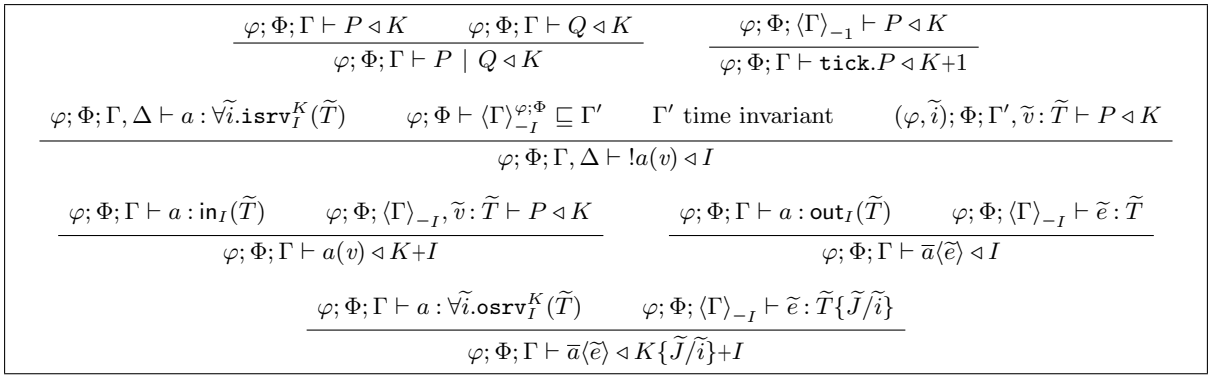

Figure 4.21: Span Typing Rules for Processes

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

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

For servers types, there is a dissymmetry between the input and output capabilities. The input capability behaves like a simple channel, if the time J is too small, we erase this capability. However, the output capability is never erased, even if the time J is too small. This is because when a server is defined, it must stay available until the end. Thus, an output to a server should always be possible, no matter the time. Still, the input capability of a server should not be available eternally, as the time J is supposed to mean the time for which a replicated input is effectively defined. So, when this time has passed, we should not be able to define a replicated input any more.

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

Definition 4.4.3 (Time Invariant Context)**.** *Given a set of index variables* φ *and a set of constraints* Φ*, a context* Γ *is said to be* time invariant *when it only contains base type variables or output server types* $\forall \widetilde{i}.\mathtt{osrv}_I^K(\widetilde{T})$ *with* $\varphi; \Phi \vDash I = 0$*.*

Such a context is thus invariant by the operator $\langle \cdot \rangle_{-I}$ for any I. This is typically the kind of context that we need to define a server, as a server should not be dependent on the time it is called. We can now present the type system. Typing rules for expressions and some processes do not change, they can be found in Figure 4.10 and Figure 4.11. In Figure 4.21, we present the remaining rules in this type system that differ from the ones in Figure 4.12. As before, a typing judgement $\varphi; \Phi; \Gamma \vdash P \triangleleft K$ intuitively means that under the constraints Φ, in a context Γ, a process P is typable and its span complexity is bounded by K.

The rule for parallel composition shows that we consider parallel complexity as we take the maximum between the two processes instead of the sum. In practice, we ask for the same complexity K in both branches of parallel composition, but with the subtyping rule, it corresponds

indeed to the maximum. For input server, we integrate some weakening on context (Δ), and we want a time invariant context to type the server, as a server should not depend on time. Weakening is important since some types are not time invariant, such as channels. So, we need to separate time invariant types that can be used in the continuation P from other types.

Some rules make the time advance in their continuation, for example the tick rule or input rule. This is expressed by the advance time operator on contexts, and because time advances, the complexity also increases. Also, notice that because of the advance of time, some channels names could disappear, thus there is a kind of "time uniqueness" for channels, contrary to the previous section on work. This will be detailed later. Also, note that in the rule for replicated input, there is an explicit subtyping in the premises. This is because $\langle\Gamma\rangle^{\varphi;\Phi}_{-I}$ is not time invariant in general, since the type of a contains the input capability. However, if this server has both input and output capabilities, we can give a time invariant type for a (or other servers) just by removing the input capability, which can be done by subtyping.

Looking back at Corollary 4.4.1, we can for example understand the rule for input by taking the judgement $\varphi;\Phi;a:\mathsf{ch}_3() \vdash a().\mathtt{tick}.0 \triangleleft 4$. This expresses that with an environment providing a message on a within 3 time units, this process terminates in 4 time units.

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

Definition 4.4.4 (Forgetting Sizes)**.** *Formally, given a sized type T, we define $\mathcal{U}(T)$ the usual input/output type ($\mathcal{U}$ is for forgetful) by:*

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

$$\mathcal{U}(\mathsf{ch}_I(\widetilde{T})) := \mathit{ch}(\mathcal{U}(\widetilde{T})) \qquad \mathcal{U}(\mathsf{in}_I(\widetilde{T})) := \mathit{in}(\mathcal{U}(\widetilde{T})) \qquad \mathcal{U}(\mathsf{out}_I(\widetilde{T})) := \mathit{out}(\mathcal{U}(\widetilde{T}))$$

$$\mathcal{U}(\forall\widetilde{i}.\mathit{srv}^K_I(\widetilde{T})) := \mathit{ch}(\mathcal{U}(\widetilde{T})) \qquad \mathcal{U}(\forall\widetilde{i}.\mathit{isrv}^K_I(\widetilde{T})) := \mathit{in}(\mathcal{U}(\widetilde{T})) \qquad \mathcal{U}(\forall\widetilde{i}.\mathit{osrv}^K_I(\widetilde{T})) := \mathit{out}(\mathcal{U}(\widetilde{T}))$$

Then, we obtain the following lemma, with the classical input/output typing:

Lemma 4.4.1. *If $\varphi;\Phi \vdash T \sqsubseteq T'$ then $\mathcal{U}(T) \sqsubseteq \mathcal{U}(T')$. Moreover, if $\varphi;\Phi;\Gamma \vdash e : T$ then $\mathcal{U}(\Gamma) \vdash e : \mathcal{U}(T)$ and if $\varphi;\Phi;\Gamma \vdash P \triangleleft K$ then $\mathcal{U}(\Gamma) \vdash P$*

Examples

Input, Time and Complexity. We first illustrate the type system on a simple generic example that will be very useful for other concrete examples.

Example 4.4.1. *Let us consider the process $P = c().b().\overline{a}\langle\rangle$. We have the following typing for P:*

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

if and only if $\varphi;\Phi \vDash I_a \geq I_b \geq I_c$.

Indeed, the typing derivation has the shape:

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

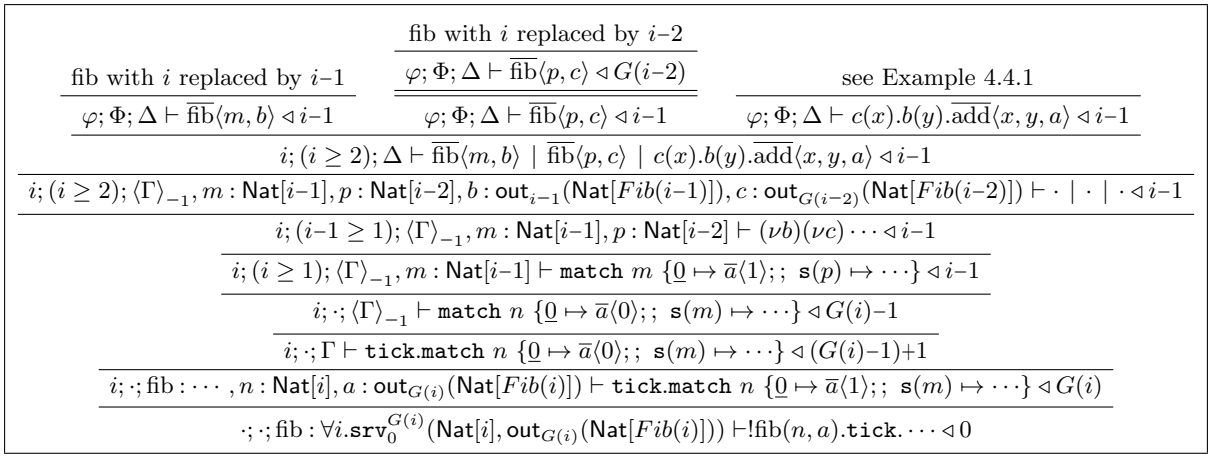

Figure 4.22: A Typing for Fibonacci (Span)

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

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

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

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

Fibonacci. Let us consider again the process P described in Figure 4.3. We pose $G(n) = \max(1,n)$ and we want to show that the server fib has a span $G(n)$. In order to do this, we give the typing described in Figure 4.22.

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

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

for the following definitions of Q_l:

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

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

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

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

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

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

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

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

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

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

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

The first constraint is because the total complexity $f(i){-}1$ must be greater than the complexity of the two recursive calls $f(i{-}1)$. Then, r' and r'' must have a time equal to $g(i{-}1)$ in order to correspond to the type of a in the outputs $\overline{a}\langle m, r'\rangle$ and $\overline{a}\langle m, r''\rangle$. Finally, the last two constraints correspond to the constraints of Example 4.4.1 and the fact that the complexity bound $f(i){-}1$ should be greater that the complexity of this subprocess with is $g(i){-}1$ again as in Example 4.4.1. So, we can satisfy the conditions with the following choice:

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

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

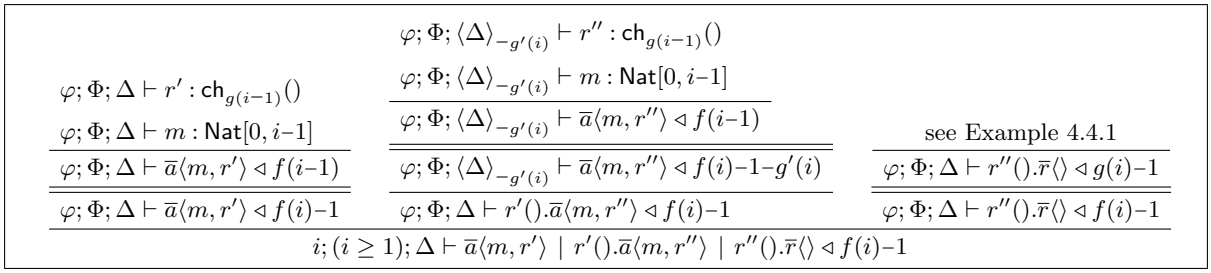

Figure 4.23: A Typing for Q_2

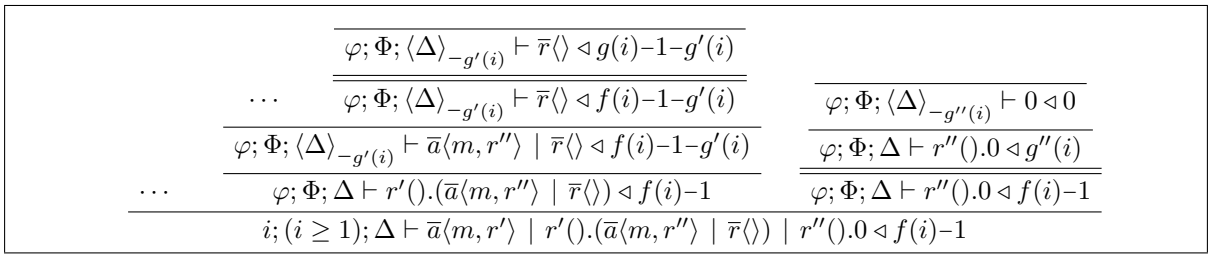

Figure 4.24: A Typing for Q_3

Then, for Q_2, we describe the typing in Figure 4.23, where $\varphi;\Phi \equiv i;(i \geq 1)$.

Thus, we obtain the following constraints:

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

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

The constraints on $f(i)$ are obtained because of the subtyping rules, expressed by the double lines in the type derivation. The equality constraints are obtained because we need both r' and r'' to have the type $\mathsf{ch}_{g(i-1)}()$, and finally the constraint $g(i){-}1 \geq g''(i)$ is as in Example 4.4.1. Thus, we can take:

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

So, we indeed obtain the exponential complexity.

However, with those two examples, the time of the channel r is always equal to the complexity of the server a, so we cannot really see the usefulness of time. Still, with the next example we obtain something more interesting. A type derivation for Q_3 is given in Figure 4.24.

And we obtain the constraints:

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

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

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

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

Thus, we can take:

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

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

Note that the two first examples used channels as a return signal for a parallel computation, whereas for the last example, channels are used as a synchronization point in the middle of a computation. We believe that this flexibility of channels justifies the use of π-calculus to reason about parallel computation. Moreover, this work is a step to a more expressive type system inspired by [72], taking in account concurrent behaviour, that we will present in Section 4.5. Indeed, as we will show, the current type system fails to capture some simple concurrency.

Limitations of the Type System. Our current type system enforces some kind of time uniqueness in channels.

Example 4.4.2 (Time Uniqueness)**.** *Indeed, take the process* $a().\texttt{tick}.\overline{a}\langle\rangle$*. When trying to type this process, we obtain:*

$$\dfrac{\dfrac{\cdot;\cdot \vdash \mathsf{ch}_I() \sqsubseteq \mathsf{in}_I()}{\cdot;\cdot;a:\mathsf{ch}_I() \vdash a:\mathsf{in}_I()} \qquad \dfrac{\textit{Error}}{\cdot;\cdot;\langle a:\mathsf{ch}_0()\rangle_{-1} \vdash \overline{a}\langle\rangle \triangleleft 0}}{\dfrac{}{\cdot;\cdot;a:\mathsf{ch}_I() \vdash a().\texttt{tick}.\overline{a}\langle\rangle \triangleleft I+1}}$$

As by definition $\langle a:\mathsf{ch}_0()\rangle_{-1}$ *is* $\emptyset$*, we cannot type the output on* a*.*

So, channels have strong constraints on the time they can be used. This is true especially when channels are not used linearly. Still, note that we can type a process of the shape $a().0 \mid \overline{a}\langle\rangle \mid \texttt{tick}.\overline{a}\langle\rangle$, so it is better than plain linearity on channels. This restriction limits examples of concurrent behaviours. For example, take two processes P_1 and P_2 that should be executed but not simultaneously. In order to do that in a concurrent setting, we can use semaphores. In π-calculus, we could consider the process $(\nu a)(a().P_1' \mid a().P_2' \mid \overline{a}\langle\rangle)$, where P_1' is P_1 with an output $\overline{a}\langle\rangle$ at the end, likewise for P_2'. This is a way to simulate semaphore in π-calculus. Now, we can see that this example has the same problem as the example given above if for example P_1 contains a `tick`, thus we cannot type this kind of processes. Formally, this is because of our parallel composition rule:

$$\dfrac{\varphi;\Phi;\Gamma \vdash P \triangleleft K \qquad \varphi;\Phi;\Gamma \vdash Q \triangleleft K}{\varphi;\Phi;\Gamma \vdash P \mid Q \triangleleft K}$$

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

Still, we believe that for parallel computation, our type system should be quite expressive in practice. Indeed, as stated above, the restriction appears especially when channels are not used linearly. However, it is known that linear π-calculus in itself is expressive for parallel computation [76]. For example, classical encodings of functional programs in a parallel setting rely on the use of linear return signals, as we will see in the example for bitonic sort in Sect. 4.4.3. Moreover, session types can also be encoded in linear π-calculus in the presence of variant types [73, 35]. Note that in order to encode a calculus as the one in [36], we would also need recursive types. Our calculus and its proof of soundness could be extended to variant types,

but not straightforwardly to recursive types. However, we believe the results on the linear π-calculus we cited suggest that the restriction given above should not be too harmful for parallel computation.

We will come back later on this example, in Section 4.5, and we will use type usages to overcome this problem.

4.4.2 Soundness

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

$$\frac{\varphi; \Phi; \langle \Gamma \rangle_{-n} \vdash P \triangleleft K}{\varphi; \Phi; \Gamma \vdash n : P \triangleleft K{+}n}$$

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

The procedure to prove the subject reduction for $\Rightarrow$ in this type system is intrinsically more difficult than the one for Theorem 4.2.1. So, from the proof of subject reduction for span, one could deduce the proof of subject reduction for work, just by forgetting the considerations with time and the constructor $n : P$ in the following proof.

Intermediate Lemmas

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

Lemma 4.4.2. *If $\varphi; \Phi \vdash T \sqsubseteq U$ then for any I, either $\langle U \rangle_{-I}$ is undefined, or both $\langle U \rangle_{-I}$ and $\langle T \rangle_{-I}$ are defined, and $\varphi; \Phi \vdash \langle T \rangle_{-I} \sqsubseteq \langle U \rangle_{-I}$.*

Proof. The proof is by induction on the subtyping derivation. First, transitivity is direct by induction hypothesis. For non-channel type, this is direct because time advancing does not do anything. Then, for channels that are not servers, time is invariant by subtyping, thus time advancing is either undefined for both types, either defined for both types, with the same time. For a server type, again time is invariant by subtyping, so either both types lose their input capability, either they both keep the same capabilities. The case where $\langle U \rangle_{-I}$ is undefined and not $\langle T \rangle_{-I}$ is when T is an input/output server that looses its input capability, and U is an input server. □

Now, for the usual properties of typing systems, we have first structure lemmas.

Lemma 4.4.3 (Weakening)**.** *Let φ, φ' be disjoint set of index variables, Φ be a set of constraints on φ, Φ' be a set of constraints on (φ, φ'), Γ and Γ' be contexts on disjoint set of variables.*

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

Proof. Point 1 is the same as in Lemma 3.1.1, it comes from the definition of indexes. Point 2 is proved by induction on the subtyping derivation, and it uses explicitly Point 1. Point 4 is a consequence of Point 1: everything that is defined in $\langle\Gamma\rangle_{-I}^{\phi;\Phi}$ is also defined in $\langle\Gamma\rangle_{-I}^{(\phi,\phi');(\Phi,\Phi')}$, and the subtyping condition is here since with more constraints, a server may not be changed into an output server by the advance of time. Point 3 and Point 5 are proved by induction on the typing derivation, and each point uses crucially the previous ones. Note that the weakening integrated in the rule for input servers is necessary to obtain Point 5. Note also that when the advance time operator is used, the weakened typing is obtained with the use of a subtyping rule. □

Lemma 4.4.4 (Strengthening)**.** *Let φ be a set of index variables, Φ be a set of constraint on φ, and C a constraint on φ such that $\varphi;\Phi \vDash C$.*

1. *If $\varphi;(\Phi,C) \vDash C'$ then $\varphi;\Phi \vDash C'$.*
2. *If $\varphi;(\Phi,C) \vdash T \sqsubseteq U$ then $\varphi;\Phi \vdash T \sqsubseteq U$.*
3. *If $\varphi;(\Phi,C);\Gamma,\Gamma' \vdash e:T$ and the variables in Γ' are not free in e, then $\varphi;\Phi;\Gamma \vdash e:T$.*
4. $\langle\Gamma\rangle_{-I}^{\varphi;(\Phi,C)} = \langle\Gamma\rangle_{-I}^{\varphi;\Phi}$.
5. *If $\varphi;(\Phi,C);\Gamma,\Gamma' \vdash P \triangleleft K$ and the variables in Γ' are not free in P, then $\varphi;\Phi;\Gamma \vdash P \triangleleft K$.*

Proof. Point 1 is a direct consequence of the definition, as in Lemma 3.1.2. Point 2 is proved by induction on the subtyping derivation. Point 4 is straightforward with Point 1 of this lemma and Point 1 of Lemma 4.4.3. Point 3 and Point 5 are proved by induction on the typing derivation. □

We also have the usual property specific to our sized type systems, index substitution.

Lemma 4.4.5 (Index Substitution)**.** *Let φ be a set of index variable and $i \notin \varphi$. Let J be an index with free variables in φ. Then,*

1. $[\![I\{J/i\}]\!]_\rho = [\![I]\!]_{\rho[i\mapsto[\![J]\!]_\rho]}$.
2. *If $(\varphi,i);\Phi \vdash C$ then $\varphi;\Phi\{J/i\} \vDash C\{J/i\}$.*
3. *If $(\varphi,i);\Phi \vdash T \sqsubseteq U$ then $\varphi;\Phi\{J/i\} \vdash T\{J/i\} \sqsubseteq U\{J/i\}$.*
4. *If $(\varphi,i);\Phi;\Gamma \vdash e:T$ then $\varphi;\Phi\{J/i\};\Gamma\{J/i\} \vdash e:T\{J/i\}$.*
5. $\langle\Gamma\{J/i\}\rangle_{-I\{J/i\}}^{\varphi;\Phi\{J/i\}} = \Delta,\Delta'$ *with* $\varphi;\Phi\{J/i\} \vdash \Delta \sqsubseteq (\langle\Gamma\rangle_{-I}^{(\varphi,i);\Phi})\{J/i\}$.
6. *If $(\varphi,i);\Phi;\Gamma \vdash P \triangleleft K$ then $\varphi;\Phi\{J/i\};\Gamma\{J/i\} \vdash P \triangleleft K\{J/i\}$.*

Proof. Again, Point 1 and Point 2 are some properties in indices, that were already stated in Lemma 3.1.3. Point 3 is proved by induction on the subtyping derivation, then Point 4 is proved by induction on the typing derivation. Point 5 is direct with the use of Point 2. And finally Point 6 is proved by induction on P. The induction is on P and not the typing derivation because of Point 5 that forces the use of weakening, which is admissible but not derivable. (Lemma 4.4.3). □

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

Definition 4.4.5 (Delaying)**.** *Given a type T and an index I, we define the* delaying *of T by I units of time, denoted $\langle T\rangle_{+I}$:*

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

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

Lemma 4.4.6 (Delaying)**.** *For any index I, we have:*

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

Proof. Point 1, Point 2, Point 3 and Point 4 are straightforward. Then, Point 5 is proved by induction on P. Point 4 is used on every rule for channel or servers, and Point 3 is used in the rule for tick. □

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

Lemma 4.4.7 (Substitution)**.** *We have:*

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

The proof is pretty straightforward.

Subject Reduction

We now present the core theorem in order to obtain the complexity soundness: subject reduction. First, we need to show that tying is invariant by congruence.

Lemma 4.4.8 (Congruence and Typing)**.** *Let P and Q be annotated processes such that $P \equiv Q$. Then, $\varphi;\Phi;\Gamma \vdash P \triangleleft K$ if and only if $\varphi;\Phi;\Gamma \vdash Q \triangleleft K$.*

Proof. We prove this by induction on $P \equiv Q$. Notice that again, for a process P, the typing system is not syntax-directed because of the subtyping rule, so we consider that a derivation has exactly one subtyping rule before any syntax-directed rule. And, as for Theorem 3.1.1, the first subtyping rule on the bottom can be ignored. We first show this property for base case of congruence. The reflexivity is trivial then we have:

- **Case** $P \mid 0 \equiv P$. Suppose $\varphi;\Phi;\Gamma \vdash P \mid 0 \triangleleft K$. Then the derivation has the shape:

$$\dfrac{\dfrac{\pi}{\varphi;\Phi;\Gamma \vdash P \triangleleft K} \qquad \dfrac{\pi'}{\varphi;\Phi;\Gamma \vdash 0 \triangleleft K}}{\varphi;\Phi;\Gamma \vdash P \mid 0 \triangleleft K}$$

Thus, π gives us the desired proof. Reciprocally, given a derivation π of $\varphi; \Phi; \Gamma \vdash P \triangleleft K$, we can derive the proof:

$$\dfrac{\dfrac{\pi}{\varphi; \Phi; \Gamma \vdash P \triangleleft K} \qquad \dfrac{\dfrac{}{\varphi; \Phi; \Gamma \vdash 0 \triangleleft 0} \qquad \varphi; \Phi \vDash 0 \leq K}{\varphi; \Phi; \Gamma \vdash 0 \triangleleft K}}{\varphi; \Phi; \Gamma \vdash P \mid 0 \triangleleft K}$$

- **Case** $P \mid Q \equiv Q \mid P$. Suppose $\varphi; \Phi; \Gamma \vdash P \mid Q \triangleleft K$. Then the derivation has the shape:

$$\dfrac{\dfrac{\pi}{\varphi; \Phi; \Gamma \vdash P \triangleleft K} \qquad \dfrac{\pi'}{\varphi; \Phi; \Gamma \vdash Q \triangleleft K}}{\varphi; \Phi; \Gamma \vdash P \mid Q \triangleleft K}$$

And so we can derive:

$$\dfrac{\dfrac{\pi'}{\varphi; \Phi; \Gamma \vdash Q \triangleleft K} \qquad \dfrac{\pi}{\varphi; \Phi; \Gamma \vdash P \triangleleft K}}{\varphi; \Phi; \Gamma \vdash Q \mid P \triangleleft K}$$

- **Case** $P \mid (Q \mid R) \equiv (P \mid Q) \mid R$. Suppose $\varphi; \Phi; \Gamma \vdash P \mid (Q \mid R) \triangleleft K$. Then the derivation has the shape:

$$\dfrac{\dfrac{\pi_P}{\varphi; \Phi; \Gamma \vdash P \triangleleft K} \qquad \dfrac{\dfrac{\dfrac{\pi_Q}{\varphi; \Phi; \Delta \vdash Q \triangleleft K'} \qquad \dfrac{\pi_R}{\varphi; \Phi; \Delta \vdash R \triangleleft K'}}{\varphi; \Phi; \Delta \vdash Q \mid R \triangleleft K'} \qquad \varphi; \Phi \vdash \Gamma \sqsubseteq \Delta \qquad \varphi; \Phi \vDash K' \leq K}{\varphi; \Phi; \Gamma \vdash Q \mid R \triangleleft K}}{\varphi; \Phi; \Gamma \vdash P \mid (Q \mid R) \triangleleft K}$$

We can derive the proof:

$$\dfrac{\dfrac{\dfrac{\pi_P}{\varphi; \Phi; \Gamma \vdash P \triangleleft K} \qquad \dfrac{\dfrac{\pi_Q}{\varphi; \Phi; \Delta \vdash Q \triangleleft K'} \qquad \varphi; \Phi \vdash \Gamma \sqsubseteq \Delta; K' \leq K}{\varphi; \Phi; \Gamma \vdash Q \triangleleft K}}{\varphi; \Phi; \Gamma \vdash P \mid Q \triangleleft K} \qquad \dfrac{\dfrac{\pi_R}{\varphi; \Phi; \Delta \vdash R \triangleleft K'}}{\varphi; \Phi; \Gamma \vdash R \triangleleft K}}{\varphi; \Phi; \Gamma \vdash (P \mid Q) \mid R \triangleleft K}$$

The reverse follows the same pattern.

- **Case** $(\nu a)(\nu b)P \equiv (\nu b)(\nu a)P$. Suppose $\varphi; \Phi; \Gamma \vdash (\nu a)(\nu b)P \triangleleft K$. Then the derivation has the shape:

$$\dfrac{\dfrac{\dfrac{\dfrac{\pi}{\varphi; \Phi; \Delta, a : T', b : U \vdash P \triangleleft K'}}{\varphi; \Phi; \Delta, a : T' \vdash (\nu b)P \triangleleft K'} \qquad \varphi; \Phi \vdash \Gamma \sqsubseteq \Delta \qquad \varphi; \Phi \vDash K' \leq K \qquad \varphi; \Phi \vdash T \sqsubseteq T'}{\varphi; \Phi; \Gamma, a : T \vdash (\nu b)P \triangleleft K}}{\varphi; \Phi; \Gamma \vdash (\nu a)(\nu b)P \triangleleft K}$$

We can derive the proof:

$$\dfrac{\dfrac{\dfrac{\dfrac{\pi}{\varphi;\Phi;\Delta,a:T',b:U\vdash P\triangleleft K'}\quad \varphi;\Phi\vdash\Gamma\sqsubseteq\Delta\quad \varphi;\Phi\vDash K'\leq K\quad \varphi;\Phi\vdash T\sqsubseteq T'\quad \varphi;\Phi\vdash U\sqsubseteq U}{\varphi;\Phi;\Gamma,a:T,b:U\vdash P\triangleleft K}}{\varphi;\Phi;\Gamma,b:U\vdash(\nu a)P\triangleleft K}}{\varphi;\Phi;\Gamma\vdash(\nu b)(\nu a)P\triangleleft K}$$

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

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

  By weakening (Lemma 4.4.3), we obtain a derivation π'_Q of $\varphi;\Phi;\Gamma,a:T\vdash Q\triangleleft K$. Thus, we have the following derivation:

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

  For the converse, suppose $\varphi;\Phi;\Gamma\vdash(\nu a)(P\mid Q)\triangleleft K$. Then the derivation has the shape:

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

  Since a is not free in Q, by Lemma 4.4.4, from π_Q we obtain a derivation π'_Q of $\varphi;\Phi;\Delta\vdash Q\triangleleft K'$. We can then derive the following typing:

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

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

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

So, we can give the following derivation:

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

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

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

By Lemma 4.4.2, from $\Gamma\sqsubseteq\Delta_1$, we obtain $\Gamma=\Theta_0,\Theta_1$ with $\langle\Theta_1\rangle_{-m}\sqsubseteq\langle\Delta_1\rangle_{-m}$. In the same way, $\Gamma=\Theta_0',\Theta_1'$ with $\langle\Theta_1'\rangle_{-m}\sqsubseteq\langle\Delta_2\rangle_{-m}$. Moreover, we have easily $\varphi;\Phi\vDash K\geq m$. Thus, by the Lemma 4.4.3 (weakening) for π_P and π_Q, we obtain:

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

This concludes this case.

- **Case** $m:(\nu a)P\equiv(\nu a)(m:P)$.

  Suppose that $\varphi;\Phi;\Gamma\vdash m:(\nu a)P\triangleleft K{+}m$. Then, the derivation has the shape:

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

  Recall that, by Lemma 4.4.6, $\langle\langle T\rangle_{+m}\rangle_{-m}=\langle T\rangle_{-0}=T$. So, we have:

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

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

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

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

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

- **Case** $m:(n:P)\equiv(m{+}n):P$.

  Suppose that $\varphi;\Phi;\Gamma\vdash m:(n:P)\triangleleft K{+}m$. Then the derivation has the shape:

$$\dfrac{\dfrac{\dfrac{\dfrac{\pi_P}{\varphi;\Phi;\langle\Delta\rangle_{-n}\vdash P\triangleleft K'}}{\varphi;\Phi;\Delta\vdash n:P\triangleleft K'{+}n}\qquad \varphi;\Phi\vdash\langle\Gamma\rangle_{-m}\sqsubseteq\Delta\qquad \varphi;\Phi\vDash K'{+}n\leq K}{\varphi;\Phi;\langle\Gamma\rangle_{-m}\vdash n:P\triangleleft K}}{\varphi;\Phi;\Gamma\vdash m:(n:P)\triangleleft K{+}m}$$

  We have $\langle\Gamma\rangle_{-(m+n)}=\langle\langle\Gamma\rangle_{-m}\rangle_{-n}$. So, by Lemma 4.4.2, we obtain $\langle\Gamma\rangle_{-m}=\Theta,\Theta'$ with $\varphi;\Phi\vdash\langle\Theta'\rangle_{-n}\sqsubseteq\langle\Delta\rangle_{-n}$. So, by weakening, we obtain:

$$\dfrac{\dfrac{\dfrac{\dfrac{\pi_P^w}{\varphi;\Phi;\langle\Theta\rangle_{-n},\langle\Delta\rangle_{-n}\vdash P\triangleleft K'}\qquad \varphi;\Phi\vdash\langle\Gamma\rangle_{-(m+n)}\sqsubseteq(\langle\Theta\rangle_{-n},\langle\Delta\rangle_{-n})\qquad \varphi;\Phi\vDash K'\leq K{-}n}{\varphi;\Phi;\langle\Gamma\rangle_{-(m+n)}\vdash P\triangleleft K{-}n}}{\varphi;\Phi;\Gamma\vdash(m{+}n):P\triangleleft K{+}m}$$

  For the converse, suppose that $\varphi;\Phi;\Gamma\vdash(m{+}n):P\triangleleft K{+}(m{+}n)$. Then the derivation has the shape:

$$\dfrac{\dfrac{\pi_P}{\varphi;\Phi;\langle\Gamma\rangle_{-(m+n)}\vdash P\triangleleft K{+}}}{\varphi;\Phi;\Gamma\vdash(m{+}n):P\triangleleft K{+}(m{+}n)}$$

  So, we have:

$$\dfrac{\dfrac{\dfrac{\pi_P}{\varphi;\Phi;\langle\Gamma\rangle_{-(m+n)}\vdash P\triangleleft K}}{\varphi;\Phi;\langle\Gamma\rangle_{-m}\vdash n:P\triangleleft K{+}n}}{\varphi;\Phi;\Gamma\vdash m:(n:P)\triangleleft K{+}(m{+}n)}$$

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

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

$$\frac{\varphi;\Phi \vDash I = J \qquad (\varphi,\widetilde{i});\Phi \vdash \widetilde{T} \sqsubseteq \widetilde{U} \qquad (\varphi,\widetilde{i});\Phi \vDash K' \leq K}{\varphi;\Phi \vdash \forall\widetilde{i}.\mathtt{srv}_I^K(\widetilde{T}) \sqsubseteq \forall\widetilde{i}.\mathtt{isrv}_J^{K'}(\widetilde{U})}$$

$$\frac{\varphi;\Phi \vDash I = J \qquad (\varphi,\widetilde{i});\Phi \vdash \widetilde{U} \sqsubseteq \widetilde{T} \qquad (\varphi,\widetilde{i});\Phi \vDash K \leq K'}{\varphi;\Phi \vdash \forall\widetilde{i}.\mathtt{srv}_I^K(\widetilde{T}) \sqsubseteq \forall\widetilde{i}.\mathtt{osrv}_J^{K'}(\widetilde{U})}$$

$$\frac{\varphi;\Phi \vDash I = J \qquad (\varphi,\widetilde{i});\Phi \vdash \widetilde{T} \sqsubseteq \widetilde{U} \qquad (\varphi,\widetilde{i});\Phi \vDash K' \leq K}{\varphi;\Phi \vdash \forall\widetilde{i}.\mathtt{isrv}_I^K(\widetilde{T}) \sqsubseteq \forall\widetilde{i}.\mathtt{isrv}_J^{K'}(\widetilde{U})}$$

$$\frac{\varphi;\Phi \vDash I = J \qquad (\varphi,\widetilde{i});\Phi \vdash \widetilde{U} \sqsubseteq \widetilde{T} \qquad (\varphi,\widetilde{i});\Phi \vDash K \leq K'}{\varphi;\Phi \vdash \forall\widetilde{i}.\mathtt{osrv}_I^K(\widetilde{T}) \sqsubseteq \forall\widetilde{i}.\mathtt{osrv}_J^{K'}(\widetilde{U})}$$

Figure 4.25: An Alternative Presentation of Subtyping

- **Case** $0 : P \equiv P$. This case is direct because the rule for $0 : P$ does nothing.

This concludes all the base case. We can then prove Lemma 4.4.8 by induction on $P \equiv Q$. All the base case have been done, symmetry and transitivity are direct by induction hypothesis. For the cases of contextual congruence, the proof is straightforward. □

Now that we have Lemma 4.4.8, we can work up to the congruence relation. In order to proceed to the subject reduction, we first need an exhaustive description of subtyping. As expressed before, the transitivity rule is not necessary and so in the following proof it will be better to consider a subtyping without transitivity. This alternative presentation of subtyping is described in Figure 4.25 for servers. The case of simple channels can easily be deduced from the case of servers.

And of course, we have:

Lemma 4.4.9 (Exhaustive Description of Subtyping)**.** *$\varphi;\Phi \vdash T \sqsubseteq U$ for the usual description if and only if $\varphi;\Phi \vdash T \sqsubseteq U$ with the rules of Figure 4.25.*

Proof. The converse is direct, all the rules described in Figure 4.25 can be derived in the original subtyping description using at most once the transitivity rule. For the direct implication, the proof is rather straightforward. We proceed by induction on the usual subtyping relation. All base cases are indeed of this form, and the only hard case is for transitivity. For this case, we have to use the induction hypothesis and consider all cases in which the right member of a rule of Figure 4.25 can match with a left member of another rule. This is a long case analysis, but all the cases are simple. □

And with all this, we can finally work on the subject reduction.

Theorem 4.4.2 (Subject Reduction)**.** *If $\varphi;\Phi;\Gamma \vdash P \triangleleft K$ and $P \Rightarrow Q$ then $\varphi;\Phi;\Gamma \vdash Q \triangleleft K$.*

Let us show this Theorem. We do this by induction on $P \Rightarrow Q$. Let us first notice that when considering the typing of P, again the first subtyping rule has no importance since we can always start the typing of Q with the exact same subtyping rule. We now proceed by doing the case analysis on the rules of Figure 4.18.

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

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

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

The first subtree can be used exactly as it is to type the server in the right part of the reduction relation. Furthermore, as the name a is used as an input and as an output, so the original type in Γ for this name must be a server type $\forall \widetilde{i}.\mathtt{srv}_I^{K_a}(\widetilde{T})$. As this server does not lose its input capacity after the time decrease, we also know that $\varphi; \Phi \vDash I \geq n$. So, by Lemma 4.4.9, we have:

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

There are now two cases to consider.

– Let us first consider that $\varphi; \Phi \vDash I \geq m$. Then, we obtain directly:

$$\varphi; \Phi \vDash I_0{+}n = I_1{+}m \qquad (\varphi, \widetilde{i}); \Phi \vdash \widetilde{T_1} \sqsubseteq \widetilde{T_0} \qquad (\varphi, \widetilde{i}); \Phi \vDash K_0' \leq K_1'$$

Thus, by subtyping, from π_P we can obtain a derivation of $(\varphi, \widetilde{i}); \Phi; \langle \Delta_0 \rangle_{-I_0}, \widetilde{v} : \widetilde{T_1} \vdash P \triangleleft K_1'$. By Lemma 4.4.5, we have a derivation of $\varphi; \Phi\{\widetilde{J}/\widetilde{i}\}; (\langle \Delta_0 \rangle_{-I_0})\{\widetilde{J}/\widetilde{i}\}, \widetilde{v} : \widetilde{T_1}\{\widetilde{J}/\widetilde{i}\} \vdash P \triangleleft K_1'\{\widetilde{J}/\widetilde{i}\}$. As $\widetilde{i}$ only appears in $\widetilde{T_1}$ and K_1', we obtain a derivation of $\varphi; \Phi; \langle \Delta_0 \rangle_{-I_0}, \widetilde{v} : \widetilde{T_1}\{\widetilde{J}/\widetilde{i}\} \vdash P \triangleleft K_1'\{\widetilde{J}/\widetilde{i}\}$.
Now, by using several times Lemma 4.4.2, we have:

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

for some ϵ_0, ϵ_1. By Lemma 4.4.3, and as $\varphi; \Phi \vdash I = I_1{+}m = I_0{+}n$ we can obtain two proofs, with the subtyping rule:

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

Thus, by the substitution lemma (Lemma 4.4.7), we have $\varphi; \Phi; \langle \Gamma \rangle_{-I} \vdash P[\widetilde{v} := \widetilde{e}] \triangleleft K_1\{\widetilde{J}/\widetilde{i}\}$. As by definition, $I \geq m$ and $I \geq n$, let us call $I' = I{-}\max(n,m)$, and we can obtain the following typing using the associated typing rule:

$$\varphi; \Phi; \langle \Gamma \rangle_{-I'} \vdash \max(n,m) : P[\widetilde{v} := \widetilde{e}] \triangleleft \max(n,m){+}K_1\{\widetilde{J}/\widetilde{i}\}$$

Then, by delaying (Lemma 4.4.6), we have $\varphi; \Phi; \langle\langle\Gamma\rangle_{-I'}\rangle_{+I'} \vdash \max(n,m) : P[\widetilde{v} := \widetilde{e}] \triangleleft I{+}K_1\{\widetilde{J}/\widetilde{i}\}$, and $\Gamma = \epsilon'_0, \epsilon'_1$ with $\varphi; \Phi \vdash \epsilon'_1 \sqsubseteq \langle\langle\Gamma\rangle_{-I'}\rangle_{+I'}$. Recall that $\varphi; \Phi \vDash I_1{+}m{+}K'_1\{\widetilde{J}/\widetilde{i}\} \leq K$. Thus, again by subtyping and weakening, we obtain

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

And this concludes this case.

– Now, we suppose that we do not have $\varphi; \Phi \vDash I \geq m$. Note that as we know $\varphi; \Phi \vDash I \geq n$, it means that $m > n$. Moreover, as $\varphi; \Phi \vDash I_1 = I - m$, then $\varphi; \Phi \vDash I_1{+}m = \max(I,m)$. We still have:

$$(\varphi, \widetilde{i}); \Phi \vdash \widetilde{T_1} \sqsubseteq \widetilde{T_0} \qquad (\varphi, \widetilde{i}); \Phi \vDash K'_0 \leq K'_1$$

Thus, by subtyping, from π_P we can obtain a derivation of $(\varphi, \widetilde{i}); \Phi; \Theta_0, \widetilde{v} : \widetilde{T_1} \vdash P \triangleleft K'_1$. By Lemma 4.4.5, we have a derivation of $\varphi; \Phi\{\widetilde{J}/\widetilde{i}\}; \Theta_0\{\widetilde{J}/\widetilde{i}\}, \widetilde{v} : \widetilde{T_1}\{\widetilde{J}/\widetilde{i}\} \vdash P \triangleleft K'_1\{\widetilde{J}/\widetilde{i}\}$. As $\widetilde{i}$ only appears in $\widetilde{T_1}$ and K'_1, we obtain a derivation of $\varphi; \Phi; \Theta_0, \widetilde{v} : \widetilde{T_1}\{\widetilde{J}/\widetilde{i}\} \vdash P \triangleleft K'_1\{\widetilde{J}/\widetilde{i}\}$.
Now, by using several times Lemma 4.4.2, we have:

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

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

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

for some ϵ_2. Note that we have $\varphi; \Phi \vDash J = max(I,m) = I_1{+}m$ since $\varphi; \Phi \vDash I_0{+}n = I$ and $\varphi; \Phi \vDash I_1{+}m = max(I,m)$.
By Lemma 4.4.3, we can obtain two proofs, with the subtyping rule,

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

Thus, by the substitution lemma (Lemma 4.4.7), we have $\varphi; \Phi; \langle\Gamma\rangle_{-J} \vdash P[\widetilde{v} := \widetilde{e}] \triangleleft K_1\{\widetilde{J}/\widetilde{i}\}$. Recall that in this case, $\max(n,m) = m$. We can obtain the following typing using the associated typing rule:

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

Then, by delaying (Lemma 4.4.6), we have $\varphi; \Phi; \langle\langle\Gamma\rangle_{-(J-m)}\rangle_{+(J-m)} \vdash m : P[\widetilde{v} := \widetilde{e}] \triangleleft J{+}K_1\{\widetilde{J}/\widetilde{i}\}$, and $\Gamma = \epsilon'_0, \epsilon'_1$ with $\varphi; \Phi \vdash \epsilon'_1 \sqsubseteq \langle\langle\Gamma\rangle_{-(J-m)}\rangle_{+(J-m)}$. Recall that $\varphi; \Phi \vDash J = I_1{+}m$ and $\varphi; \Phi \vDash I_1{+}m{+}K_1\{\widetilde{J}/\widetilde{i}\} \leq K$. Thus, again by subtyping and weakening, we obtain

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

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

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

$$\dfrac{\dfrac{\dfrac{\varphi;\Phi;\Delta_0 \vdash a : \mathsf{in}_{I_0}(\widetilde{T_0}) \qquad \dfrac{\pi_P}{\varphi;\Phi;\langle\Delta_0\rangle_{-I_0}, \widetilde{v} : \widetilde{T_0} \vdash P \triangleleft K_0'}}{\varphi;\Phi;\Delta_0 \vdash a(\widetilde{v}).P \triangleleft I_0{+}K_0'} \qquad \varphi;\Phi \vdash \langle\Gamma_0\rangle_{-n} \sqsubseteq \Delta_0 \qquad \varphi;\Phi \vDash I_0{+}K_0' \leq K_0}{\varphi;\Phi;\langle\Gamma_0\rangle_{-n} \vdash a(\widetilde{v}).P \triangleleft K_0}}{\dfrac{\varphi;\Phi;\Gamma_0 \vdash n : a(\widetilde{v}).P \triangleleft K_0{+}n \qquad \varphi;\Phi \vdash \Gamma \sqsubseteq \Gamma_0; K_0{+}n \leq K}{\varphi;\Phi;\Gamma \vdash n : a(\widetilde{v}).P \triangleleft K}}$$

$$\dfrac{\dfrac{\dfrac{\varphi;\Phi;\Delta_1 \vdash a : \mathsf{out}_{I_1}(\widetilde{T_1}) \qquad \dfrac{\pi_e}{\varphi;\Phi;\langle\Delta_1\rangle_{-I_1} \vdash \widetilde{e} : \widetilde{T_1}}}{\varphi;\Phi;\Delta_1 \vdash \overline{a}\langle\widetilde{e}\rangle \triangleleft I_1} \qquad \varphi;\Phi \vdash \langle\Gamma_1\rangle_{-m} \sqsubseteq \Delta_1 \qquad \varphi;\Phi \vDash I_1 \leq K_1}{\varphi;\Phi;\langle\Gamma_1\rangle_{-m} \vdash \overline{a}\langle\widetilde{e}\rangle \triangleleft K_1}}{\dfrac{\varphi;\Phi;\Gamma_1 \vdash m : \overline{a}\langle\widetilde{e}\rangle \triangleleft K_1{+}m \qquad \varphi;\Phi \vdash \Gamma \sqsubseteq \Gamma_1; K_1{+}m \leq K}{\varphi;\Phi;\Gamma \vdash m : \overline{a}\langle\widetilde{e}\rangle \triangleleft K}}$$

As the name a is used as an input and as an output, so the original type in Δ for this name must be a channel type $\mathsf{ch}_I(\widetilde{T})$. As this channel is not erased after the time decrease, we also know that $\varphi;\Phi \vDash I \geq n$ and $\varphi;\Phi \vDash I \geq m$. So, by Lemma 4.4.9, we have:

$$\varphi;\Phi \vDash I_0 = I{-}n \qquad (\varphi,\widetilde{i});\Phi \vdash \widetilde{T} \sqsubseteq \widetilde{T_0} \qquad \varphi;\Phi \vDash I{-}m = I_1 \qquad (\varphi,\widetilde{i});\Phi \vdash \widetilde{T_1} \sqsubseteq \widetilde{T}$$

So, we obtain directly:

$$\varphi;\Phi \vDash I_0{+}n = I_1{+}m \qquad (\varphi,\widetilde{i});\Phi \vdash \widetilde{T_1} \sqsubseteq \widetilde{T_0}$$

Thus, by subtyping, from π_P we can obtain a derivation of $\varphi;\Phi;\langle\Delta_0\rangle_{-I_0}, \widetilde{v} : \widetilde{T_1} \vdash P \triangleleft K_0'$. Now, by using several times Lemma 4.4.2, we have:

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

for some ϵ_0, ϵ_1. By Lemma 4.4.3, and as $\varphi;\Phi \vdash I = I_1{+}m = I_0{+}n$ we can obtain two proofs, with the subtyping rule,

$$\varphi;\Phi;\langle\Gamma\rangle_{-I}, \widetilde{v} : \widetilde{T_1} \vdash P \triangleleft K_0' \qquad \varphi;\Phi;\langle\Gamma\rangle_{-I} \vdash \widetilde{e} : \widetilde{T_1}$$

Thus, by the substitution lemma (Lemma 4.4.7), we have $\varphi;\Phi;\langle\Gamma\rangle_{-I} \vdash P[\widetilde{v} := \widetilde{e}] \triangleleft K_0'$. As by definition, $I \geq m$ and $I \geq n$, let us call $I' = I{-}\max(n,m)$, and we can obtain the following typing using the associated typing rule:

$$\varphi;\Phi;\langle\Gamma\rangle_{-I'} \vdash \max(n,m) : P[\widetilde{v} := \widetilde{e}] \triangleleft \max(n,m){+}K_0'$$

Then, by delaying (Lemma 4.4.6), we have $\varphi;\Phi;\langle\langle\Gamma\rangle_{-I'}\rangle_{+I'} \vdash \max(n,m) : P[\widetilde{v} := \widetilde{e}] \triangleleft I{+}K_0'$, and $\Delta = \epsilon_0', \epsilon_1'$ with $\varphi;\Phi \vdash \epsilon_1' \sqsubseteq \langle\langle\Gamma\rangle_{-I'}\rangle_{+I'}$. Recall that $\varphi;\Phi \vDash I_0{+}n{+}K_0' \leq K'$. Thus, again by subtyping and weakening, we obtain

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

And this concludes this case.

- **Case** $\texttt{tick}.P \Rightarrow 1 : P$. This case is direct because both constructors have exactly the same typing rule.

- **Case** $\texttt{match}\ []\ \{[] \mapsto P;;\ x :: y \mapsto Q\} \Rightarrow P$. This case is similar to its counterpart for natural number, so we only detail this one. Suppose given a derivation $\varphi; \Phi; \Gamma \vdash \texttt{match}\ []\ \{[] \mapsto P;;\ x :: y \mapsto Q\} \triangleleft K$. Then the derivation has the shape:

$$\dfrac{\dfrac{\overline{\varphi; \Phi; \Delta \vdash [] : \mathsf{List}[0,0](\mathcal{B}')} \qquad \varphi; \Phi \vdash \Gamma \sqsubseteq \Delta; \mathsf{List}[0,0](\mathcal{B}') \sqsubseteq \mathsf{List}[I,J](\mathcal{B})}{\varphi; \Phi; \Gamma \vdash [] : \mathsf{List}[I,J](\mathcal{B})} \qquad \dfrac{\pi_P}{\varphi; (\Phi, I \leq 0); \Gamma \vdash P \triangleleft K}}{\varphi; \Phi; \Gamma \vdash \texttt{match}\ []\ \{[] \mapsto P;;\ x :: y \mapsto Q\} \triangleleft K}$$

  Where we ignore the branch for Q as it does not interest us in this case. By Lemma 4.4.9, we obtain:

$$\varphi; \Phi \vDash I \leq 0 \qquad \varphi; \Phi \vDash 0 \leq J \qquad \varphi; \Phi \vdash \mathcal{B}' \sqsubseteq \mathcal{B}$$

  As $\varphi; \Phi \vDash I \leq 0$, by Lemma 4.4.4, we obtain directly from π_P a derivation $\varphi; \Phi; \Gamma \vdash P \triangleleft K$.

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

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

  Where we ignore the branch for P and π_Q proves $\varphi; (\Phi, J \geq 1); \Gamma, x{:}\mathcal{B}, y{:}\mathsf{List}[I{-}1, J{-}1](\mathcal{B}) \vdash Q \triangleleft K$. Lemma 4.4.9 gives us the following information:

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

  From this, we can deduce the following constraints:

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

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

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

  Then, by Lemma 4.4.4, from π_Q we obtain a derivation of $\varphi; \Phi; \Gamma, x{:}\mathcal{B}, y{:}\mathsf{List}[I{-}1, J{-}1](\mathcal{B}) \vdash Q \triangleleft K$. By the substitution lemma (Lemma 4.4.7), we obtain $\varphi; \Phi; \Gamma \vdash Q[x, y := e, e'] \triangleleft K$. This concludes this case.

- **Case** $P \mid R \Rightarrow Q \mid R$ with $P \Rightarrow Q$. Suppose that $\varphi; \Phi; \Gamma \vdash P \mid R \triangleleft K$. Then the derivation has the shape:

$$\dfrac{\dfrac{\pi_P}{\varphi; \Phi; \Gamma \vdash P \triangleleft K} \qquad \dfrac{\pi_R}{\varphi; \Phi; \Gamma \vdash R \triangleleft K}}{\varphi; \Phi; \Gamma \vdash P \mid R \triangleleft K}$$

  By induction hypothesis, with the derivation π_P of $\varphi; \Phi; \Gamma \vdash P \triangleleft K$, we obtain a derivation π_Q of $\varphi; \Phi; \Gamma \vdash Q \triangleleft K$. Then, we can derive the following proof:

$$\dfrac{\dfrac{\pi_Q}{\varphi;\Phi;\Gamma \vdash Q \triangleleft K} \quad \dfrac{\pi_R}{\varphi;\Phi;\Gamma \vdash R \triangleleft K}}{\varphi;\Phi;\Gamma \vdash Q \mid R \triangleleft K}$$

This concludes this case.

- **Case** $(\nu a)P \Rightarrow (\nu a)Q$ with $P \Rightarrow Q$. Suppose that $\varphi;\Phi;\Gamma \vdash (\nu a)P \triangleleft K$. Then the derivation has the shape:

$$\dfrac{\dfrac{\pi_P}{\varphi;\Phi;\Gamma, a:T \vdash P \triangleleft K}}{\varphi;\Phi;\Gamma \vdash (\nu a)P \triangleleft K}$$

By induction hypothesis, with the derivation π_P of $\varphi;\Phi;\Gamma, a:T \vdash P \triangleleft K$, we obtain a derivation π_Q of $\varphi;\Phi;\Gamma, a:T \vdash Q \triangleleft K$ We can then derive the proof:

$$\dfrac{\dfrac{\pi_Q}{\varphi;\Phi;\Gamma, a:T \vdash Q \triangleleft K}}{\varphi;\Phi;\Gamma \vdash (\nu a)Q \triangleleft K}$$

This concludes this case.

- **Case** $n : P \Rightarrow n : Q$ with $P \Rightarrow Q$. Suppose that $\varphi;\Phi;\Gamma \vdash n : P \triangleleft K{+}n$. Then, the derivation has the shape:

$$\dfrac{\dfrac{\pi_P}{\varphi;\Phi;\langle\Gamma\rangle_{-n} \vdash P \triangleleft K}}{\varphi;\Phi;\Gamma \vdash n : P \triangleleft K{+}n}$$

By induction hypothesis, we have a derivation π_Q of $\varphi;\Phi;\langle\Gamma\rangle_{-n} \vdash Q \triangleleft K$, thus we can give the derivation:

$$\dfrac{\dfrac{\pi_Q}{\varphi;\Phi;\langle\Gamma\rangle_{-n} \vdash Q \triangleleft K}}{\varphi;\Phi;\Gamma \vdash n : Q \triangleleft K{+}n}$$

- **Case** $P \Rightarrow Q$ with $P \equiv P'$, $P' \Rightarrow Q'$ and $Q \equiv Q'$. Suppose that $\varphi;\Phi;\Gamma \vdash P \triangleleft K$. By Lemma 4.4.8, we have $\varphi;\Phi;\Gamma \vdash P' \triangleleft K$. By induction hypothesis, we obtain $\varphi;\Phi;\Gamma \vdash Q' \triangleleft K$. Then, again by Lemma 4.4.8, we have $\varphi;\Phi;\Gamma \vdash Q \triangleleft K$. This concludes this case.

This concludes the proof of Theorem 4.4.2.

Now that we have the subject reduction for $\Rightarrow$, we can easily deduce a more generic form of Theorem 4.4.1.

Theorem 4.4.3. *Let P be an annotated process and let m be its global parallel complexity. Then, for a typing $\varphi;\Phi;\Gamma \vdash P \triangleleft K$, we have $\varphi;\Phi \vDash K \geq m$.*

Proof. By Theorem 4.4.2, all reductions from P using $\Rightarrow$ preserve the typing. Moreover, for any process Q, if we have a typing $\varphi;\Phi;\Gamma \vdash Q \triangleleft K$, then $\varphi;\Phi \vDash K \geq \mathcal{C}_\ell(Q)$. Indeed, a constructor $n : P$ incurs an increment of the complexity of n both in typing and in the definition of $\mathcal{C}_\ell(Q)$, and for parallel composition the typing imposes a complexity greater than the maximum as in the definition for $\mathcal{C}_\ell(Q)$. Thus, for any process Q reachable from P, we have $\varphi;\Phi \vDash K \geq \mathcal{C}_\ell(Q)$, so K is indeed a bound on the parallel complexity. □

Corollary 4.4.1 is then obtained with the substitution lemma and the rule for parallel composition.

4.4.3 Examples: Bitonic Sort / Brute Force

Bitonic Sort

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

- A *bitonic sequence* is either a sequence composed of an increasing sequence followed by a decreasing sequence (e.g. [2, 7, 23, 19, 8, 5]), or a cyclic rotation of such a sequence (e.g. [8, 5, 2, 7, 23, 19]).
- The algorithm uses two main functions, `bmerge` and `bsort`.
- `bmerge` takes a bitonic sequence and recursively sorts it, as follows:

  Assume $s = [a_0, \ldots, a_{n-1}]$ is a bitonic sequence such that the sequence $[a_0, \ldots, a_{n/2-1}]$ is increasing and the sequence $[a_{n/2} \ldots, a_{n-1}]$ is decreasing, then we consider:

  $s_1 = [\min(a_0, a_{n/2}), \min(a_1, a_{n/2+1}) \ldots, \min(a_{n/2-1}, a_{n-1})]$

  $s_2 = [\max(a_0, a_{n/2}), \max(a_1, a_{n/2+1}) \ldots, \max(a_{n/2-1}, a_{n-1})]$

  Then we have: s_1 and s_2 are bitonic and satisfy: $\forall x \in s_1, \forall y \in s_2, x \leq y$.

  `bmerge` then applies recursively to s_1 and s_2 to produce a sorted sequence.
- `bsort` takes a sequence and recursively sorts it. It starts by separating the sequence in two. Then, it recursively sorts the first sequence in increasing order, and the second sequence in decreasing order. With this, we obtain a bitonic sequence that can be sorted with `bmerge`.

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

First, we suppose that a server for comparison `lessthan` is already implemented. We start with `bcompare` such that given two lists of same length, it creates the list of maximum and the list of minimum. This is described in Figure 4.26.

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

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

The important things to notice is that this server has complexity 1, and the channel taken in input has a time 1. A sketch of this typing is given in Figure 4.27. The cases of empty lists are not detailed, but they are easy. In the non-empty case, for the ν constructor, we must give a type to the channels b and c. We use:

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

And we can then type the different processes in parallel.

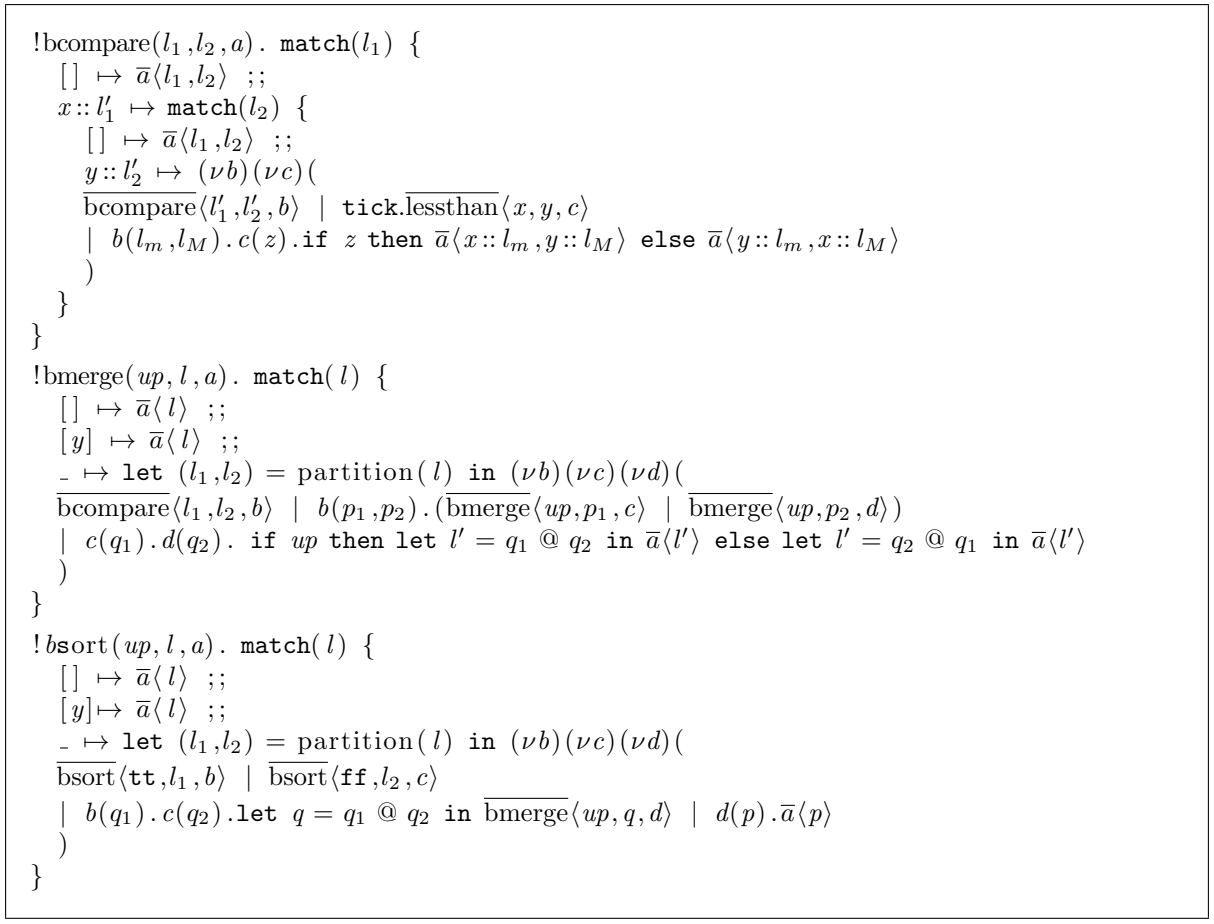

Figure 4.26: Bitonic Sort

- For the call to `bcompare`, the arguments have the expected type, and this call has complexity 1 because of the type of `bcompare`.
- For the process $\texttt{tick}.\overline{\texttt{lessthan}}\langle x, y, c\rangle$, the tick enforces a decreasing of time 1 in the context. This modifies in particular the time of c, that becomes 0. Thus, we can do the call to `lessthan` as everything is well-typed.
- Finally, for the last process, we have in the two branches a shape $b(\cdots).c(\cdots).\overline{a}\langle\cdots\rangle$. So, by Example 4.4.1, as all those three channels b, c and a have a time equal to 1, we have a complexity 1 for this typing.

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

Then, to present the processes for bitonic sort, let us use the macro $\texttt{let}\ \widetilde{v} = f(\widetilde{e})\ \texttt{in}\ P$ to represent $(\nu a)(\overline{f}\langle\widetilde{e}, a\rangle \mid a(\widetilde{v}).P)$, and let us also use a generalized pattern matching. We also assume that we have a function for concatenation of lists and a function `partition` taking a list of size $2n$, and giving two lists corresponding to the first n elements and the last n elements. Then, the process for bitonic sort is given in Figure 4.26.

Then, the main point in the typing of the two remaining servers is to find a solution to a recurrence relation for the complexity of server types. In the typing of `bmerge`, we suppose given a list of size smaller than 2^i and we choose both the complexity of this type and the time of the channel a equal to an index $f(i)$ as in Section 4.4.1. So, it means we choose for `bmerge`

$$\dfrac{\dfrac{\dfrac{}{i;i\geq 1;\Delta\vdash(l_1',l_2',b):\widetilde{T}(i\text{–}1)}}{i;i\geq 1;\Delta\vdash\overline{bc}\langle l_1',l_2',b\rangle\triangleleft 1}\qquad\dfrac{\dfrac{}{\cdots\vdash(x,y,c):(\mathcal{B},\mathcal{B},\mathsf{ch}_0(\mathsf{Bool}))}}{i;i\geq 1;\langle\Delta\rangle_{-1}\vdash\overline{lt}\langle x,y,c\rangle\triangleleft 0}}{i;i\geq 1;\Delta\vdash\mathtt{tick}.\overline{lt}\langle x,y,c\rangle\triangleleft 1}\qquad\dfrac{\text{See Example 4.4.1}}{i;i\geq 1;\Delta\vdash b(l_m,l_M).c(z).\mathtt{if}\ \cdots\triangleleft 1}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{i;i\geq 1;\Gamma',(l_1,l_2,a):\widetilde{T}(i),(l_1',l_2',b):\widetilde{T}(i\text{–}1),x,y:\mathcal{B},c:\mathsf{ch}_1(\mathsf{Bool})\vdash\cdots\mid\cdots\mid\cdots\triangleleft 1}}{i;i\geq 1;\Gamma',(l_1,l_2,a):\widetilde{T}(i),x,y:\mathcal{B},l_1',l_2':\mathsf{List}[0,i\text{–}1](\mathcal{B})\vdash(\nu b)(\nu c)\cdots\triangleleft 1}}{i;i\geq 1;\Gamma',(l_1,l_2,a):\widetilde{T}(i),x:\mathcal{B},l_1':\mathsf{List}[0,i\text{–}1](\mathcal{B})\vdash\mathtt{match}\ l_2\ \{[]\mapsto\cdots;;\ y::l_2'\mapsto\cdots\}\triangleleft 1}}{i;\cdot;\Gamma',(l_1,l_2,a):\widetilde{T}(i)\vdash\mathtt{match}\ l_1\ \{[]\mapsto\cdots;;\ x::l_1'\mapsto\cdots\}\triangleleft 1\qquad\Gamma'\text{ time invariant}\qquad\cdot;\cdot;\vdash\langle\Gamma\rangle_{-0}\sqsubseteq\Gamma'}$$

$$\cdot;\cdot;\Gamma\vdash!bc(l_1,l_2,a).\ldots\triangleleft 0$$

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

$$\Gamma\equiv lt:\mathtt{srv}_0^0(\mathcal{B},\mathcal{B},\mathsf{ch}_0(\mathsf{Bool})),bc:\forall i.\mathtt{srv}_0^1(\widetilde{T}(i))$$

$$\Gamma'\equiv lt:\mathtt{srv}_0^0(\mathcal{B},\mathcal{B},\mathsf{ch}_0(\mathsf{Bool})),bc:\forall i.\mathtt{osrv}_0^1(\widetilde{T}(i))$$

Figure 4.27: Type Derivation for Bitonic Comparison

the type:

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

Then, the typing given in Figure 4.28 gives us the following conditions, for the three branches, when $i\geq 1$:

$$f(i)\geq 1\qquad f(i)\geq 1{+}f(i\text{–}1)\qquad f(i)\geq f(i\text{–}1)\geq f(i\text{–}1)$$

So, we can take $f(i)=i$, and thus `bmerge` has logarithmic complexity.

In the same way, for `bsort` we choose the type:

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

The typing is given in Figure 4.29, which gives us the following conditions, again for the three branches, when $i\geq 1$:

$$g(i)\geq g(i\text{–}1)\qquad g(i)\geq g(i\text{–}1){+}f(i)\qquad g(i)\geq g(i\text{–}1){+}f(i)$$

So, if we take $f(i)=i$ as previously, we have:

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

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

Notice that in this example, the type system gives recurrence relations corresponding to the usual recurrence relations we would obtain with a complexity analysis by hand. Here, the recurrence relation is only on f or g because channel names are only used as return channels, so their time is always equal to the complexity of the server that uses them. In general this is not the case as we saw before in Section 4.4.1, so we obtain in general mutually recurrent relations when defining a server.

Brute Force

In this example, we focus on a brute forcing algorithm: given a propositional logic formula on n boolean variables, can we find an assignment for those variables such that the formula is true? For this example, we use both binary words and booleans as data-types. The formal rules for

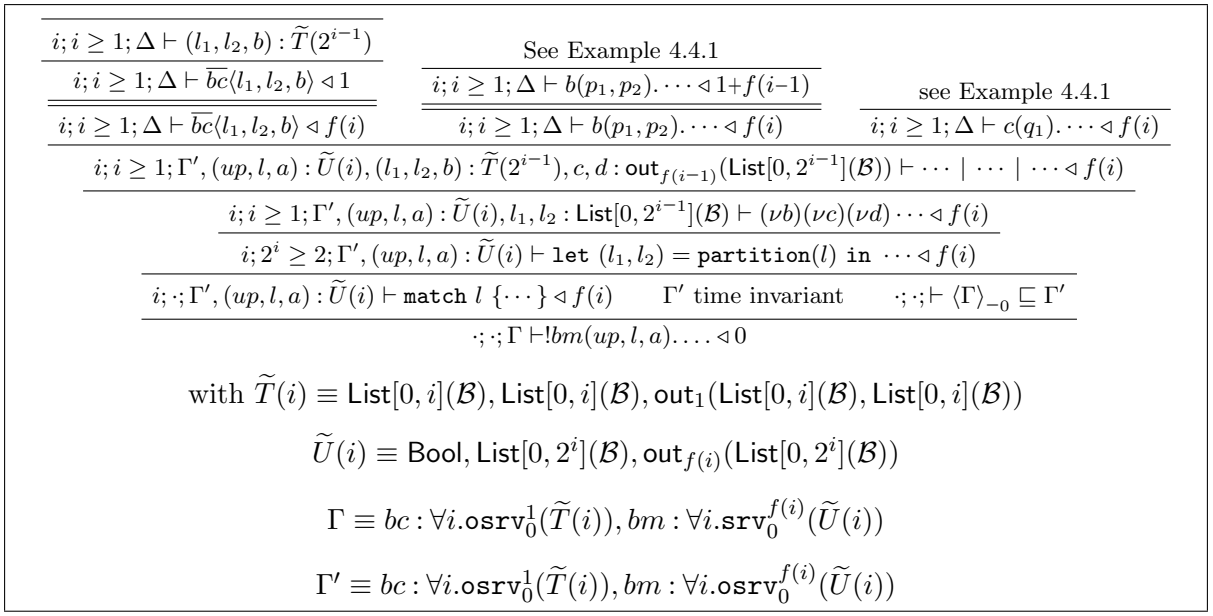

Figure 4.28: Type Derivation for Bitonic Merge

those types are not written in Figure 4.10 and Figure 4.11, but they can be deduced from the rules for integers.

To begin with, we will give an abstraction of formulas on n boolean variables in the π-calculus. A formula is represented by a server name *formula* with the following type:

$$\forall i.\mathtt{srv}_0^k(\mathsf{Word}[0, i], \mathsf{out}_k(\mathsf{Bool}))$$

Thus, a formula receives an assignment for the n variables given by a binary word w. This word $w = w_1 w_2 \cdots w_m$ represents the assignment $x_i = \mathtt{tt}$ if and only if $w_i = 1$. If $w_i = 0$ or $m < i$ then $x_i = \mathtt{ff}$. In particular, the formula should not read more that the n first letters of w, so we can assume that for a given formula, evaluating the truth of this formula can be done in a constant time k. Please note that this k can still depend on the evaluated formula, and in particular it can depend on n. Finally, this server returns the truth value of the formula on a return channel, sending a value of type Bool.

We do not detail how to obtain this server in practice. For propositional formulas for example, it should be clear that basic constructors on booleans are enough in order to have a simple encoding. The problem we consider is, given a server representing a formula, is there an assignment that satisfies this formula? In order to do this, we construct a function (a server) *test* of type

$$\forall j, k.\mathtt{srv}_0^{f(j,k)}(\mathsf{Nat}[0, j], \forall i.\mathtt{osrv}_0^k(\mathsf{Word}[0, i], \mathsf{out}_k(\mathsf{Bool})), \mathsf{out}_{f(j,k)}(\mathsf{Bool}))$$

So, this *test* function takes as input the integer n, a formula, and a return channel. For the sake of simplicity, we do not send the assignment that satisfies the formula if it exists, but we could have done this without changing anything to complexity analysis. This function will use an auxiliary recursive function *constr* that constructs all the possible assignments and feeds them to the formula, with type:

$$\forall j', j, k.\mathtt{srv}_0^{g(j',j,k)}(\mathsf{Word}[0, j'], \mathsf{Nat}[0, j{-}j'], \forall i.\mathtt{osrv}_0^k(\mathsf{Word}[0, i], \mathsf{out}_k(\mathsf{Bool})), \mathsf{out}_{g(j',j,k)}(\mathsf{Bool}))$$

Intuitively, this auxiliary function takes an unfinished assignment (the word of size j') and completes it depending on the number of variables that must still be assigned (the integer of size $j{-}j'$). The actual processes are given in Figure 4.30.

$$\dfrac{\dfrac{\dfrac{i; i \geq 1; \Delta \vdash (\mathtt{tt}, l_1, b) : \widetilde{v}(i\text{–}1)}{i; i \geq 1; \Delta \vdash \overline{bs}\langle \mathtt{tt}, l_1, b\rangle \triangleleft g(i\text{–}1)}}{i; i \geq 1; \Delta \vdash \overline{bs}\langle \mathtt{tt}, l_1, b\rangle \triangleleft g(i)} \qquad \dfrac{\text{See Example 4.4.1}}{i; i \geq 1; \Delta \vdash b(q_1).\cdots \triangleleft g(i\text{–}1)+f(i)} \qquad \dfrac{\text{See Example 4.4.1}}{i; i \geq 1; \Delta \vdash d(p).\overline{a}\langle p\rangle \triangleleft g(i)}}{\dfrac{\dfrac{\dfrac{\dfrac{i; i \geq 1; \Gamma'', b, c : \mathsf{out}_{g(i-1)}(\mathsf{List}[0, 2^{i-1}])(\mathcal{B}), d : \mathsf{out}_{(g(i-1)+f(i))}(\mathsf{List}[0, 2^i])(\mathcal{B}) \vdash \cdot \triangleleft g(i)}{i; i \geq 1; \Gamma', (up, l, a) : \widetilde{v}(i), l_1, l_2 : \mathsf{List}[0, 2^{i-1}](\mathcal{B}) \vdash (\nu b)(\nu c)(\nu d)\cdots \triangleleft g(i)}}{i; 2^i \geq 2; \Gamma', (up, l, a) : \widetilde{v}(i) \vdash \mathtt{let}\ (l_1, l_2) = \mathtt{partition}(l)\ \mathtt{in}\ \cdots \triangleleft g(i)}}{i; \cdot; \Gamma', (up, l, a) : \widetilde{v}(i) \vdash \mathtt{match}\ l\ \{\cdots\} \triangleleft g(i) \qquad \Gamma' \text{ time invariant} \qquad \cdot; \cdot; \vdash \langle\Gamma\rangle_{-0} \sqsubseteq \Gamma'}}{\cdot; \cdot; \Gamma \vdash !bs(up, l, a).\ldots \triangleleft 0}$$

with $\widetilde{U}(i) \equiv \mathsf{Bool}, \mathsf{List}[0, 2^i](\mathcal{B}), \mathsf{out}_{f(i)}(\mathsf{List}[0, 2^i](\mathcal{B}))$

$\widetilde{v}(i) \equiv \mathsf{Bool}, \mathsf{List}[0, 2^i](\mathcal{B}), \mathsf{out}_{g(i)}(\mathsf{List}[0, 2^i](\mathcal{B}))$

$\Gamma \equiv bm : \forall i.\mathtt{osrv}_0^{f(i)}(\widetilde{U}(i)), bs : \forall i.\mathtt{srv}_0^{g(i)}(\widetilde{v}(i))$

$\Gamma \equiv bm : \forall i.\mathtt{osrv}_0^{f(i)}(\widetilde{U}(i)), bs : \forall i.\mathtt{osrv}_0^{g(i)}(\widetilde{v}(i))$

Figure 4.29: Type Derivation for Bitonic Sort

```
!test(n, f, r). constr⟨ϵ, n, f, r⟩

!constr(w, n, f, r). match (n) {
  0 ↦ f⟨w, r⟩
  s(m) ↦ (νa)(νb) (constr⟨s0(w), m, f, a⟩) | (constr⟨s1(w), m, f, b⟩)
    | a(t). b(t'). if t then r⟨t⟩ else r⟨t'⟩
}
```

Figure 4.30: Brute Forcing Formula Satisfiability

In a first analysis, we do not add any tick anywhere, thus the complexity only comes from the evaluation of the formula. As before, the complexity analysis consists in finding the functions f and g. By the definition of test, we obtain immediately that:

$$f(j, k) = g(0, j, k)$$

Then, from the definition of *constr*, we obtain in the first branch of the pattern matching:

$$g(j', j, k) \geq k$$

for any j, j', since the call to f has complexity k. Moreover, in the second branch, we obtain, for $j - j' \geq 1$ (equivalently, $j \geq j' + 1$)

$$g(j', j, k) \geq g(j' + 1, j, k)$$

Indeed, the two recursive calls to *constr* are done on a word of size $j' + 1$ and an integer of size j–$(j' + 1) = j$–j'–1. Moreover, the subprocess $if\ t\ then\ \overline{r} < t >\ else\ \overline{r} < t' >$ imposes that the time of r must be greater than the time of a and b, giving again the same inequality $g(j', j, k) \geq g(j' + 1, j, k)$.

Thus, we can take in particular

$$f(j, k) = g(j', j, k) = k$$

Indeed, all those calls to the formula can be done in parallel, and the span is then the complexity of one call to the formula.

A first thing to remark with this result is that even if the span seems good, in practice we would need to run 2^n computations in parallel in order to obtain this span. In particular, for n large enough, this is not feasible in practice. This is were the work analysis comes in handy. Indeed, a work analysis of this process gives us $2^j \cdot k'$. We should have, if we consider the work complexity functions f' and g':

$$f'(j,k) = g'(0,j,k) \qquad g'(j',j,k) \geq 2 \cdot g'(j'+1,j,k)$$

Thus, we can take:

$$g'(j',j,k) = 2^{j-j'} \cdot k \qquad f'(j,k) = 2^j \cdot k$$

So, from the huge difference of complexity between work and span, one can see that even if this process is highly parallelisable, it requires a huge amount of computations in parallel. That is why we believe a span analysis alone is often not sufficient and it should be completed with a work analysis.

On a second approach, we consider that evaluating the formula is not the only costly operation for this process, and that the construction of those assignments should be taken in account. A way to do that is to add a `tick` after an input of *constr*. In this case, the constraints become:

$$f(j,k) = g(j',j,k) \qquad g(j',j,k) \geq k+1 \qquad g(j',j,k) \geq g(j'+1,j,k)+1$$

And we can take

$$g(j',j,k) = k + (j - j') + 1 \qquad f(j,k) = k + j + 1$$

And so the cost of constructing those assignments is linear in j. In particular, in the case of a propositional formula of size polynomial in n (such as SAT), we obtain a process with a polynomial time complexity for span, and an exponential complexity for work.

Overall, the type system seems expressive enough to type interesting parallel programs, however it does not behave well with some concurrent behaviours because of time uniqueness. We thus present a usage type system in order to solve this problem.

4.5 Span Analysis with Sized Types and Usages

In this section, we present some results obtained in joint work with Naoki Kobayashi [15], on a usage type system with sizes for parallel complexity analysis. As stated in Section 4.1.3, the usage type system we define here differs from the one in the literature [77, 72, 75] since we manage time differently, and thus all the operations must be defined accordingly. In particular, our type system relies on time intervals, in order to have more precise complexity bounds (see Section 4.5.3), and the notion of reliability should take in account any possible reduction path, which was not the case in deadlock-freedom analysis.

Still, in order to introduce the relevance of usages for complexity, let us look back at an example presented in Section 4.4.1

Example 4.5.1 (Motivating Example)**.** *We define*

$$P := a().\texttt{tick}.\overline{a}\langle\rangle$$

Then, the complexity of $P \mid P \mid P \mid \cdots \mid P \mid \overline{a}\langle\rangle$ *is equal to the number of* P *in parallel.*

Let us look at the rule for parallel composition. If we take, as before, a rule of the shape:

$$\frac{\Gamma \vdash Q_1 \triangleleft K_1 \qquad \Gamma \vdash Q_2 \triangleleft K_2}{\Gamma \vdash Q_1 \mid Q_2 \triangleleft \max(K_1, K_2)}$$

given a typing $\Gamma \vdash P \triangleleft K$, then we would obtain $\Gamma \vdash P \mid \cdots \mid P \triangleleft K$ and so we have two processes with different complexities when composed with $\overline{a}\langle\rangle$ that are not distinguished by typing. In Section 4.4.1, P was not typable and so this did not break soundness, but it implied some limitations on the expressivity of the type system. A better alternative is to separate contexts.

$$\frac{\Gamma \vdash Q_1 \triangleleft K_1 \qquad \Delta \vdash Q_2 \triangleleft K_2}{f(\Gamma, \Delta) \vdash Q_1 \mid Q_2 \triangleleft \max(K_1, K_2)}$$

This could for example lead to linear type systems, session types, or usages [73]. As we want a type system that can type example such as P, usages seem adapted. Indeed, the concept of reliability, the central notion of usages, that allows types to adapt compositionally, is especially useful here, as we will see in Example 4.5.3.

4.5.1 Usages and Type System

Recall from Section 4.1.3 that in order to define a usage type system, we need first to define usages as some simple parallel programs, and we need to give a meaning to time annotations. As we saw before, an infinite time annotation may be useful for usages, so, for this section, we modify slightly the size annotations.

Indices

Definition 4.5.1. *We take again $\mathcal{V}$ as a countable set of index variables. The set of indices, representing integers in $\mathbb{N}_\infty = \mathbb{N} \cup \{\infty\}$, is given by the following grammar.*

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

where $i \in \mathcal{V}$. So, an index $I_{\mathbb{N}}$ corresponds to an index from the previous sections, and now we use the notation I to consider indexes that may have an infinite value.

As before, given an index valuation $\rho : \mathcal{V} \to \mathbb{N}$, we can extend the interpretation of function symbols to indices, this gives us a number $[\![I]\!]_\rho \in \mathbb{N}_\infty$. As for substitution, we ask that an index variable is always replaced by an integer index, so it cannot be replaced by infinity. Formally, for an index I, the substitution of the occurrences of i in I by $J_{\mathbb{N}}$ (denoted $I\{J_{\mathbb{N}}/i\}$), is only defined for an integer index $J_{\mathbb{N}}$, and not for global index J. This substitution is easily defined for infinity by $\infty\{J_{\mathbb{N}}/i\} = \infty$.

As before, we will use constraints on indices, using this time relations on $\mathbb{N}_\infty$. Finally, we will use the following notations.

Definition 4.5.2 (Notations for Indices)**.** *In order to harmonize notation, we extend some operations on indices $I_{\mathbb{N}}, J_{\mathbb{N}}$ to indices I, J. We will use the following operations:*

$$\infty + J = I + \infty = \infty \qquad \max(\infty, J) = \max(I, \infty) = \infty$$

$$\min(\infty, J) = \min(J, \infty) = J \qquad \infty\text{–}1 = \infty$$

Usages

We use *usages* to express the channel-wise behaviour of a process. Our notion of usages has been inspired by the usages introduced in type systems for deadlock-freedom [91, 72, 77], but differs from the original one in a significant manner. We define usages as a kind of CCS [89] processes on a single channel, where each action is annotated with two time intervals.

The set of usages, ranged over by U or V, is given by:

$$U, V ::= 0 \mid (U|V) \mid \mathtt{In}_{J_c}^{A_o}.U \mid \mathtt{Out}_{J_c}^{A_o}.U \mid !U \mid U + V$$
$$A_o, B_o ::= [I, J] \qquad J_c, I_c ::= J \mid [I, J]$$

Given a set of index variables φ and a set of constraints Φ, for an interval $[I, J]$, we always require that $\varphi; \Phi \vDash I \leq J$. For an interval $A_o = [I, J]$, we denote $\mathsf{Left}(A_o) = I$ and $\mathsf{Right}(A_o) = J$. In the original notion of usages [91, 72, 77], A_o and J_c were just numbers. The extension to intervals plays an important role in our analysis. Note that J_c is not always an interval, as it can be a single index J. However, this single index J should be understood as the interval $[-\infty, J]$.

Intuitively, a channel with usage 0 is not used at all. A channel of usage $U \mid V$ can be used according to U and V possibly in parallel. The usage $\mathtt{In}_{J_c}^{A_o}.U$ describes a channel that may be used for input, and then used according to U. The two intervals A_o and J_c, called *obligation* and *capacity* respectively, are used to achieve a kind of assume-guarantee reasoning. The obligation A_o indicates a *guarantee* that if the channel is indeed used for input, then the input will become ready during the interval A_o. The capacity J_c indicates the *assumption* that if the environment performs a corresponding output, that output will be provided during the time interval J_c after the input becomes ready. For example, if a channel a has usage $\mathtt{In}_{J_c}^{[1,1]}.0$, then the process $\mathtt{tick}.a().0$ conforms to the usage, but $a().0$ and $\mathtt{tick}.\mathtt{tick}.a().0$ do not. Furthermore, if $J_c = [0, 1]$, and if a process $\mathtt{tick}^k.\overline{a}$ is running in parallel with $\mathtt{tick}.a().0$, then k belongs to the interval $[1, 1] + [0, 1] = [1, 2]$. Similarly, $\mathtt{Out}_{J_c}^{A_o}.U$ has the same meaning but for output. The usage $!U$ denotes the usage U that can be replicated infinitely, and $U + V$ denotes a non-deterministic choice between the usages U and V. This is useful for example in a case of pattern matching where a channel can be used very differently in the two branches. For the sake of conciseness, we may use $\alpha_{J_c}^{A_o}.U$ to denote either the usage $\mathtt{Out}_{J_c}^{A_o}.U$ or $\mathtt{In}_{J_c}^{A_o}.U$.

Recall that the obligation and capacity intervals in usages express a sort of assume-guarantee reasoning. We thus require that the assume-guarantee reasoning in a usage is "consistent" (or *reliable*, in the terminology of usages). For example, the usage $\mathtt{In}_{[1,1]}^{[0,0]} \mid \mathtt{Out}_0^{[1,1]}$ is reliable, since the part $\mathtt{In}_{[1,1]}^{[0,0]}$ assumes that a corresponding output will become ready at time 1, and the other part $\mathtt{Out}_0^{[1,1]}$ indeed guarantees that. Then, $\mathtt{Out}_0^{[1,1]}$ assumes that a corresponding input will be ready by the time the output becomes ready, and the part $\mathtt{In}_{[1,1]}^{[0,0]}$ guarantees that. In contrast, the usage $\mathtt{In}_{[1,1]}^{[0,0]} \mid \mathtt{Out}_0^{[2,2]}$ is problematic because, although the part $\mathtt{In}_{[1,1]}^{[0,0]}$ assumes that an output will be ready at time 1, $\mathtt{Out}_0^{[2,2]}$ provides the output only at time 2. The consistency on assume-guarantee reasoning must hold during the whole computation; for example, in the usage $\mathtt{In}_{[0,0]}^{[0,0]}.\mathtt{In}_{[1,1]}^{[0,0]} \mid \mathtt{Out}_{[0,0]}^{[0,0]}.\mathtt{Out}_0^{[2,2]}$, the assume/guarantee on the first input/output pair is fine, but the usage expressing the next communication: $\mathtt{In}_{[1,1]}^{[0,0]} \mid \mathtt{Out}_0^{[2,2]}$ is problematic. To properly define the reliability of usages during the whole computation, we first prepare a reduction semantics for usages, by viewing usages as CCS processes.

Definition 4.5.3 (Congruence for Usages)**.** *Congruence on usages is defined as the least congruence relation closed under:*

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

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

We have the usual relations for parallel composition. We also add a relation defining replication $!U \equiv !U \mid U$. The other relations allow manipulation of replication. This will be useful for the subusage relation, as it can increase the set of typable programs.

Now that we have congruence, as before, we give the reduction semantics. Let us first introduce some notations.

Definition 4.5.4 (Operations on Usages)**.** *We define the operations $\oplus$, $\sqcup$, and $+$ by:*

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

Note that $\oplus$ is an operation that takes an obligation interval and a capacity and returns an interval. This is where we see the fact that a capacity J alone should be understood as the interval $[-\infty, J]$. Indeed, when considering a single index, the lower bound of the sum becomes 0. So, in particular, we have $A_o \oplus J \neq A_o \oplus [0, J]$ in general. A case where we need this single index will be explained in Example 4.5.2.

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

$$\begin{aligned}
&\uparrow^{A_o} 0 = 0 \qquad \uparrow^{A_o} (U \mid V) = \uparrow^{A_o} U \mid \uparrow^{A_o} V \\
&\uparrow^{A_o} (U + V) = \uparrow^{A_o} U + \uparrow^{A_o} V \\
&\uparrow^{A_o} \alpha^{B_o}_{J_c}.U = \alpha^{A_o + B_o}_{J_c}.U \qquad \uparrow^{A_o} (!U) = !(\uparrow^{A_o} U)
\end{aligned}$$

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

Intuitively, a usage $\uparrow^{A_o} U$ corresponds to the usage U delayed by a time approximated by the interval A_o. Given two obligations A_o and B_o, $A_o \sqcup B_o$ corresponds to an interval of time approximating the time for which those two obligations are respected. For example, if an input has the obligation to be ready in the interval of time $[4, 8]$ and an output has the obligation to be ready in the interval of time $[5, 7]$, then we now for sure that the input and the output will both be ready in the interval of time $[5, 8]$.

The reduction relation is given by the rules of Figure 4.31. The first rule means that to reduce a usage, we choose one input and one output, and then we trigger the communication between them. This communication occurs and does not lead to an error when the capacity of an action corresponds indeed to a bound on the time the dual action is defined. This is given by the relation $A_o \subseteq B_o \oplus J_c$. As an example, let us suppose that $B_o = [1, 3]$, and the time for which the output becomes ready is in fact 2, then the capacity J_c says that after two units of time, the synchronization should happen in the interval J_c. So, if we take $J_c = [5, 7]$ for example, then if we call t the time for which the dual input becomes ready, we must have $t \in [2 + 5, 2 + 7]$. This should be true for any time value in B_o, so we want that $\forall t' \in [1, 3], \forall t \in A_o, t \in [t' + 5, t' + 7]$, and this is equivalent to $A_o \subseteq B_o \oplus [5, 7] = [8, 8]$. Indeed, 8 is the only time that is in the three intervals $[6, 8]$, $[7, 9]$ and $[8, 10]$. The case where $J_c = J$ is a single index occurs when t can be smaller than t', and in this case we only ask that the upper bound is correct: $\forall t' \in B_o, \forall t \in A_o, t \leq t' + J$.

If the bound was incorrect, we trigger an error, see the second rule. In the case everything went well, the continuation is delayed by an approximation of the time when this communication occurs (see $\uparrow^{A_o \sqcup B_o}$ in the first rule). The idea is that we would like, after a communication, to synchronize the time of the continuation with the other subprocesses of a usage. As it is not

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

$$\frac{}{\varphi;\Phi \vdash U + V \longrightarrow U} \qquad \frac{}{\varphi;\Phi \vdash U + V \longrightarrow V}$$

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

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

Figure 4.31: Reduction Rules for Usages

easy to advance the time of all the other subprocesses, delaying the current subprocess leads to an easier semantics. In the rules for $U + V$, a reduction step in usages can also make a non-deterministic choice.

An error in a usage reduction means that the assume-guarantee reasoning was inconsistent. Keeping that in mind, we define what is a reliable usage, that is to say a usage with consistent time indications.

Definition 4.5.5 (Reliability)**.** *A usage U is* reliable *under $\varphi;\Phi$ when for any reduction from U using $\longrightarrow$, it does not lead to an error.*

A type T is reliable under $\varphi;\Phi$ if it is a base type or a channel (resp. server) type with a reliable usage.

Example 4.5.2. *Take the usage*

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

The only possible reduction step (with symmetry) is:

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

as we have indeed $[1,1] \subseteq [0,0] \oplus [1,1] = [1,1]$ and $[0,0] \subseteq [1,1] \oplus 1 = [0,2]$. Note that the capacity $[0,1]$ instead of 1 for the input would not have worked since $[1,1] \oplus [0,1] = [1,2]$. Then, we end the reduction with

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

since $[2,2] \subseteq [1,1] \oplus 1 = [0,2]$ and $[1,1] \subseteq [2,2] \oplus 0 = [0,2]$. Thus, this usage is reliable. It corresponds for example to the usage of the channel a in the following process P

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

The obligation $[1,1]$ corresponds to waiting exactly one tick. Then, the capacities say that once they are ready, the two inputs will indeed communicate before one time unit for any reduction. And at the end, we obtain an output available at time 3, and this output has no communication. One can see that those capacities and obligations give indeed the complexity of this process. Thus, we will ask in the type system that all usages are reliable, and so the time indications will give some complexity bounds on the behaviour of a channel.

Example 4.5.3. *Let us take as another example a non-reliable usage. We take the previous example and add another input in parallel*

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

Again, the reduction gives:

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

because $[1,1] \oplus 1 = [0,2]$*, so the capacity here is not a good assumption. Therefore, this usage is not reliable. However, if we change the usage to*

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

this time we obtain a reliable channel. This example shows how reliability adapts compositionally.

We introduce another relation $U \sqsubseteq V$ called the *subusage* relation, which will be used later to define the subtyping relation. It is defined by the rules of Figure 4.32. The relation $U \sqsubseteq V$ intuitively means that if a channel is given a usage V, then it is safe for the typing to consider that the usage is U. For example, $U \sqsubseteq 0$ says that if a channel is not used (usage equal to 0), then it is safe to give any usage to this channel. Recall that an obligation and a capacity express a guarantee and an assumption respectively. The last but one rule says, read from right to left, that it is safe to weaken the guarantee and strengthen the assumption, as explained in Section 4.1.3. We use the relation $I_c \leq J_c$ to denote the relation $\subseteq$ on intervals, where a single index J is considered as the interval $[-\infty, J]$. The last rule can be understood as follows. If a channel is safely typed with the right usage. Then, in particular the usage V should happen after the action $\alpha_{J_c}^{A_o}$. Since the action $\alpha_{J_c}^{A_o}$ is finished during the interval $A_o + J_c$, the channel is used according to V only after the interval $A_o + J_c$. thus it is safe to move V outside the guard of $\alpha_{J_c}^{A_o}$, as long as it is delayed by $A_o + J_c$ unit of time. This last rule is especially useful for substitution, as explained in the example below.

Example 4.5.4. *Let us consider the process:*

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

Let us give usages to b *and* r*; here we omit time annotations for the sake of simplicity.*

$$U_r = \mathit{In} \qquad U_b = \mathit{In} \mid U_r$$

Indeed, r *is used only once as an input, and* b *is used as an input on the left, and it is sent to be used as* r *on the right. Thus, after a reduction step we obtain* $P \to b().b()$ *where* b *has usage* $U_b' = \mathit{In}.\mathit{In}$*. So, the channel* b *had usage* U_b *in* P*, but it ended up being used according to* U_b'*; that is valid since we have the subusage relation* $U_b \sqsubseteq U_b'$*.*

Before describing the type system, we give some intermediate results on subusages.

Lemma 4.5.1 (Properties of Subusage)**.** *For a set of index variables* φ *and a set of constraints* Φ *on* φ *we have:*

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

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

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

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

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

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

Figure 4.32: Rules Defining the Subusage Relation

3. If $\varphi;\Phi \vdash U \sqsubseteq V$ and U is reliable under $\varphi;\Phi$ then V is reliable under $\varphi;\Phi$.

The first point shows that subtyping is invariant by delaying. The second property means that the subusage relation serves as a simulation relation, and the last one means that the reliability is closed under the subusage relation. In our setting, it means that the subusage relation cannot lead to unsound complexity bounds.

In order to proceed with a proof, we first give the following lemma:

Lemma 4.5.2. *If $\varphi;\Phi \vDash B_o \subseteq A_o$ then $\varphi;\Phi \vdash (\uparrow^{A_o} U) \sqsubseteq (\uparrow^{B_o} U)$*

This can be proved easily by induction on U. We now give elements of proof for Point 2 and Point 3 of Lemma 4.5.1. For Point 2, we can see two possible directions, either proceed by induction on $U \sqsubseteq V$ and then do a case analysis on $V \longrightarrow V'$ on proceed by induction on $V \longrightarrow V'$ and then do a case analysis on $U \sqsubseteq V$. In both cases, the definition of subusage with the transitivity rule and congruence that can be used everywhere make the proof complicated. So, we chose to first simplify the definition of subusage.

Definition 4.5.6 (Decomposition of Subusage)**.** *Let us call $\sqsubseteq_{ct}$ the relation defined by the rule of Figure 4.32 without using congruence and transitivity. Then, we define $\sqsubseteq_t$ as:*

$$\frac{U \equiv U' \qquad \varphi;\Phi \vdash U' \sqsubseteq_{ct} V' \qquad V' \equiv V'}{\varphi;\Phi \vdash U \sqsubseteq_t V}$$

And we have the following lemma.

Lemma 4.5.3 (Decomposition of Subusage)**.** *The subusage relation $\sqsubseteq$ is equivalent to the reflexive and transitive closure of $\sqsubseteq_t$.*

The proof can be done by induction on $\sqsubseteq$, and it relies mainly on the fact that congruence can be used in any context, so we can use it at the end. In the same way, as the subusage relation can also be used in any context, the transitivity can be done at the end. So, with this lemma we got rid of the complicated rules by putting them always at the end of a derivation. Moreover, note that it is easy to describe exhaustively subtyping for $\sqsubseteq_t$. Let us drop the $\varphi;\Phi$ notation for the sake of conciseness, and we have:

Lemma 4.5.4 (Exhaustive Description of Subusage)**.** *Let us use $* \in \{\cdot, !\}$. Then, $*U$ denotes either U or $!U$ according to the value of $*$. If $U \sqsubseteq_t V$, then one of the following cases hold.*

$$
\begin{array}{llll}
U \equiv (U_0 \mid *U_1) & V \equiv U_0 & & \\
U \equiv U_0 \mid *(U_1 + U_2) & V \equiv U_0 \mid *U_i & i \in \{1;2\} & \\
U \equiv U_0 \mid *\alpha^{A_o}_{J_c}.W & V \equiv U_0 \mid *\alpha^{A_o}_{J_c}.W' & W \sqsubseteq_{ct} W' & \\
U \equiv U_0 \mid *(U_1 + U_2) & V \equiv U_0 \mid *(U_1' + U_2) & U_1 \sqsubseteq_{ct} U_1' & \\
U \equiv U_0 \mid *(U_1 + U_2) & V \equiv U_0 \mid *(U_1 + U_2') & U_2 \sqsubseteq_{ct} U_2' & \\
U \equiv U_0 \mid *\alpha^{A_o}_{J_c}.W & V \equiv U_0 \mid *\alpha^{A_o'}_{J_c'}.W & A_o' \subseteq A_o & J_c \leq J_c' \\
U \equiv U_0 \mid *((\alpha^{A_o}_{J_c}.W) \mid (\uparrow^{A_o+J_c} U_1)) & V \equiv U_0 \mid *\alpha^{A_o}_{J_c}.(W \mid U_1) & &
\end{array}
$$

This proof is done directly by induction on $U \sqsubseteq_{ct} V$. The replication context rule works because we have $!(U_0 \mid W) \equiv !U_0 \mid !W$ and $!!W \equiv !W$. We now go back to Point 2 of Lemma 4.5.1.

Proof. We rely on Lemma 4.5.3. So, we will prove first this intermediate lemma:

Lemma 4.5.5. *If $U \sqsubseteq_t V$ and $V \to V'$, then there exists U' such that $U \to^* U'$ and $U' \sqsubseteq V'$*

Then, if this lemma is proved, we conclude by transitivity of the property. So, it is sufficient to prove it. We will also use the following lemma

Lemma 4.5.6. *If $V \equiv U_0 \mid !U_1$ and $V \longrightarrow V'$ then $V' \equiv W \mid !U_1$ with $U_0 \mid U_1 \longrightarrow W$*

Indeed, there are three cases for $V \longrightarrow V'$. Either it is only a reduction step in U_0 independently of $!U_1$, either it is a reduction step within U_1 (note that one copy is always sufficient), either it is a synchronization between one action in U_0 and one action in U_1. In the first case, the lemma is true because we can arbitrarily add U_1. In the same way, in the second case the lemma is correct because we can just ignore U_0. In the third case, the lemma is verified since we allow this synchronization between U_0 and U_1.

We now start the proof. We proceed by induction on $U \sqsubseteq_{ct} V$, and we use the exhaustive description given by Lemma 4.5.4. We always consider the case $* =!$ as it is the harder of the two cases. Let us give some interesting cases:

- $$U \equiv U_0 \mid !\alpha^{A_o}_{J_c}.W \qquad V \equiv U_0 \mid !\alpha^{A_o}_{J_c}.W' \quad W \sqsubseteq_{ct} W'$$

  The easy case is when V' is obtained by a reduction step in U_0. So, we suppose that the reduction step is a synchronization between U_0 and $\alpha^{A_o}_{J_c}.W'$. So, we have:

  $$U_0 \equiv U_0' \mid \overline{\alpha}^{B_o}_{I_c}.W_0$$

  If $V' = \texttt{err}$, then we take $U' = \texttt{err}$ and concludes this case. Otherwise, we have:

  $$V' \equiv U_0' \mid !\alpha^{A_o}_{J_c}.W' \mid \uparrow^{(A_o \sqcup B_o)}(W_0 \mid W')$$

  So, we take

  $$U' = U_0' \mid !\alpha^{A_o}_{J_c}.W \mid \uparrow^{A_o \sqcup B_o}(W_0 \mid W)$$

  And, we have indeed:

  $$U \longrightarrow U' \qquad U' \sqsubseteq V'$$

  The fact that $U' \sqsubseteq V'$ is given by the previous point of Lemma 4.5.1.

- $$U \equiv U_0 \,|\, !\alpha^{A_o}_{J_c}.W \qquad V \equiv U_0 \,|\, !\alpha^{A'_o}_{J'_c}.W$$
$$A'_o \subseteq A_o \qquad J_c \leq J'_c$$
Again, the only interesting case is when the synchronization is not only in U_0. So, we have:
$$U_0 \equiv U'_0 \,|\, !\overline{\alpha}^{B_o}_{I_c}.W_0$$
If we have $A_o \subseteq B_o \oplus I_c$ and $B_o \subseteq A_o \oplus J_c$ then
$$V' \equiv U'_0 \,|\, !\alpha^{A'_o}_{J'_c}.W \mid \uparrow^{A'_o \sqcup B_o}(W_0 \mid W)$$
because $A'_o \subseteq A_o$ and $J_c \leq J'_c$ so we have $A'_o \subseteq B_o \oplus I_c$ and $B_o \subseteq A_o \oplus J'_c$. Thus, we take
$$U' = U'_0 \,|\, !\alpha^{A_o}_{J_c}.W \mid \uparrow^{A_o \sqcup B_o}(W_0 \mid W)$$
We have indeed $U \longrightarrow U'$ and $U' \sqsubseteq V'$ by Lemma 4.5.2. Otherwise, we obtain an error for U' and so it indeed is a subusage of V'.

- $$U \equiv U_0 \,|\, !((\alpha^{A_o}_{J_c}.W) \mid (\uparrow^{A_o + J_c} U_1))$$
$$V \equiv U_0 \,|\, !\alpha^{A_o}_{J_c}.(W \mid U_1)$$
Again, if the reduction step $V \to V'$ is a synchronization between subprocesses in U_0, it is simple. So let us suppose that:
$$U_0 \equiv U'_0 \mid \overline{\alpha}^{B_o}_{I_c}.W_0$$
If we have $B_o \subseteq A_o \oplus J_c$ and $A_o \subseteq B_o \oplus I_c$, then
$$V' \equiv U'_0 \,|\, !\alpha^{A_o}_{J_c}.(W \mid U_1) \mid (\uparrow^{A_o \sqcup B_o}(W \mid U_1 \mid W_0))$$
We pose U' equal to:
$$U'_0 \,|\, !((\alpha^{A_o}_{J}.W) \mid (\uparrow^{A_o + J_c} U_1)) \mid (\uparrow^{A_o \sqcup B_o}(W \mid W_0))$$
We have indeed $U \to U'$ and we have $U' \sqsubseteq V'$ because $A_o \sqcup B_o \subseteq A_o + J_c$. Indeed,
$$\mathsf{Left}(A_o + J_c) \leq \max(\mathsf{Left}(A_o), \mathsf{Left}(B_o))$$
As either $J_c = J$ and so $\mathsf{Left}(A_o = J_c) = \mathsf{Left}(A_o)$, either $J_c = [I, J]$ and
$$\mathsf{Left}(A_o) + I \leq \mathsf{Right}(A_o) + I \leq \mathsf{Left}(B_o)$$
since $B_o \subseteq A_o \oplus J_c$. Moreover, we have
$$\max(\mathsf{Right}(A_o), \mathsf{Right}(B_o)) \leq \mathsf{Right}(A_o + J_c)$$
again because $B_o \subseteq A_o \oplus J_c$.

Thus, we have indeed Lemma 4.5.5, and we deduce the second point of Lemma 4.5.1.

Finally, the third point is a direct consequence of the second point. Indeed, suppose that U is reliable. So, for any reduction from U, it does not lead to an error. Let us take a reduction from V. By the third point, it gives us a reduction from U where some steps are a subtype of the steps in V. So, as an error cannot happen in the steps from U, and the only usage U such that $U \sqsubseteq \texttt{err}$ is $\texttt{err}$, we know that the reduction from V does not lead to an error. Thus, V is reliable. □

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

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

Figure 4.33: Subtyping Rules for Usage Types

Finally, we also have the following lemmas, saying that delaying does not modify the behaviour of a type.

Lemma 4.5.7 (Invariance by Delaying)**.** *For any interval A_o:*

1. *If $\varphi;\Phi \vdash \uparrow^{A_o}U \longrightarrow V'$ then, there exists V such that $\uparrow^{A_o}V = V'$ and $\varphi;\Phi \vdash U \longrightarrow V$. (with $\mathtt{err} = \uparrow^{A_o}\mathtt{err}$)*
2. *If $\varphi;\Phi \vdash U \longrightarrow V$ then $\varphi;\Phi \vdash \uparrow^{A_o}U \longrightarrow \uparrow^{A_o}V$.*
3. *U is reliable under $\varphi;\Phi$ if and only if $\uparrow^{A_o}U$ is reliable under $\varphi;\Phi$.*

In our setting, this lemma shows among other things that the `tick` constructor, or more generally the annotation $n : P$, does not break reliability.

Type System with Sizes

Definition 4.5.7 (Usage Types)**.** *We define* types *by the following grammar:*

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

So, as expected, usual types for π-calculus are replaced by a type with usage, and we keep track of sizes for integers.

As before, channels are classified into servers and simple channels. The type $\mathtt{ch}(\widetilde{T})/U$ describes a simple channel that is used for transmitting values of type $\widetilde{T}$ according to usage U. The type $\forall\widetilde{i}.\mathtt{srv}^{K}(\widetilde{T})/U$ describes a server channel that is used for transmitting values of type $\widetilde{T}$ according to usage U; the superscript K, is an interval for this section. It denotes a lower bound and an upper bound on the cost incurred when a server is invoked.

Also note that for a simple channel, only one type $\widetilde{T}$ is associated to all usages. So for example, in a channel of type $\mathtt{ch}(\mathsf{Nat}[I,J])/U$, at any time this channel is used, all messages must be integers between I and J.

The subtyping relation is defined by the rules of Figure 4.33. The only thing subtyping can do is to change the usage of a channel or modify the size bound on an integer.

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

Definition 4.5.8 (Parallel Composition of Types)**.** *The parallel composition of two types $T \mid T'$ is defined by:*

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

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

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

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

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

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

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

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

$$(\nu)\ \frac{\varphi;\Phi;\Gamma, a : T \vdash P \triangleleft K \qquad T \text{ reliable}}{\varphi;\Phi;\Gamma \vdash (\nu a)P \triangleleft K} \qquad \text{(sub)}\ \frac{\varphi;\Phi;\Delta \vdash P \triangleleft K \qquad \varphi;\Phi \vdash \Gamma \sqsubseteq \Delta \qquad \varphi;\Phi \vDash K \subseteq K'}{\varphi;\Phi;\Gamma \vdash P \triangleleft K'}$$

Figure 4.34: Typing Rules for Processes with Usages

Definition 4.5.9 (Replication of Type)**.** *The replication of a type* $!T$ *is defined by:*

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

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

We also introduce the following notation.

Definition 4.5.10. *Given a capacity* J_c *and an interval* $K = [K_1, K_2]$*, we define* $J_c; K$ *by;*

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

Intuitively, $J_c; K$ represents the complexity of an input/output process when the input/output has capacity J_c and the complexity of the continuation is K. $J_c = [\infty,\infty]$ means the input/output will never succeed (because there is no corresponding output/input); hence the complexity is 0. A case where this is useful is given later in Example 4.5.7. Otherwise, an upper-bound is given by $J + K_2$ (the time spent for the input/output to succeed, plus K). The lower-bound is 0, since the input/output may be blocked forever.

The type system is given in Figures 4.10 and 4.34. The typing rules for expressions are the standard ones for sized types.

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

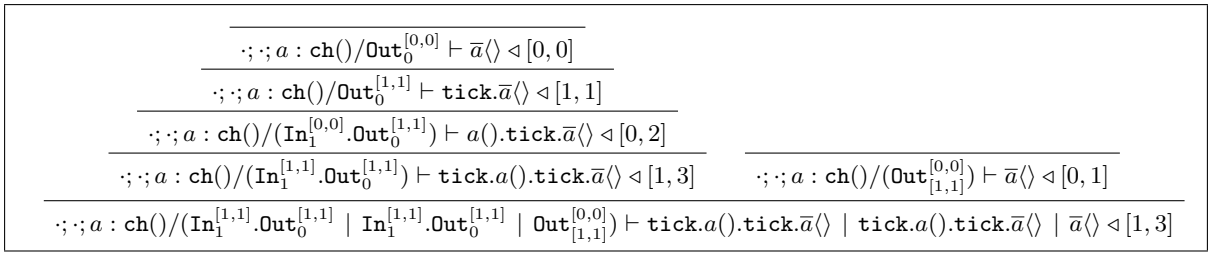

Figure 4.35: Typing of Example 4.5.2

a lower bound on the parallel complexity of P. But in practice, this lower bound is often too imprecise.[1]

The (par) rule separates a context into two parts, and the complexity is the maximum over the two complexities, both for lower bound and upper bound. The (tick) rule shows the addition of a tick implies a delay of $[1,1]$ in the context and the complexity. The (ν) rule imposes that all names must have a reliable usage when they are created. In order to type a channel with the (ich) rule, the channel must have an input usage, with obligation $[0,0]$. Note that with the subusage relation, we have $\mathtt{In}_{J_c}^{A_o} \sqsubseteq \mathtt{In}_{J_c}^{[0,0]}$ if and only if $A_o = [0,I]$ for some I. So, this typing rule imposes that the lower-bound guarantee is correct, but the rule is not restrictive for upper-bound. This rule induces a delay of J_c in both context and complexity. Indeed, in practice this input does not happen immediately as we need to wait for output. This is where the assumption on when this output is ready, given by the capacity, is useful. The rule for output (och) is similar. For a server, the rules for both input and output are similar to the one for channels in principle but differ in the way complexity is managed, similarly to Section 4.4. Finally, the (case) rule is the standard rule for a sized type system. Note that contexts are not separated in this rule. In the typing for the expression this is not a problem since names are not useful for the typing of an integer. Then for both branches, it means that the usage of channels must be the same. However, because we have the choice usage $(U+V)$, in practice we can use different usages in those two branches.

As an illustration of the type system, let us take back again the reliable usage described in Example 4.5.2 and show that it corresponds indeed to the typing of a in the process described in Example .

Example 4.5.5. *The typing derivation of the process in Example 4.5.2 is given in Figure 4.35. Note that the process*

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

is also typable, in the same way, using the usage described in Example 4.5.2, and with complexity $[1,3]$*. However, we saw in Example 4.5.3 that the usage was not reliable, and that is why we do not obtain a valid complexity bound. If we take the reliable usage*

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

It gives us back the correct complexity bound $[1,4]$

So, our type system can adapt compositionally with the use of reliability. And we saw on this example that reliability is needed to obtain soundness. An example for the use of servers and sizes is given later, in Example 4.5.6.

Remark 4.5.1. *A careful reader may wonder why we need* intervals *for obligations and capacities, instead of single numbers. An informal justification is given in the Appendix 4.5.3.*

[1]This is because in the definition of $J_C;K$ in Definition 4.5.10, we pessimistically take into account the possibility that each input/output may be blocked forever. We can avoid the pessimistic estimation of the lower-bound by incorporating information about lock-freedom [72, 74].

4.5.2 Soundness

The proof of soundness relies again on standard lemmas for type systems. In the following we will always consider P as an annotated process. As in Section 4.4.2, we introduce a typing rule for the annotation, corresponding to a generalization of the rule for `tick`.

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

Intermediate Lemmas

As usual, we start with the structure lemmas.

Lemma 4.5.8 (Weakening)**.** *Let φ, φ' be disjoint set of index variables, Φ be a set of constraints on φ, Φ' be a set of constraints on (φ, φ'), Γ and Γ' be contexts on disjoint set of variables.*

1. *If $\varphi; \Phi \vDash C$ then $(\varphi, \varphi'); (\Phi, \Phi') \vDash C$*
2. *If $\varphi; \Phi \vdash U \longrightarrow V$ and $V \neq \texttt{err}$ then $(\varphi, \varphi'); (\Phi, \Phi') \vdash U \longrightarrow V$*
3. *If U is reliable under $\varphi; \Phi$ then it is reliable under $(\varphi, \varphi'; \Phi, \Phi')$*
4. *If $\varphi; \Phi \vdash U \sqsubseteq V$ then $(\varphi, \varphi'); (\Phi, \Phi') \vdash U \sqsubseteq V$.*
5. *If $\varphi; \Phi \vdash T \sqsubseteq T'$ then $(\varphi, \varphi'); (\Phi, \Phi') \vdash T \sqsubseteq T'$.*
6. *If $\varphi; \Phi; \Gamma \vdash e : T$ then $(\varphi, \varphi'); (\Phi, \Phi'); \Gamma, \Gamma' \vdash e : T$.*
7. *If $\varphi; \Phi; \Gamma \vdash P \triangleleft K$ then $(\varphi, \varphi'); (\Phi, \Phi'); \Gamma, \Gamma' \vdash P \triangleleft K$.*

Because of our definitions that always rely on other inductive definitions, in order to prove lemmas we often need to go back to the first definition and then go over all definitions to reach type derivation. None of those points are difficult, but they all rely on previous points. Note that for the last point, we can always consider usage equal to 0 in Γ', and with this Γ' is unaffected by the $\uparrow$ operation.

Lemma 4.5.9 (Strengthening)**.** *Let φ be a set of index variables, Φ be a set of constraints on φ, and C a constraint on φ such that $\varphi; \Phi \vDash C$.*

1. *We have $\varphi; (\Phi, C) \vDash C'$ if and only if $\varphi; \Phi \vDash C'$.*
2. *If $\varphi; (\Phi, C) \vdash U \longrightarrow V$ then $\varphi; \Phi \vdash U \longrightarrow V$*
3. *If U is reliable under $\varphi; (\Phi, C)$ then it is reliable under $\varphi; \Phi$.*
4. *If $\varphi; (\Phi, C) \vdash U \sqsubseteq V$ then $\varphi; \Phi \vdash U \sqsubseteq V$*
5. *If $\varphi; (\Phi, C) \vdash T \sqsubseteq T'$ then $\varphi; \Phi \vdash T \sqsubseteq T'$.*
6. *If $\varphi; (\Phi, C); \Gamma, \Gamma' \vdash e : T$ and the variables in Γ' are not free in e, then $\varphi; \Phi; \Gamma \vdash e : T$.*
7. *If $\varphi; (\Phi, C); \Gamma, \Gamma' \vdash P \triangleleft K$ and the variables in Γ' are not free in P, then $\varphi; \Phi; \Gamma \vdash P \triangleleft K$.*

Again, those are direct induction proofs.
We now proceed to the substitution lemmas.

Lemma 4.5.10 (Index Substitution)**.** *Let φ be a set of index variables and $i \notin \varphi$. Let $J_{\mathbb{N}}$ be an index with free variables in φ. Then,*

1. *If $(\varphi, i); \Phi \vDash C$ then $\varphi; \Phi\{J_{\mathbb{N}}/i\} \vDash C\{J_{\mathbb{N}}/i\}$.*
2. *If $(\varphi, i); \Phi \vdash U \longrightarrow V$ and $V \neq \mathtt{err}$ then $\varphi; \Phi\{J_{\mathbb{N}}/i\} \vdash U\{J_{\mathbb{N}}/i\} \longrightarrow V\{J_{\mathbb{N}}/i\}$*
3. *If U is reliable under $(\varphi, i); \Phi$ then $U\{J_{\mathbb{N}}/i\}$ is reliable under $\varphi; \Phi\{J_{\mathbb{N}}/i\}$*
4. *If $(\varphi, i); \Phi \vdash U \sqsubseteq V$ then $\varphi; \Phi\{J_{\mathbb{N}}/i\} \vdash U\{J_{\mathbb{N}}/i\} \sqsubseteq V\{J_{\mathbb{N}}/i\}$*
5. *If $(\varphi, i); \Phi \vdash T \sqsubseteq T'$ then $\varphi; \Phi\{J_{\mathbb{N}}/i\} \vdash T\{J_{\mathbb{N}}/i\} \sqsubseteq T'\{J_{\mathbb{N}}/i\}$.*
6. *If $(\varphi, i); \Phi; \Gamma \vdash e : T$ then $\varphi; \Phi\{J_{\mathbb{N}}/i\}; \Gamma\{J_{\mathbb{N}}/i\} \vdash e : T\{J_{\mathbb{N}}/i\}$.*
7. *If $(\varphi, i); \Phi; \Gamma \vdash P \triangleleft K$ then $\varphi; \Phi\{J_{\mathbb{N}}/i\}; \Gamma\{J_{\mathbb{N}}/i\} \vdash P \triangleleft K\{J_{\mathbb{N}}/i\}$.*

Again, this lemma is rather easy. For the last point, in order to change the order of substitution, we need to show that, as usual:

$$I\{\widetilde{I_{\mathbb{N}}}/\widetilde{i}\}\{J_{\mathbb{N}}/j\} = I\{J_{\mathbb{N}}/j\}\{\widetilde{I_{\mathbb{N}}}\{J_{\mathbb{N}}/j\}/\widetilde{i}\}$$

We now present the variable substitution lemmas. In the setting of usages, this lemma is a bit more complex than usual. Indeed, we have a separation of contexts with the parallel composition, and we have to rely on subusage, especially the rule $\varphi; \Phi \vdash (\alpha_J^{A_o}.U) \mid (\uparrow^{A_o+J_c} V) \sqsubseteq \alpha_{J_c}^{A_o}.(U \mid V)$ as expressed in the Example 4.5.4 above. We put some emphasis on the following notation: when we write $\Gamma, v : T$ as a context in typing, it means that v does not appear in Γ.

Lemma 4.5.11 (Substitution)**.** *Let Γ and Δ be contexts such that $\Gamma \mid \Delta$ is defined. Then we have:*

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

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

Lemma 4.5.12 (Difficult Cases of Substitution)**.** *We have:*

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

Indeed, this corresponds to the cases when $v = c$ and $\Delta = (\Gamma/0), b : \mathtt{ch}(\widetilde{S})/W_1$ or $\Delta = (\Gamma/0), b : \forall \widetilde{i}.\mathtt{srv}^K(\widetilde{S})/W_1$ where $\Gamma/0$ is Γ with all usages replaced by 0. Note that it is sufficient to prove this intermediate lemma, since if Δ had another shape, it could be obtained again by subtyping. For those two points, the main difficulty is for the input and output rules. We first detail the first point of this lemma, and we will detail the difference for the second point.

1. • **Case** of input, with $a \neq b$ and $a \neq c$.

$$\frac{\varphi;\Phi;\Gamma,b:\mathtt{ch}(\widetilde{S})/W_0,c:\mathtt{ch}(\widetilde{S})/W_1,a:\mathtt{ch}(\widetilde{T})/U,\widetilde{v}:\widetilde{T}\vdash P\triangleleft K}{\varphi;\Phi;\uparrow^{J_c}\Gamma,b:\mathtt{ch}(\widetilde{S})/(\uparrow^{J_c}W_0),c:\mathtt{ch}(\widetilde{S})/(\uparrow^{J_c}W_1),a:\mathtt{ch}(\widetilde{T})/\mathtt{In}_{J_c}^{[0,0]}.U\vdash a(\widetilde{v}).P\triangleleft J_c;K}$$

By induction hypothesis, we obtain $\varphi;\Phi;\Gamma,b:\mathtt{ch}(\widetilde{S})/(W_0\mid W_1),a:\mathtt{ch}(\widetilde{T})/U,\widetilde{v}:\widetilde{T}\vdash P[c:=b]\triangleleft K$.

We then give the following proof:

$$\frac{\varphi;\Phi;\Gamma,b:\mathtt{ch}(\widetilde{S})/(W_0\mid W_1),a:\mathtt{ch}(\widetilde{T})/U,\widetilde{v}:\widetilde{T}\vdash P[c:=b]\triangleleft K}{\varphi;\Phi;\uparrow^{J_c}\Gamma,b:\mathtt{ch}(\widetilde{S})/(\uparrow^{J_c}(W_0\mid W_1)),a:\mathtt{ch}(\widetilde{T})/\mathtt{In}_{J_c}^{[0,0]}.U\vdash a(\widetilde{v}).P\triangleleft J_c;K}$$

This case is similar to all other cases when b and c do not interfere with the typing rule. Thus, we only need to show the cases when they interfere.

- **Case** of input, with $a=b$.

$$\frac{\varphi;\Phi;\Gamma,b:\mathtt{ch}(\widetilde{T})/U,c:\mathtt{ch}(\widetilde{T})/W_1,\widetilde{v}:\widetilde{T}\vdash P\triangleleft K}{\varphi;\Phi;\uparrow^{J_c}\Gamma,b:\mathtt{ch}(\widetilde{T})/\mathtt{In}_{J_c}^{[0,0]}.U,c:\mathtt{ch}(\widetilde{T})/(\uparrow^{J_c}W_1)\vdash b(\widetilde{v}).P\triangleleft J_c;K}$$

By induction hypothesis, we obtain $\varphi;\Phi;\Gamma,b:\mathtt{ch}(\widetilde{T})/(U\mid W_1),\widetilde{v}:\widetilde{T}\vdash P[c:=b]\triangleleft K$. So, we give the typing:

$$\frac{\dfrac{\varphi;\Phi;\Gamma,b:\mathtt{ch}(\widetilde{T})/(U\mid W_1),\widetilde{v}:\widetilde{T}\vdash P[c:=b]\triangleleft K}{\varphi;\Phi;\uparrow^{J_c}\Gamma,b:\mathtt{ch}(\widetilde{T})/\mathtt{In}_{J_c}^{[0,0]}.(U\mid W_1)\vdash b(\widetilde{v}).P[c:=b]\triangleleft J_c;K}}{\varphi;\Phi;\uparrow^{J_c}\Gamma,b:\mathtt{ch}(\widetilde{T})/\mathtt{In}_{J_c}^{[0,0]}.U\mid(\uparrow^{J_c}W_1)\vdash b(\widetilde{v}).P[c:=b]\triangleleft J_c;K}$$

Indeed, the last rule represents subtyping. This concludes this case.

- **Case** of input, with $a=c$.

$$\frac{\varphi;\Phi;\Gamma,b:\mathtt{ch}(\widetilde{T})/W_0,c:\mathtt{ch}(\widetilde{T})/U,\widetilde{v}:\widetilde{T}\vdash P\triangleleft K}{\varphi;\Phi;\uparrow^{J_c}\Gamma,b:\mathtt{ch}(\widetilde{T})/(\uparrow^{J_c}W_0),c:\mathtt{ch}(\widetilde{T})/\mathtt{In}_{J_c}^{[0,0]}.U\vdash c(\widetilde{v}).P\triangleleft J_c;K}$$

By induction hypothesis, we obtain $\varphi;\Phi;\Gamma,b:\mathtt{ch}(\widetilde{T})/(W_0\mid U),\widetilde{v}:\widetilde{T}\vdash P[c:=b]\triangleleft K$. So, we give the typing:

$$\frac{\dfrac{\varphi;\Phi;\Gamma,b:\mathtt{ch}(\widetilde{T})/(W_0\mid U),\widetilde{v}:\widetilde{T}\vdash P[c:=b]\triangleleft K}{\varphi;\Phi;\uparrow^{J_c}\Gamma,b:\mathtt{ch}(\widetilde{T})/\mathtt{In}_{J_c}^{[0,0]}.(W_0\mid U)\vdash b(\widetilde{v}).P[c:=b]\triangleleft J_c;K}}{\varphi;\Phi;\uparrow^{J_c}\Gamma,b:\mathtt{ch}(\widetilde{T})/\mathtt{In}_{J_c}^{[0,0]}.U\mid(\uparrow^{J_c}W_0)\vdash(c(\widetilde{v}).P)[c:=b]\triangleleft J_c;K}$$

- **Case** of output, with $a=b$.

$$\frac{\varphi;\Phi;\Gamma',b:\mathtt{ch}(\widetilde{T})/V,c:\mathtt{ch}(\widetilde{T})/W_1'\vdash\widetilde{e}:\widetilde{T}\qquad\varphi;\Phi;\Gamma,b:\mathtt{ch}(\widetilde{T})/U,c:\mathtt{ch}(\widetilde{T})/W_1\vdash P\triangleleft K}{\varphi;\Phi;\uparrow^{J_c}(\Gamma\mid\Gamma'),b:\mathtt{ch}(\widetilde{T})/\mathtt{Out}_{J_c}^{[0,0]}.(V\mid U),c:\mathtt{ch}(\widetilde{T})/\uparrow^{J_c}(W_1'\mid W_1)\vdash\overline{b}\langle\widetilde{e}\rangle.P\triangleleft J_c;K}$$

By point 1 of Lemma 4.5.11 and induction hypothesis, we obtain

$$\varphi;\Phi;\Gamma',b:\mathtt{ch}(\widetilde{T})/(V\mid W_1')\vdash\widetilde{e}:\widetilde{T}\qquad\varphi;\Phi;\Gamma,b:\mathtt{ch}(\widetilde{T})/(U\mid W_1)\vdash P\triangleleft K$$

Thus, we have:

$$\frac{\dfrac{\varphi;\Phi;\Gamma',b:\mathtt{ch}(\widetilde{T})/(V\mid W_1')\vdash\widetilde{e}:\widetilde{T}\qquad\varphi;\Phi;\Gamma,b:\mathtt{ch}(\widetilde{T})/(U\mid W_1)\vdash P\triangleleft K}{\varphi;\Phi;\uparrow^{J_c}(\Gamma\mid\Gamma'),b:\mathtt{ch}(\widetilde{T})/\mathtt{Out}_{J_c}^{[0,0]}.(V\mid U\mid W_1\mid W_1')\vdash\overline{b}\langle\widetilde{e}\rangle.P\triangleleft J_c;K}}{\varphi;\Phi;\uparrow^{J_c}(\Gamma\mid\Gamma'),b:\mathtt{ch}(\widetilde{T})/\mathtt{Out}_{J_c}^{[0,0]}.(V\mid U)\mid\uparrow^{J_c}(W_1\mid W_1')\vdash\overline{b}\langle\widetilde{e}\rangle.P\triangleleft J_c;K}$$

Again, the last rule is obtained by subtyping. We have a similar proof for the case $a=c$.

2. We now work on the case of servers. The notations are a bit cumbersome but the proofs are similar to the one for channels. The only point that need some details is for server input as there is the replication in usages that appear.

   - **Case** of input, with $a \neq b$ and $a \neq c$.

   $$\frac{(\varphi, \widetilde{i}); \Phi; \Gamma, a : \forall \widetilde{i}.\mathtt{srv}^K(\widetilde{T})/U, b : \forall \widetilde{j}.\mathtt{srv}^{K'}(\widetilde{S})/W_0, c : \forall \widetilde{j}.\mathtt{srv}^{K'}(\widetilde{S})/W_1, \widetilde{v} : \widetilde{T} \vdash P \triangleleft K}{\varphi; \Phi; \uparrow^{J_c}!\Gamma, a : \forall \widetilde{i}.\mathtt{srv}^K(\widetilde{T})/!\mathtt{In}_{J_c}^{[0,0]}.U, b : \forall \widetilde{j}.\mathtt{srv}^K(\prime)/\widetilde{S}(\uparrow^{J_c}!W_0), c : \forall \widetilde{j}.\mathtt{srv}^{K'}(\widetilde{S})/(\uparrow^{J_c}!W_1) \vdash !a(\widetilde{v}).P \triangleleft [0,0]}$$

   By induction hypothesis, we have $(\varphi, \widetilde{i}); \Phi; \Gamma, a : \forall \widetilde{i}.\mathtt{srv}^K(\widetilde{T})/U, b : \forall \widetilde{j}.\mathtt{srv}^{K'}(\widetilde{S})/(W_0 \mid W_1), \widetilde{v} : \widetilde{T} \vdash P[c := b] \triangleleft K$. So, we have the proof

   $$\frac{(\varphi, \widetilde{i}); \Phi; \Gamma, a : \forall \widetilde{i}.\mathtt{srv}^{K'}(\widetilde{T})/U, b : \forall \widetilde{j}.\mathtt{srv}^{K'}(\widetilde{S})/(W_0 \mid W_1), \widetilde{v} : \widetilde{T} \vdash P[c := b] \triangleleft K}{\varphi; \Phi; \uparrow^{J_c}!\Gamma, a : \forall \widetilde{i}.\mathtt{srv}^K(\widetilde{T})/!\mathtt{In}_{J_c}^{[0,0]}.U, b : \forall \widetilde{j}.\mathtt{srv}^{K'}(\widetilde{S})/(\uparrow^{J_c}!(W_0 \mid W_1)) \vdash !a(\widetilde{v}).P[c := b] \triangleleft [0,0]}$$

   - **Case** of input, with $a = b$.

   $$\frac{(\varphi, \widetilde{i}); \Phi; \Gamma, b : \forall \widetilde{i}.\mathtt{srv}^K(\widetilde{T})/U, c : \forall \widetilde{i}.\mathtt{srv}^K(\widetilde{T})/W_1, \widetilde{v} : \widetilde{T} \vdash P \triangleleft K}{\varphi; \Phi; \uparrow^{J_c}!\Gamma, b : \forall \widetilde{i}.\mathtt{srv}^K(\widetilde{T})/!\mathtt{In}_{J_c}^{[0,0]}.U, c : \forall \widetilde{i}.\mathtt{srv}^K(\widetilde{T})/(\uparrow^{J_c}!W_1) \vdash !a(\widetilde{v}).P \triangleleft [0,0]}$$

   By induction hypothesis, we obtain $(\varphi, \widetilde{i}); \Phi; \Gamma, b : \forall \widetilde{i}.\mathtt{srv}^K(\widetilde{T})/(U \mid W_1), \widetilde{v} : \widetilde{T} \vdash P \triangleleft K$

   $$\frac{\dfrac{(\varphi, \widetilde{i}); \Phi; \Gamma, b : \forall \widetilde{i}.\mathtt{srv}^K(\widetilde{T})/(U \mid W_1), \widetilde{v} : \widetilde{T} \vdash P[c := b] \triangleleft K}{\varphi; \Phi; \uparrow^{J_c}!\Gamma, b : \forall \widetilde{i}.\mathtt{srv}^K(\widetilde{T})/!\mathtt{In}_{J_c}^{[0,0]}.(U \mid W_1) \vdash !a(\widetilde{v}).P[c := b] \triangleleft [0,0]}}{\varphi; \Phi; \uparrow^{J_c}!\Gamma, b : \forall \widetilde{i}.\mathtt{srv}^K(\widetilde{T})/(!\mathtt{In}_{J_c}^{[0,0]}.U \mid \uparrow^{J_c}!W_1) \vdash !a(\widetilde{v}).P[c := b] \triangleleft [0,0]}$$

   This last derivation is obtained by subtyping. Indeed, by definition we have $\uparrow^{J_c}!W_1 = !\uparrow^{J_c}W_1$. Then,

   $$!\mathtt{In}_{J_c}^{[0,0]}.U \,|\, !\uparrow^{J_c}W_1 \equiv !(\mathtt{In}_{J_c}^{[0,0]}.U \mid \uparrow^{J_c}W_1) \sqsubseteq !\mathtt{In}_{J_c}^{[0,0]}.(U \mid W_1)$$

   - The case $a = c$ is similar to the previous one.

This concludes the proof.

Subject Reduction and Soundness

We then explain the subject reduction. Let us first introduce a notation:

Definition 4.5.11 (Reduction for Contexts)**.** *We say that a context Γ reduces to a context Γ' under $\varphi; \Phi$, denoted $\varphi; \Phi \vdash \Gamma \longrightarrow^* \Gamma'$ when one of the following holds:*

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

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

We then formalize the subject reduction.

Theorem 4.5.1 (Subject Reduction)**.** *If $\varphi; \Phi; \Gamma \vdash P \triangleleft K$ with all types in Γ reliable and $P \Rightarrow Q$ then there exists Γ' with $\varphi; \Phi \vdash \Gamma \longrightarrow^* \Gamma'$ and $\varphi; \Phi; \Gamma' \vdash Q \triangleleft K$.*

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

Lemma 4.5.13 (Congruence and Typing)**.** *Let P and Q be annotated processes such that $P \equiv Q$. Then, $\varphi; \Phi; \Gamma \vdash P \triangleleft K$ if and only if $\varphi; \Phi; \Gamma \vdash Q \triangleleft K$.*

Proof. Let us show Lemma 4.5.13. We prove this by induction on $P \equiv Q$. Note that as usual, for a process P, the typing system is not syntax-directed because of the subtyping rule, but we can always assume that a derivation has exactly one subtyping rule before any syntax-directed rule. We first show this property for base case of congruence. The reflexivity is trivial, then we have those interesting cases:

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

$$\dfrac{\dfrac{\dfrac{\dfrac{\pi}{\varphi; \Phi; \Gamma', a : T \vdash P \triangleleft K_1'} \quad T \text{ reliable}}{\varphi; \Phi; \Gamma' \vdash (\nu a)P \triangleleft K_1'} \quad \varphi; \Phi \vdash \Gamma \sqsubseteq \Gamma'; K_1' \subseteq K_1}{\varphi; \Phi; \Gamma \vdash (\nu a)P \triangleleft K_1} \quad \dfrac{\pi'}{\varphi; \Phi; \Delta \vdash Q \triangleleft K_2}}{\varphi; \Phi; \Gamma \mid \Delta \vdash (\nu a)P \mid Q \triangleleft K_1 \sqcup K_2}$$

  By weakening (Lemma 4.5.8), we obtain a derivation π_w' of $\varphi; \Phi; \Delta, a : (T/0) \vdash Q \triangleleft K_2$. Thus, we have the following derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{\pi}{\varphi; \Phi; \Gamma', a : T \vdash P \triangleleft K_1'} \quad \varphi; \Phi \vdash \Gamma \sqsubseteq \Gamma'; K_1' \subseteq K_1}{\varphi; \Phi; \Gamma, a : T \vdash P \triangleleft K_1} \quad \dfrac{\pi_w'}{\varphi; \Phi; \Delta, a : (T/0) \vdash Q \triangleleft K_2}}{\varphi; \Phi; \Gamma \mid \Delta, a : T \vdash P \mid Q \triangleleft K_1 \sqcup K_2} \quad T \text{ reliable}}{\varphi; \Phi; \Gamma \mid \Delta \vdash (\nu a)(P \mid Q) \triangleleft K_1 \sqcup K_2}$$

  For the converse, suppose $\varphi; \Phi; \Gamma \vdash (\nu a)(P \mid Q) \triangleleft K$. Then the derivation has the shape:

$$\dfrac{\dfrac{\dfrac{\dfrac{\pi}{\varphi; \Phi; \Gamma_P, a : T_P \vdash P \triangleleft K_1} \quad \dfrac{\pi'}{\varphi; \Phi; \Gamma_Q, a : T_Q \vdash Q \triangleleft K_2}}{\varphi; \Phi; \Gamma_P \mid \Gamma_Q, a : T_P \mid T_Q \vdash P \mid Q \triangleleft K_1 \sqcup K_2} \quad \varphi; \Phi \vdash \Gamma \sqsubseteq \Gamma_P \mid \Gamma_Q; T \sqsubseteq T_P \mid T_Q; K_1 \sqcup K_2 \subseteq K}{\varphi; \Phi; \Gamma, a : T \vdash P \mid Q \triangleleft K \quad T \text{ reliable}}}{\varphi; \Phi; \Gamma \vdash (\nu a)(P \mid Q) \triangleleft K}$$

  Since a is not free in Q, by Lemma 4.5.9, from π' we obtain a derivation π_s' of $\varphi; \Phi; \Gamma_Q \vdash Q \triangleleft K_2$. We then derive the following typing:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\pi}{\varphi; \Phi; \Gamma_P, a : T_P \vdash P \triangleleft K_1} \quad \varphi; \Phi \vdash T \sqsubseteq T_P \mid T_Q \sqsubseteq T_P}{\varphi; \Phi; \Gamma_P, a : T \vdash P \triangleleft K_1} \quad T \text{ reliable}}{\varphi; \Phi; \Gamma_P \vdash (\nu a)P \triangleleft K_1} \quad \dfrac{\pi_s'}{\varphi; \Phi; \Gamma_Q \vdash Q \triangleleft K_2}}{\varphi; \Phi; \Gamma_P \mid \Gamma_Q \vdash (\nu a)P \mid Q \triangleleft K_1 \sqcup K_2} \quad \varphi; \Phi \vdash \Gamma \sqsubseteq \Gamma_P \mid \Gamma_Q; K_1 \sqcup K_2 \subseteq K}{\varphi; \Phi; \Gamma \vdash (\nu a)P \mid Q \triangleleft K}$$

- **Case** $m : (P \mid Q) \equiv m : P \mid m : Q$. Suppose $\varphi; \Phi; \uparrow^{[m,m]}\Gamma \vdash m : (P \mid Q) \triangleleft K + [m, m]$. Then we have:

$$\dfrac{\dfrac{\dfrac{\dfrac{\pi_P}{\varphi;\Phi;\Gamma_P \vdash P \triangleleft K_1} \quad \dfrac{\pi_Q}{\varphi;\Phi;\Gamma_Q \vdash Q \triangleleft K_2}}{\varphi;\Phi;\Gamma_P \mid \Gamma_Q \vdash (P \mid Q) \triangleleft K_1 \sqcup K_2} \quad \varphi;\Phi \vdash \Gamma \sqsubseteq \Gamma_P \mid \Gamma_Q; K_1 \sqcup K_2 \subseteq K}{\varphi;\Phi;\Gamma \vdash (P \mid Q) \triangleleft K}}{\varphi;\Phi;\uparrow^{[m,m]}\Gamma \vdash m : (P \mid Q) \triangleleft K + [m,m]}$$

By Lemma 4.5.1, from $\varphi;\Phi \vdash \Gamma \sqsubseteq \Gamma_P \mid \Gamma_Q$ we obtain $\varphi;\Phi \vdash \uparrow^{[m,m]}\Gamma \sqsubseteq (\uparrow^{[m,m]}\Gamma_P) \mid (\uparrow^{[m,m]}\Gamma_Q)$. So, we give the following derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\pi_P}{\varphi;\Phi;\Gamma_P \vdash P \triangleleft K_1}}{\varphi;\Phi;\uparrow^{[m,m]}\Gamma_P \vdash m : P \triangleleft K_1 + [m,m]} \quad \dfrac{\dfrac{\pi_Q}{\varphi;\Phi;\Gamma_Q \vdash Q \triangleleft K_2}}{\varphi;\Phi;\uparrow^{[m,m]}\Gamma_Q \vdash m : Q \triangleleft K_2 + [m,m]}}{\varphi;\Phi;(\uparrow^{[m,m]}\Gamma_P) \mid (\uparrow^{[m,m]}\Gamma_Q) \vdash m : P \mid m : Q \triangleleft (K_1 \sqcup K_2) + [m,m]} \quad (1)}{\varphi;\Phi;\uparrow^{[m,m]}\Gamma \vdash m : P \mid m : Q \triangleleft (K_1 \sqcup K_2) + [m,m]} \quad \varphi;\Phi \vDash K_1 \sqcup K_2 \subseteq K}{\varphi;\Phi;\uparrow^{[m,m]}\Gamma \vdash m : P \mid m : Q \triangleleft K + [m,m]}$$

where (1) is

$$\varphi;\Phi \vdash \uparrow^{[m,m]}\Gamma \sqsubseteq (\uparrow^{[m,m]}\Gamma_P) \mid (\uparrow^{[m,m]}\Gamma_Q)$$

Now, suppose we have a typing $\varphi;\Phi;\Gamma_P \mid \Gamma_Q \vdash m : P \mid m : Q \triangleleft K_1 \sqcup K_2$. The typing has the shape:

$$\dfrac{\dfrac{\dfrac{\dfrac{\pi_P}{\varphi;\Phi;\Delta_P \vdash P \triangleleft K_P}}{\varphi;\Phi;\uparrow^{[m,m]}\Delta_P \vdash m : P \triangleleft K_P + [m,m]}}{\varphi;\Phi;\Gamma_P \vdash m : P \triangleleft K_1} \quad \dfrac{\dfrac{\dfrac{\pi_Q}{\varphi;\Phi;\Delta_Q \vdash Q \triangleleft K_Q}}{\varphi;\Phi;\uparrow^{[m,m]}\Delta_Q \vdash m : Q \triangleleft K_Q + [m,m]}}{\varphi;\Phi;\Gamma_Q \vdash m : Q \triangleleft K_2}}{\varphi;\Phi;\Gamma_P \mid \Gamma_Q \vdash m : P \mid m : Q \triangleleft K_1 \sqcup K_2}$$

with

$$\varphi;\Phi \vdash \Gamma_P \sqsubseteq \uparrow^{[m,m]}\Delta_P \qquad \varphi;\Phi \vdash \Gamma_Q \sqsubseteq \uparrow^{[m,m]}\Delta_Q \qquad \varphi;\Phi \vDash K_P + [m,m] \subseteq K_1 \qquad \varphi;\Phi \vDash K_Q + [m,m] \subseteq K_2$$

So, we derive:

$$\dfrac{\dfrac{\dfrac{\dfrac{\pi_P}{\varphi;\Phi;\Delta_P \vdash P \triangleleft K_P} \quad \dfrac{\pi_Q}{\varphi;\Phi;\Delta_Q \vdash Q \triangleleft K_Q}}{\varphi;\Phi;\Delta_P \mid \Delta_Q \vdash (P \mid Q) \triangleleft K_P \sqcup K_Q}}{\varphi;\Phi;\uparrow^{[m,m]}(\Delta_P \mid \Delta_Q) \vdash m : (P \mid Q) \triangleleft (K_P \sqcup K_Q) + [m,m]} \quad (2)}{\varphi;\Phi;\Gamma_P \mid \Gamma_Q \vdash m : (P \mid Q) \triangleleft K}$$

where (2) is

$$\varphi;\Phi \vdash \Gamma_P \mid \Gamma_Q \sqsubseteq \uparrow^{[m,m]}(\Delta_P \mid \Delta_Q); (K_P \sqcup K_Q) + [m,m] \subseteq K_1 \sqcup K_2$$

This concludes this case.

- **Case** $m : (\nu a)P \equiv (\nu a)(m : P)$.
  Suppose $\varphi;\Phi;\uparrow^{[m,m]}\Gamma \vdash m : (\nu a)P \triangleleft K + [m,m]$. Then, the typing has the shape:

$$\dfrac{\dfrac{\dfrac{\dfrac{\pi}{\varphi;\Phi;\Gamma',a:T\vdash P\triangleleft K'}\quad T\text{ reliable}}{\varphi;\Phi;\Gamma'\vdash(\nu a)P\triangleleft K'}\qquad \varphi;\Phi\vdash\Gamma\sqsubseteq\Gamma';K'\subseteq K}{\varphi;\Phi;\Gamma\vdash(\nu a)P\triangleleft K}}{\varphi;\Phi;\uparrow^{[m,m]}\Gamma\vdash m:(\nu a)P\triangleleft K+[m,m]}$$

By Lemma 4.5.7, we know that $\uparrow^{[m,m]}T$ is reliable. So, we have:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\pi}{\varphi;\Phi;\Gamma',a:T\vdash P\triangleleft K'}\qquad \varphi;\Phi\vdash\Gamma\sqsubseteq\Gamma';K'\subseteq K}{\varphi;\Phi;\Gamma,a:T\vdash P\triangleleft K}}{\varphi;\Phi;\uparrow^{[m,m]}(\Gamma,a:T)\vdash(m:P)\triangleleft K+[m,m]}\qquad \uparrow^{[m,m]}T\text{ reliable}}{\varphi;\Phi;\uparrow^{[m,m]}\Gamma\vdash(\nu a)(m:P)\triangleleft K+[m,m]}$$

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

$$\dfrac{\dfrac{\dfrac{\dfrac{\pi}{\varphi;\Phi;\Gamma',a:T'\vdash P\triangleleft K'}}{\varphi;\Phi;\uparrow^{[m,m]}\Gamma',a:\uparrow^{[m,m]}T'\vdash(m:P)\triangleleft K'+[m,m]}\qquad \varphi;\Phi\vdash\Gamma\sqsubseteq\uparrow^{[m,m]}\Gamma';T\sqsubseteq\uparrow^{[m,m]}T';K'+[m,m]\subseteq K}{\varphi;\Phi;\Gamma,a:T\vdash(m:P)\triangleleft K\qquad T\text{ reliable}}}{\varphi;\Phi;\Gamma\vdash(\nu a)(m:P)\triangleleft K}$$

As T is reliable, by Lemma 4.5.1, we have $\uparrow^{[m,m]}T'$ reliable. Then, by Lemma 4.5.7, we have T' reliable. So, we give the typing:

$$\dfrac{\dfrac{\dfrac{\dfrac{\pi}{\varphi;\Phi;\Gamma',a:T'\vdash P\triangleleft K'}\quad T'\text{ reliable}}{\varphi;\Phi;\Gamma'\vdash(\nu a)P\triangleleft K'}}{\varphi;\Phi;\uparrow^{[m,m]}\Gamma'\vdash m:(\nu a)P\triangleleft K'+[m,m]}\qquad \varphi;\Phi\vdash\Gamma\sqsubseteq\uparrow^{[m,m]}\Gamma';K'+[m,m]\subseteq K}{\varphi;\Phi;\Gamma\vdash m:(\nu a)P\triangleleft K}$$

This concludes the interesting base case. Symmetry and transitivity are direct, and for the cases of contextual congruence, the proof is straightforward. □

And now that we can work up to the congruence relation with Lemma 4.5.13. Theorem 4.5.1 is proved by induction on $P\Rightarrow Q$. Without surprise, the most difficult case is for a communication, and it greatly relies on reliability.

Again, when considering the typing of P, the first subtyping rule has no importance. We now proceed by doing the case analysis on the rules of Figure 4.18. In order to simplify the proof, we will also consider that types and indexes invariant by subtyping (like the complexity in a server) are not renamed with subtyping. Note that this only adds cumbersome notations but it does not change the core of the proof.

- **Case** $(n:!a(\widetilde{v}).P)\mid(m:\overline{a}\langle\widetilde{e}\rangle.Q)\Rightarrow(n:!a(\widetilde{v}).P)\mid(\max(m,n):(P[\widetilde{v}:=\widetilde{e}]\mid Q))$. Consider the typing $\varphi;\Phi;\Gamma_0\mid\Delta_0,a:\forall\widetilde{i}.\mathtt{srv}^{K_a}(\widetilde{T})/(U_0\mid V_0)\vdash(n:!a(\widetilde{v}).P)\mid(m:\overline{a}\langle\widetilde{e}\rangle.Q)\triangleleft K_0\sqcup K_0'$. The first rule is the rule for parallel composition, then the derivation is split into the two following subtrees:

$$\dfrac{\dfrac{\dfrac{\dfrac{\pi_P}{(\varphi,\widetilde{i});\Phi;\Gamma_2,a:\forall\widetilde{i}.\mathtt{srv}^{K_a}(\widetilde{T})/U_2,\widetilde{v}:\widetilde{T}\vdash P\triangleleft K_a}}{\varphi;\Phi;\uparrow^{J_c}!\Gamma_2,a:\forall\widetilde{i}.\mathtt{srv}^{K_a}(\widetilde{T})/!\mathtt{In}^{[0,0]}_{J_c}.U_2\vdash !a(\widetilde{v}).P\triangleleft[0,0]}\ (2)}{\varphi;\Phi;\Gamma_1,a:\forall\widetilde{i}.\mathtt{srv}^{K_a}(\widetilde{T})/U_1\vdash !a(\widetilde{v}).P\triangleleft K_1}}{\varphi;\Phi;\uparrow^{[n,n]}\Gamma_1,a:\forall\widetilde{i}.\mathtt{srv}^{K_a}(\widetilde{T})/\uparrow^{[n,n]}U_1\vdash n:!a(\widetilde{v}).P\triangleleft K_1+[n,n]}\ (1)}{\varphi;\Phi;\Gamma_0,a:\forall\widetilde{i}.\mathtt{srv}^{K_a}(\widetilde{T})/U_0\vdash n:!a(\widetilde{v}).P\triangleleft K_0}$$

where (1) is

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

and (2) is

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

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\pi_e}{\varphi;\Phi;\Delta_2,a:\forall\widetilde{i}.\mathtt{srv}^{K_a}(\widetilde{T})/V_2\vdash\widetilde{e}:\widetilde{T}\{\widetilde{I_{\mathbb{N}}}/\widetilde{i}\}}\quad\dfrac{\pi_Q}{\varphi;\Phi;\Delta_2',a:\forall\widetilde{i}.\mathtt{srv}^{K_a}(\widetilde{T})/V_2'\vdash Q\triangleleft K_2}}{\varphi;\Phi;\uparrow^{J_c'}(\Delta_2\mid\Delta_2'),a:\forall\widetilde{i}.\mathtt{srv}^{K_a}(\widetilde{T})/\mathtt{Out}^{[0,0]}_{J_c'}.(V_2\mid V_2')\vdash\overline{a}\langle\widetilde{e}\rangle.Q\triangleleft J_c';(K_2\sqcup K_a\{\widetilde{I_{\mathbb{N}}}/\widetilde{i}\})}\ (4)}{\varphi;\Phi;\Delta_1,a:\forall\widetilde{i}.\mathtt{srv}^{K_a}(\widetilde{T})/V_1\vdash\overline{a}\langle\widetilde{e}\rangle.Q\triangleleft K_1'}}{\varphi;\Phi;\uparrow^{[m,m]}\Delta_1,a:\forall\widetilde{i}.\mathtt{srv}^{K_a}(\widetilde{T})/\uparrow^{[m,m]}V_1\vdash m:\overline{a}\langle\widetilde{e}\rangle.Q\triangleleft K_1'+[m,m]}\ (3)}{\varphi;\Phi;\Delta_0,a:\forall\widetilde{i}.\mathtt{srv}^{K_a}(\widetilde{T})/V_0\vdash m:\overline{a}\langle\widetilde{e}\rangle.Q\triangleleft K_0'}$$

where (3) is

$$\varphi;\Phi\vdash\Delta_0\sqsubseteq\uparrow^{[n,n]}\Delta_1;V_0\sqsubseteq\uparrow^{[m,m]}V_1;K_1'+[m,m]\subseteq K_0'$$

and (4) is

$$\varphi;\Phi\vdash\Delta_1\sqsubseteq\uparrow^{J_c'}(\Delta_2\mid\Delta_2')\qquad\varphi;\Phi\vdash V_1\sqsubseteq\mathtt{Out}^{[0,0]}_{J_c'}.(V_2\mid V_2')\qquad\varphi;\Phi\vDash J_c';(K_2\sqcup K_a\{\widetilde{I_{\mathbb{N}}}/\widetilde{i}\})\subseteq K_1'$$

First, by the index substitution lemma (Lemma 4.5.10), from π_P we obtain a proof:

$$\pi_P\{\widetilde{I_{\mathbb{N}}}/\widetilde{i}\}:\qquad\varphi;\Phi;\Gamma_2,a:\forall\widetilde{i}.\mathtt{srv}^{K_a}(\widetilde{T})/U_2,\widetilde{v}:\widetilde{T}\{\widetilde{I_{\mathbb{N}}}/\widetilde{i}\}\vdash P\triangleleft K_a\{\widetilde{I_{\mathbb{N}}}/\widetilde{i}\}$$

Since the index variables $\widetilde{i}$ can only be free in $\widetilde{T}$ and K_a.
Then, we know that $\Gamma_0\mid\Delta_0$ is defined. Moreover, we have

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

So, for the channel and server types, in those seven contexts, the shape of the type does not change (only the usage can change). Let us look at base types. For a context Γ, we write Γ^{Nat} the restriction of Γ to base types. Then, we have:

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

Similarly, we note Γ^{ν} the restriction of a context to its channel and server types. Thus, we have $\Gamma=\Gamma^{\nu},\Gamma^{\mathsf{Nat}}$.
So, from π_e and $\pi_P\{\widetilde{I_{\mathbb{N}}}/\widetilde{i}\}$ we obtain by subtyping:

$$\pi_P':\qquad\varphi;\Phi;\Gamma_0^{\mathsf{Nat}},\Gamma_2^{\nu},a:\forall\widetilde{i}.\mathtt{srv}^{K_a}(\widetilde{T})/U_2,\widetilde{v}:\widetilde{T}\{\widetilde{I_{\mathbb{N}}}/\widetilde{i}\}\vdash P\triangleleft K_a\{\widetilde{I_{\mathbb{N}}}/\widetilde{i}\}$$

$$\pi_e':\qquad\varphi;\Phi;\Gamma_0^{\mathsf{Nat}},\Delta_2^{\nu},a:\forall\widetilde{i}.\mathtt{srv}^{K_a}(\widetilde{T})/V_2\vdash\widetilde{e}:\widetilde{T}\{\widetilde{I_{\mathbb{N}}}/\widetilde{i}\}$$

So, we use the substitution lemma (Lemma 4.5.11) and we obtain:

$$\pi_{sub} : \qquad \varphi; \Phi; \Gamma_0^{\mathsf{Nat}}, (\Gamma_2^\nu \mid \Delta_2^\nu), a : \forall \widetilde{i}.\mathtt{srv}^{K_a}(\widetilde{T})/(U_2 \mid V_2) \vdash P[\widetilde{v} := \widetilde{e}] \triangleleft K_a\{\widetilde{I_{\mathbb{N}}}/\widetilde{i}\}$$

As previously, by subtyping from π_Q, we have:

$$\pi'_Q : \qquad \varphi; \Phi; \Gamma_0^{\mathsf{Nat}}, \Delta_2'^\nu, a : \forall \widetilde{i}.\mathtt{srv}^{K_a}(\widetilde{T})/V_2' \vdash Q \triangleleft K_2$$

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

$$\pi_{PQ} : \qquad \varphi; \Phi; \Gamma_0^{\mathsf{Nat}}, (\Gamma_2^\nu \mid \Delta_2^\nu \mid \Delta_2'^\nu), a : \forall \widetilde{i}.\mathtt{srv}^{K_a}(\widetilde{T})/(U_2 \mid V_2 \mid V_2') \vdash (P[\widetilde{v} := \widetilde{e}] \mid Q) \triangleleft K_2 \sqcup K_a\{\widetilde{I_{\mathbb{N}}}/\widetilde{i}\}$$

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

$$\dfrac{\dfrac{\pi_{PQ}}{\varphi; \Phi; \Gamma_0^{\mathsf{Nat}}, (\Gamma_2^\nu \mid \Delta_2^\nu \mid \Delta_2'^\nu), a : \forall \widetilde{i}.\mathtt{srv}^{K_a}(\widetilde{T})/(U_2 \mid V_2 \mid V_2') \vdash (P[\widetilde{v} := \widetilde{e}] \mid Q) \triangleleft K_2 \sqcup K_a\{\widetilde{I_{\mathbb{N}}}/\widetilde{i}\}}}{\varphi; \Phi; \mathcal{E} \vdash M : (P[\widetilde{v} := \widetilde{e}] \mid Q) \triangleleft (K_2 \sqcup K_a\{\widetilde{I_{\mathbb{N}}}/\widetilde{i}\}) + [M, M]}$$

where $\mathcal{E}$ is:

$$\Gamma_0^{\mathsf{Nat}}, \uparrow^{[M,M]}(\Gamma_2^\nu \mid \Delta_2^\nu \mid \Delta_2'^\nu), a : \forall \widetilde{i}.\mathtt{srv}^{K_a}(\widetilde{T})/\uparrow^{[M,M]}(U_2 \mid V_2 \mid V_2')$$

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

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

So, by Point 1 of Lemma 4.5.1, with transitivity and parallel composition of subusage, we have:

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

By Point 3 of Lemma 4.5.1, we have $!\mathtt{In}_{J_c}^{[n,n]}.U_2 \mid \mathtt{Out}_{J_c'}^{[m,m]}(V_2 \mid V_2')$ reliable. So, in particular, we have:

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

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

Thus, we deduce immediately that neither J_c or J_c' are $[\infty, \infty]$ and that

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

So, we have in particular, with Lemma 4.5.2 and Point 1 of Lemma 4.5.1 and parallel composition:

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

We also have

$$\varphi; \Phi \vDash (K_2 \sqcup K_a\{\widetilde{I_{\mathbb{N}}}/\widetilde{i}\}) + [M, M] \subseteq (J_c'; (K_2 \sqcup K_a\{\widetilde{I_{\mathbb{N}}}/\widetilde{i}\}) + [m, m]) \subseteq K_0'$$

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

$$\dfrac{\dfrac{\dfrac{\pi_{PQ}}{\varphi;\Phi;\Gamma_0^{\mathsf{Nat}},(\Gamma_2^\nu \mid \Delta_2^\nu \mid \Delta_2'^\nu),a:\forall\widetilde{i}.\mathtt{srv}^{K_a}(\widetilde{T})/(U_2 \mid V_2 \mid V_2') \vdash (P[\widetilde{v}:=\widetilde{e}] \mid Q) \triangleleft K_2 \sqcup K_a\{\widetilde{I_{\mathbb{N}}}/\widetilde{i}\}}}{\varphi;\Phi;\mathcal{E} \vdash M : (P[\widetilde{v}:=\widetilde{e}] \mid Q) \triangleleft (K_2 \sqcup K_a\{\widetilde{I_{\mathbb{N}}}/\widetilde{i}\}) + [M,M]}}{\varphi;\Phi;\mathcal{E} \vdash M : (P[\widetilde{v}:=\widetilde{e}] \mid Q) \triangleleft K_0'}$$

We also have the following derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\pi_P}{(\varphi,\widetilde{i});\Phi;\Gamma_2,a:\forall\widetilde{i}.\mathtt{srv}^{K_a}(\widetilde{T})/U_2,\widetilde{v}:\widetilde{T} \vdash P \triangleleft K_a}}{\varphi;\Phi;\uparrow^{J_c}!\Gamma_2,a:\forall\widetilde{i}.\mathtt{srv}^{K_a}(\widetilde{T})/!\mathtt{In}_{J_c}^{[0,0]}.U_2 \vdash !a(\widetilde{v}).P \triangleleft [0,0]}}{\varphi;\Phi;\uparrow^{[n,n]+J_c}!\Gamma_2,a:\forall\widetilde{i}.\mathtt{srv}^{K_a}(\widetilde{T})/!\mathtt{In}_{J_c}^{[n,n]}.U_2 \vdash n :!a(\widetilde{v}).P \triangleleft [n,n]}}{\varphi;\Phi;\Gamma_0^{\mathsf{Nat}},\uparrow^{[M,M]}!\Gamma_2^\nu,a:\forall\widetilde{i}.\mathtt{srv}^{K_a}(\widetilde{T})/!\mathtt{In}_{J_c}^{[n,n]}.U_2 \vdash n :!a(\widetilde{v}).P \triangleleft K_0}$$

So, by parallel composition of those two derivations we obtain a derivation of:

$$\varphi;\Phi;\Gamma_0^{\mathsf{Nat}},\uparrow^{[M,M]}(!\Gamma_2^\nu \mid \Gamma_2^\nu \mid \Delta_2^\nu \mid \Delta_2'^\nu),a:\forall\widetilde{i}.\mathtt{srv}^{K_a}(\widetilde{T})/!\mathtt{In}_{J_c}^{[n,n]}.U_2 \mid (\uparrow^{[M,M]}(U_2 \mid V_2 \mid V_2'))$$
$$\vdash (n :!a(\widetilde{v}).P) \mid M : (P[\widetilde{v}:=\widetilde{e}] \mid Q) \triangleleft K_0 \sqcup K_0'$$

By Point 2 of Lemma 4.5.1, there exists W such that:

$$\varphi;\Phi \vdash U_0 \mid V_0 \longrightarrow^* W \qquad \varphi;\Phi \vdash W \sqsubseteq !\mathtt{In}_{J_c}^{[n,n]}.U_2 \mid \uparrow^{[M,M]}(U_2 \mid V_2 \mid V_2')$$

So, by subtyping we have a proof:

$$\varphi;\Phi;\Gamma_0^{\mathsf{Nat}},\Gamma_0^\nu \mid \Delta_0^\nu,a:\forall\widetilde{i}.\mathtt{srv}^{K_a}(\widetilde{T})/W \vdash (n :!a(\widetilde{v}).P) \mid M : (P[\widetilde{v}:=\widetilde{e}] \mid Q) \triangleleft K_0 \sqcup K_0'$$

This concludes this case.

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

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

where (3) is

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

and (4) is

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

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\pi_e}{\varphi;\Phi;\Delta_2,a:\mathtt{ch}(\widetilde{T})/V_2\vdash\widetilde{e}:\widetilde{T}}\quad\dfrac{\pi_Q}{\varphi;\Phi;\Delta_2',a:\mathtt{ch}(\widetilde{T})/V_2'\vdash Q\triangleleft K_2'}}{\varphi;\Phi;\uparrow^{J_c'}(\Delta_2\mid\Delta_2'),a:\mathtt{ch}(\widetilde{T})/\mathtt{Out}_{J_c'}^{[0,0]}.(V_2\mid V_2')\vdash\overline{a}\langle\widetilde{e}\rangle.Q\triangleleft J_c';K_2'}\ (1)}{\varphi;\Phi;\Delta_1,a:\mathtt{ch}(\widetilde{T})/V_1\vdash\overline{a}\langle\widetilde{e}\rangle.Q\triangleleft K_1'}}{\varphi;\Phi;\uparrow^{[m,m]}\Delta_1,a:\mathtt{ch}(\widetilde{T})/\uparrow^{[m,m]}V_1\vdash m:\overline{a}\langle\widetilde{e}\rangle.Q\triangleleft K_1'+[m,m]}\ (2)}{\varphi;\Phi;\Delta_0,a:\mathtt{ch}(\widetilde{T})/V_0\vdash m:\overline{a}\langle\widetilde{e}\rangle.Q\triangleleft K_0'}$$

where (1) is

$$\varphi;\Phi\vdash\Delta_1\sqsubseteq\uparrow^{J_c'}(\Delta_2\mid\Delta_2')\qquad\varphi;\Phi\vdash V_1\sqsubseteq\mathtt{Out}_{J_c'}^{[0,0]}.(V_2\mid V_2')\qquad\varphi;\Phi\vDash J_c';K_2'\subseteq K_1'$$

and (2) is

$$\varphi;\Phi\vdash\Delta_0\sqsubseteq\uparrow^{[n,n]}\Delta_1;V_0\sqsubseteq\uparrow^{[m,m]}V_1;K_1'+[m,m]\subseteq K_0'$$

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

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

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

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

So, from π_e and π_P we obtain by subtyping:

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

So, we use the substitution lemma (Lemma 4.5.11) and we obtain:

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

As previously, by subtyping from π_Q, we have:

$$\varphi;\Phi;\Gamma_0^{\mathsf{Nat}},\Delta_2'^{\nu},a:\mathtt{ch}(\widetilde{T})/V_2'\vdash Q\triangleleft K_2'$$

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

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

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

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

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

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

So, by Point 1 of Lemma 4.5.1, with transitivity and parallel composition of subusage, we have:

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

By Point 3 of Lemma 4.5.1, we have $\mathtt{In}_{J_c}^{[n,n]}.U_2 \mid \mathtt{Out}_{J_c'}^{[m,m]}(V_2 \mid V_2')$ reliable. So, in particular, we have:

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

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

Thus, we deduce that

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

So, we have in particular, with Lemma 4.5.2 and Point 1 of Lemma 4.5.1 and parallel composition:

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

We also have

$$\varphi; \Phi \vDash K_2 + [M,M] \subseteq J_c; K_2 + [n,n] \subseteq K_0 \qquad \varphi; \Phi \vDash K_2' + [M,M] \subseteq J_c'; K_2' + [m,m] \subseteq K_0'$$

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

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

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

By Point 2 of Lemma 4.5.1, there exists W such that:

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

So, by subtyping we have a proof:

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

This concludes this case.

- **Case** $\mathtt{match}\ \mathtt{s}(e)\ \{\underline{0} \mapsto P;;\ \mathtt{s}(x) \mapsto Q\} \Rightarrow Q[x := e]$. The case when we match on the integer 0 is similar, so we only present the successor case. Suppose given a derivation for $\mathtt{match}\ \mathtt{s}(e)\ \{\underline{0} \mapsto P;;\ \mathtt{s}(x) \mapsto Q\} \triangleleft K$. Then the derivation has the shape:

$$\dfrac{\dfrac{\dfrac{\dfrac{\pi_e}{\varphi; \Phi; \Delta \vdash e : \mathsf{Nat}[I', J']}}{\varphi; \Phi; \Delta \vdash \mathtt{s}(e) : \mathsf{Nat}[I'+1, J'+1]} \qquad \varphi; \Phi \vdash \Gamma \sqsubseteq \Delta; \mathsf{Nat}[I'+1, J'+1] \sqsubseteq \mathsf{Nat}[I, J]}{\varphi; \Phi; \Gamma \vdash \mathtt{s}(e) : \mathsf{Nat}[I, J]} \qquad \pi_P \qquad \pi_Q}{\mathtt{match}\ \mathtt{s}(e)\ \{\underline{0} \mapsto P;;\ \mathtt{s}(x) \mapsto Q\} \triangleleft K}$$

where π_Q is a derivation of $\phi; (\Phi, J \geq 1); \Gamma, x : \mathsf{Nat}[I{-}1][J{-}1] \vdash Q \triangleleft K$, and π_P is a typing derivation for P that does not interest us in this case.

By definition of subtyping, we have:

$$\varphi; \Phi \vDash I \leq I' + 1 \qquad \varphi; \Phi \vDash J' + 1 \leq J$$

From this, we deduce the following constraints:

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

Thus, with the subtyping rule and the derivation π_e we obtain:

$$\varphi; \Phi; \Delta \vdash e : \mathsf{Nat}[I{-}1, J{-}1]$$

Then, by Lemma 4.5.9, from π_Q we obtain a derivation of $\varphi; \Phi; \Gamma, x : \mathsf{Nat}[I{-}1][J{-}1] \vdash Q \triangleleft K$. By the substitution lemma (Lemma 4.5.11), we obtain $\varphi; \Phi; \Gamma \vdash Q[x := e] \triangleleft K$. This concludes this case.

- **Case** $n : P \Rightarrow n : Q$ with $P \Rightarrow Q$. Suppose that $\varphi; \Phi; \uparrow^{[n,n]}\Gamma \vdash n : P \triangleleft K + [n, n]$. Then, the derivation has the shape:

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

  By Lemma 4.5.7, if $\uparrow^{[n,n]}\Gamma$ is reliable then Γ is reliable. By induction hypothesis, we have a derivation $\varphi; \Phi; \Gamma' \vdash Q \triangleleft K$ with $\varphi; \Phi \vdash \Gamma \longrightarrow^* \Gamma'$.

  We give the proof:

$$\frac{\varphi; \Phi; \Gamma' \vdash Q \triangleleft K}{\varphi; \Phi; \uparrow^{[n,n]}\Gamma' \vdash n : Q \triangleleft K + [n, n]}$$

  And we have indeed $\varphi; \Phi \vdash \uparrow^{[n,n]}\Gamma \longrightarrow^* \uparrow^{[n,n]}\Gamma'$ by Lemma 4.5.7.

- **Case** $P \Rightarrow Q$ with $P \equiv P'$, $P' \Rightarrow Q'$ and $Q \equiv Q'$. Suppose that $\varphi; \Phi; \Gamma \vdash P \triangleleft K$. By Lemma 4.4.8, we have $\varphi; \Phi; \Gamma \vdash P' \triangleleft K$. By induction hypothesis, we obtain $\varphi; \Phi; \Gamma' \vdash Q' \triangleleft K$ with $\varphi; \Phi \vdash \Gamma \longrightarrow^* \Gamma'$. Then, again by Lemma 4.4.8, we have $\varphi; \Phi; \Gamma' \vdash Q \triangleleft K$. This concludes this case.

This concludes the proof of Theorem 4.5.1.
Finally, we conclude with the following theorem:

Theorem 4.5.2. *Let P be an annotated process and n be its global parallel complexity. Then, if we have $\varphi; \Phi; \Gamma \vdash P \triangleleft [I, J]$, then $\varphi; \Phi \vDash J \geq n$. Moreover, if Γ does not contain any integer variables, we have $\varphi; \Phi \vDash I \leq n$.*

Proof. By Theorem 4.5.1, all reductions from P using $\Rightarrow$ preserve the typing. The context may be reduced too, but as reducibility does not harm reliability, we can still apply the subject reduction through all the reduction steps of $\Rightarrow$ Moreover, for a process Q, if we have a typing $\varphi; \Phi; \Gamma \vdash Q \triangleleft [I, J]$, then $J \geq \mathcal{C}_\ell(Q)$. Indeed, a constructor $n : P$ forces an increment of the complexity of n both in typing and in the definition of $\mathcal{C}_\ell(Q)$, and for parallel composition the typing imposes a complexity greater than the maximum as in the definition for $\mathcal{C}_\ell(Q)$. Thus, J is indeed a bound on the parallel complexity by definition. As for the lower bound, one can see

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

4.5.3 Examples

Let us also present again how sizes and polymorphism over indices in servers can type processes defined by replication such as the factorial in the context of usages. On such simple replicated input, the similarities with Section 4.4 should be clear.

Example 4.5.6 (Factorial)**.** *Suppose given a function on expressions for multiplication:* $\texttt{mult} : \mathsf{Nat}[I, J] \times \mathsf{Nat}[I', J'] \to \mathsf{Nat}[I * I', J * J']$. *In practice, this should be encoded as a server in π-calculus, but for the sake of simplicity we consider it as a function. We will describe the factorial and count the number of multiplications with* $\texttt{tick}$. *For the sake of conciseness, we write* $\mathsf{Nat}[I]$ *to denotes* $\mathsf{Nat}[I, I]$. *We use the usual notation* $I!$ *to represent the factorial function in indices. The process representing factorial and its typing derivation are given in Figure 4.36. We denote* T *the following type:*

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

This type is reliable and it would be reliable even if composed with any kind of output $\mathit{Out}_0^{A_o}$ *if we want to call this server. Let us denote:*

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

and we also pose:

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

where $S_1(i)$ *and* $S_2(i)$ *are the expected separation of the usage. This type* $S(i{-}1)$ *is reliable under* $(i); (i \geq 1)$. *Thus, we give the typing described in Figure 4.36. From the type of* f, *we see on its complexity* $[0, i]$ *that it does at most a linear number of multiplications.*

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

Example 4.5.7 (Deadlock)**.** *Let us consider the process*

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

Remark that this process is similar to the one we described in Section 4.1.3. In order to understand the constraints needed to type this process, we will give a typing with variables for obligation and capacity, and we will look at which values those variables can take. This is described in Figure 4.37.

First, as a *and* b *have exactly the same behaviour, they must have the same typing. And from just this, we already have some constraints to satisfy. Indeed, because of reliability, we have:*

$$B_o \subseteq A_o \oplus I_c \qquad A_o \subseteq B_o \oplus J_c$$

Moreover, in order to continue the typing, we must have

$$\mathit{In}_{I_c}^{A_o} \sqsubseteq \mathit{In}_{I_c'}^{[0,0]} \qquad \mathit{Out}_{J_c}^{B_o} \sqsubseteq \uparrow^{I_c'}\uparrow^{[1,1]}\mathit{Out}_{J_c'}^{[0,0]}$$

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

$$\dfrac{\dfrac{\dfrac{}{i;\cdot;n:\mathsf{Nat}[i]\vdash n:\mathsf{Nat}[i]} \quad \dfrac{\dfrac{}{i;(i\leq 0)\vDash i! = 1}}{i;i\leq 0;f:T',n:\mathsf{Nat}[i],r:S_1(i)\vdash \overline{r}\langle\underline{1}\rangle \lhd [0,i]}}{i;\cdot;f:T',n:\mathsf{Nat}[i],r:S_1(i)\vdash \texttt{match}\ n\ \{\underline{0}\mapsto\overline{r}\langle\underline{1}\rangle;;\ \mathtt{s}(m)\mapsto(\nu r')\cdots\}\lhd[0,i]}\ \pi_1}{\cdot;\cdot;f:T\vdash !f(n,r).\texttt{match}\ n\ \{\underline{0}\mapsto\overline{r}\langle\underline{1}\rangle;;\ \mathtt{s}(m)\mapsto(\nu r')(\overline{f}\langle m,r'\rangle\mid r'(x).\texttt{tick}.\overline{r}\langle mult(n,x)\rangle)\}\lhd[0,0]}$$

with π_1 :

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{(i;i\geq 1)\vDash i*(i-1)! = i!}}{\varphi;\Phi;n:\mathsf{Nat}[i],x:\mathsf{Nat}[(i-1)!]\vdash mult(n,x):\mathsf{Nat}[i!]}}{\varphi;\Phi;n:\mathsf{Nat}[i],x:\mathsf{Nat}[(i-1)!],r:\mathtt{ch}(\mathsf{Nat}[i!])/\mathtt{Out}_0^{[0,0]}\vdash\overline{r}\langle mult(n,x)\rangle\lhd[0,0]}}{\varphi;\Phi;\Gamma\vdash(m,r'):(\mathsf{Nat}[i-1],S_1(i-1)) \quad \varphi;\Phi;n:\mathsf{Nat}[i],x:\mathsf{Nat}[(i-1)!],r:\mathtt{ch}(\mathsf{Nat}[i!])/\mathtt{Out}_0^{[1,1]}\vdash\texttt{tick}.\cdots\lhd[1,1]}}{\varphi;\Phi;m:\mathsf{Nat}[i-1],r':S_1(i-1),f:T_1\vdash\overline{f}\langle m,r'\rangle\lhd[0,i] \quad \varphi;\Phi;n:\mathsf{Nat}[i],r':S_2(i-1),r:S_1(i)\vdash r'(x).\cdots\lhd[0,i]}}{\varphi;\Phi;n:\mathsf{Nat}[i],m:\mathsf{Nat}[i-1],r:S_1(i),f:T',r':S(i-1)\vdash\overline{f}\langle m,r'\rangle\mid r'(x).\texttt{tick}.\overline{r}\langle mult(n,x)\rangle\lhd[0,i]}}{i;i\geq 1;n:\mathsf{Nat}[i],m:\mathsf{Nat}[i-1],r:S_1(i),f:T'\vdash(\nu r')(\overline{f}\langle m,r'\rangle\mid r'(x).\texttt{tick}.\overline{r}\langle mult(n,x)\rangle)\lhd[0,i]}$$

Figure 4.36: Representation and Typing of Factorial

$$\dfrac{\cdot;\cdot;a:\mathtt{ch}()/\mathtt{In}_{I_c}^{A_o},b:\mathtt{ch}()/\mathtt{Out}_{J_c}^{B_o}\vdash a().\texttt{tick}.\overline{b}\langle\rangle\lhd K_1 \quad \cdot;\cdot;a:\mathtt{ch}()/\mathtt{Out}_{J_c}^{B_o},b:\mathtt{ch}()/\mathtt{In}_{I_c}^{A_o}\vdash b().\texttt{tick}.\overline{a}\langle\rangle\lhd K_2}{\dfrac{\cdot;\cdot;a:\mathtt{ch}()/(\mathtt{In}_{I_c}^{A_o}\mid\mathtt{Out}_{J_c}^{B_o}),b:\mathtt{ch}()/(\mathtt{Out}_{J_c}^{B_o}\mid\mathtt{In}_{I_c}^{A_o})\vdash a().\texttt{tick}.\overline{b}\langle\rangle\mid b().\texttt{tick}.\overline{a}\langle\rangle\lhd K_1\sqcup K_2}{\cdot;\cdot;\cdot\vdash(\nu a)(\nu b)(a().\texttt{tick}.\overline{b}\langle\rangle\mid b().\texttt{tick}.\overline{a}\langle\rangle)\lhd K_1\sqcup K_2}}$$

Figure 4.37: Typing Constraints for Example 4.5.7

This gives us the additional constraints:

$$[0,0]\subseteq A_o \qquad I_c\leq I_c' \qquad ([1,1]+I_c')\subseteq B_o \qquad J_c\leq J_c'$$

So, if we put it together, we have:

$$([1,1]+I_c)\subseteq([1,1]+I_c')\subseteq B_o\subseteq A_o\oplus I_c$$

As $[0,0]\subseteq A_o$ *we have* $\mathsf{Left}(A_o)=0$. *In order to have* $\mathsf{Right}([1,1]+I_c)\leq\mathsf{Right}(A_o\oplus I_c)$ *then we are forced to take* $I_c=\infty$ *or* $I_c=[I,\infty]$ *for some* I. *If we take such a capacity, this could induce a infinite upper bound. However, recall that we have the special case* $[\infty,\infty];K=[0,0]$. *So, if we can give an infinite lower bound to this capacity, we recover the complexity* 0 *of this deadlock. In fact, this is possible as described in Figure 4.38. Thus, the capacity* $[\infty,\infty]$, *describing input or output that will never be reduced, allows us to derive a complexity of zero.*

Example 4.5.8. *We describe here informally an example for which our system can give a complexity, but fails to catch a precise bound, compared to Section 4.4. Let us consider the process:*

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

This process has complexity 2, *and the type system of Section 4.4 can infer this complexity. However, if we want to give a usage to the server* a, *we must have a usage:*

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

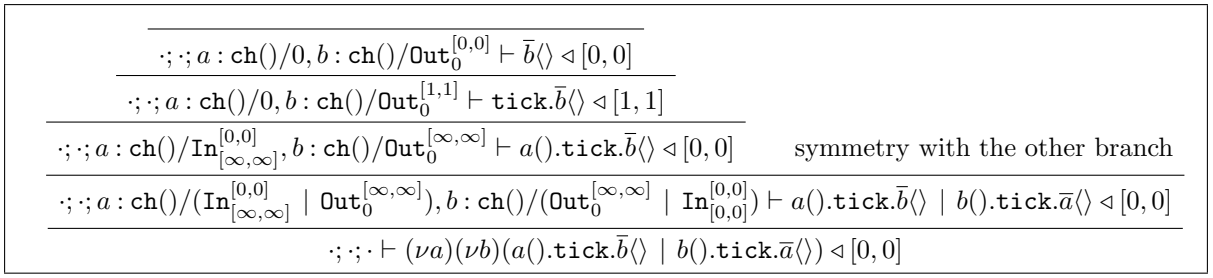

Figure 4.38: Typing of Example 4.5.7

We took as obligations the number of ticks before the action, and as capacity the minimal number for which we have reliability. So in particular, because of the capacity 1 *in the usage* $\mathtt{Out}_1^{[0,0]}$*, typing the recursive call* $\overline{a}\langle m\rangle$ *increases the complexity by one, and so typing* n *recursive calls generates a complexity of* n *in the type system. So, in the usage setting, the complexity of this process can only be bounded by* 10*. Overall, this type system may not behave well when there are more than one replicated input process on each server channel, since an imprecision on a capacity for a recursive call leads to an overall imprecision depending on the number of recursive calls. This issue is the main source of imprecision we found with respect to the type system of Section 4.4.*

The Need for Intervals in Usages

In this paragraph, we describe informally on an example where the use of intervals is important in our work. The need for an interval capacity is apparent for the process

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

where v has type $\mathsf{Nat}[0,1]$. Indeed, depending on the value of v (which may be statically unknown), an output on a may be available at time 0 or 1. Thus, the input usage on a should have a capacity interval $[0,1]$. As a result, the obligation of the output usage on b should also be an interval $[0,1]$.

Now, one might think that we can assume that lower-bounds are always 0 (or ∞, to consider processes like Example 4.5.7) and omit lower-bounds, since we are mainly interested in an *upper-bound* of the parallel complexity. Information about lower-bounds is, however, actually required for precise reasoning on upper-bounds. For example, consider the following process:

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

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

If we set lower-bounds to 0 and assign to a the usage $\mathtt{In}_{[0,1]}^{[0,0]} \mid \mathtt{Out}_0^{[0,1]}$ to a, then the usage of b can only be: $\mathtt{Out}_1^{[0,1]} \mid \mathtt{In}_1^{[0,1]}$. Note that according to the imprecise usage of a, the output on b may become ready at time 0 and then have to wait for one time unit until the input on b becomes ready; thus, the capacity of the output on b is 1, instead of $[0,0]$. An upper-bound of the parallel complexity would therefore be inferred to be $1+1=2$ (because the usages tell us that the lefthand side process may wait for one time unit at a, and then for another time unit at b), which is too imprecise. The problem described here was already in [72], even if it was not shown in this paper. Overall, we are certain that even by trying to modify slightly the definition of [72] in order to account for this imprecision, there is no way to still have a sound type system without lower-bounds. This is because the proof of soundness relies on important

properties of subusage (namely Lemma 4.5.1 and Lemma 4.5.2), and in order to satisfy this lemma, all the definitions must be carefully designed.

Elements of Comparison with Section 4.4

Finally, we give intuitively a description of how to simulate types from Section 4.4 in a linear setting with usage. We say that a process has a linear use of channels if it uses channel names at most one time for input and at most one time for output. For servers, we suppose that the replicated input is once and for all defined at the beginning of a process, and as free variables it can only use others servers. The main idea is to represent the channel type $\mathtt{ch}_I(\widetilde{T})$ by a type $\mathtt{ch}(\widetilde{\underline{T}})/(\mathtt{In}_{J_c^1}^{[I_1,I_1]} \mid \mathtt{Out}_{J_c^2}^{[I_2,I_2]})$ where either J_c^1 is 0 and then $I_1 \leq I$, either $J_c^1 = [J_1, J_1]$ and then $I_1 + J_1 \leq I$. We have the same thing for J_c^2 and I_2. To be more precise, the typing in our setting should be a non-deterministic choice (using $+$) over such usages, and the capacity should adapt to the obligation of the dual action in order to be reliable. So, for example if $I_1 \leq I_2$, then we would take: $\mathtt{ch}(\widetilde{\underline{T}})/(\mathtt{In}_{[I_2-I_1,I_2-I_1]}^{[I_1,I_1]} \mid \mathtt{Out}_0^{I_2,I_2})$. Note that this shape of type adapts well to the way time is delayed in Section 4.4. For example, the `tick` constructor makes the time advance by 1 in this previous setting, and in the usage setting, then we would obtain the usage $(\mathtt{In}_{[I_2-I_1,I_2-I_1]}^{[I_1+1,I_1+1]} \mid \mathtt{Out}_0^{I_2+1,I_2+1})$ and we still have $I_2 - I_1 = (I_2 + 1) - (I_1 + 1)$.

In the same way, when doing an output (or input), the time was delayed by I. Here, with usages, it would be delayed by J_c which is, by definition, a delay of the shape $\uparrow^{[J,J]}$ with $J \leq I$. So, we would keep the invariant that our time annotation have the shape of singleton interval with a smaller value than the time annotation in the setting of Section 4.4.

For servers, in the linear setting, types had the shape: $\forall\widetilde{i}.\mathtt{srv}_0^K(\widetilde{T})$ where the time is zero because of the assumption given above on servers. So, in the usage setting it would be:

$$\forall\widetilde{i}.\mathtt{srv}^{[0,K]}(\widetilde{\underline{T}})/!\mathtt{In}_{\infty}^{[0,0]}.!\mathtt{Out}_0^{[0,\infty]} \mid !\mathtt{Out}_0^{[0,\infty]}$$

Note that this usage is reliable. The main point here is this infinite capacity for input. Please note that because of our input rule for servers, it does not generate an infinite complexity. However, it imposes a delaying $\uparrow^{[0,\infty]}!\Gamma$ in the context. Because of the shape we gave to types, it means that the context can only have outputs for other servers as free variables, but this was the condition imposed by linearity, and it is similar to the time invariant hypothesis of Figure 4.21. As an example, the bitonic sort described before could be typed similarly in the usage setting with this kind of type. This type corresponds also to the type we used for Example 4.5.6.

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

However, the transformation described in this section may not be possible if we lose some precision on the sizes of values. Indeed, we showed in the previous example that an imprecision on a value can lead to a need for a non-singleton obligation in a usage, which contradicts the typing we want to give. However, it is not clear if this imprecise obligation leads to a less precise complexity bound with regards to Section 4.4. We believe it may depend on the context, and sometimes usages will be more efficient, and sometimes not. However, what we know for certain is that the usage type system is not theoretically limited by the time uniqueness presented in Example 4.4.2.

Chapter 5

Perspectives

We see several perspectives for this thesis:

- A direct perspective for our type system for π-calculus would be to study type inference for span. For the type system of Section 4.4, it is possible to draw inspiration from usual sized types inference algorithm [6], as for work. The main difference is to find a way to derive the constraints obtained because of the advance of time (the $\langle\cdot\rangle_{-I}$ operation), as in the examples of Section 4.4.1. For the usage type system, we could build on previous work by Naoki Kobayashi on type inference for usages [74, 77] in order to derive the constraints.

  Then, once we obtain this set of constraints, an interesting question is how to solve them in practice. This is usually done by chosing a particular shape for index functions (for example, all functions are polynomials of degree two), and then transforming the inequations on functions to linear inequations. In the analysis of span, the use of logarithms should be important, and so it would be interesting to explore the literature on how to efficiently solve constraints on logarithmic functions, see for instance the recent work [64].

- If we think of our notions of complexity in terms of circuits, then work corresponds to *size* and span corresponds to *depth*. Thus, another interesting notion, corresponding intuitively to the *width*, would be the number of processors needed to obtain the span. This notion seems harder to capture easily in a small-step semantics. An idea could be to consider those kinds of reduction for `tick`:

  $$(\nu\widetilde{a})(n : \texttt{tick}.P \mid Q), \rho \Rightarrow (\nu\widetilde{a})(n+1 : P \mid Q), \rho[n \mapsto \rho(n)+1]$$

  With the intuition that $\rho(n)$ gives the number of computations that happen simultaneously at time n. Then, the width of P would be

  $$\max\{\rho(i) \mid P, (fun\ n \mapsto 0) \Rightarrow^* Q, \rho \text{ and } i \in \mathbb{N}\}$$

  Again, sized types should be useful as the width of a function may depend on the size of input, but it seems that an adaptation of our previous type system would not be sufficient for a width analysis.

- An interesting extension of our type systems would be to consider amortized complexity analysis, which could be especially useful if we add trees in the data-types. Some recent work on resource-aware types for parallel programs [37] could be studied to see if the addition of potential in our type systems is feasible.

- Our work is especially focused on π-calculus, however, in particular for the type system of Section 4.4, working on the whole π-calculus may be difficult. It would be interesting to see if our approach can lead to better results on theoretically simpler parallel languages (a language with fork [54], or a functional language with parallel computations [59]), or even other expressive approaches of parallel computations (such as actor-based language [80]).
- Finally, an interesting approach could be to go back to ICC, and to work on characterizations of complexity classes in the π-calculus with our approach. A linear logic approach has been used for session types [31], and safe recursion lead to results on parallel complexity [45], but otherwise this problem has not be much studied.

Bibliography

[1] Alejandro Aguirre, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Pierre-Yves Strub. A relational logic for higher-order programs. *Journal of Functional Programming*, 29, 2019.

[2] Selim G. Akl. *Encyclopedia of Parallel Computing*, chapter Bitonic Sort, pages 139–146. Springer US, Boston, MA, 2011.

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

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

[5] Sandra Alves, Maribel Fernández, Mário Florido, and Ian Mackie. The power of linear functions. In *Computer Science Logic*, pages 119–134. Springer Berlin Heidelberg, 2006.

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

[7] Martin Avanzini, Ugo Dal Lago, and Alexis Ghyselen. Type-based complexity analysis of probabilistic functional programs. In *34th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2019*, pages 1–13. IEEE, 2019.

[8] Patrick Baillot. On the expressivity of elementary linear logic: Characterizing ptime and an exponential time hierarchy. *Information and Computation*, 241:3–31, 2015.

[9] Patrick Baillot, Gilles Barthe, and Ugo Dal Lago. Implicit Computational Complexity of Subrecursive Definitions and Applications to Cryptographic Proofs (Long version). Research report, ENS Lyon, September 2015.

[10] Patrick Baillot, Erika De Benedetti, and Simona Ronchi Della Rocca. Characterizing Polynomial and Exponential Complexity Classes in Elementary Lambda-Calculus. In *8th IFIP International Conference on Theoretical Computer Science (TCS)*, Theoretical Computer Science, pages 151–163, Sep 2014.

[11] Patrick Baillot, Marco Gaboardi, and Virgile Mogbil. A polytime functional language from light linear logic. In *European Symposium on Programming*, pages 104–124. Springer, 2010.

[12] Patrick Baillot and Alexis Ghyselen. Combining Linear Logic and Size Types for Implicit Complexity. In *27th EACSL Annual Conference on Computer Science Logic (CSL 2018)*, volume 119 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 9:1–9:21, 2018.

[13] Patrick Baillot and Alexis Ghyselen. Combining linear logic and size types for implicit complexity. *Theor. Comput. Sci.*, 813:70–99, 2020.

[14] Patrick Baillot and Alexis Ghyselen. Types for complexity of parallel computation in pi-calculus. In *Programming Languages and Systems - 30th European Symposium on Programming, ESOP 2021*, volume 12648 of *Lecture Notes in Computer Science*, pages 59–86. Springer, 2021.

[15] Patrick Baillot, Alexis Ghyselen, and Naoki Kobayashi. Sized types with usages for parallel complexity of pi-calculus processes, 2021.

[16] Patrick Baillot and Damiano Mazza. Linear logic by levels and bounded time complexity. *Theoretical Computer Science*, 411(2):470–503, 2010.

[17] Patrick Baillot and Kazushige Terui. A feasible algorithm for typing in elementary affine logic. In *TLCA*, volume 5, pages 55–70. Springer, 2005.

[18] Gilles Barthe, Raphaëlle Crubillé, Ugo Dal Lago, and Francesco Gavazzo. On the versatility of open logical relations. In Peter Müller, editor, *Programming Languages and Systems*, pages 56–83, Cham, 2020. Springer International Publishing.

[19] Gilles Barthe, Marco Gaboardi, Emilio Jesùs Gallego Arias, Justin Hsu, Aaron Roth, and Pierre-Yves Strub. Higher-order approximate relational refinement types for mechanism design and differential privacy. In *POPL42*, volume 50, pages 55–68. ACM, 2015.

[20] Stephen Bellantoni and Stephen Cook. A new recursion-theoretic characterization of the polytime functions. In *Proceedings of the twenty-fourth annual ACM symposium on Theory of computing*, pages 283–293. ACM, 1992.

[21] Laura Bocchi, Weizhen Yang, and Nobuko Yoshida. Timed multiparty session types. In Paolo Baldan and Daniele Gorla, editors, *CONCUR 2014 – Concurrency Theory*, pages 419–434, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[22] Flavien Breuvart and Ugo Dal Lago. On intersection types and probabilistic lambda calculi. In *PPDP20*, pages 8:1–8:13, 2018.

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

[24] Krishnendu Chatterjee, Hongfei Fu, and Aniket Murhekar. Automated recurrence analysis for almost-linear expected-runtime bounds. In *Computer Aided Verification*, pages 118–139. Springer International Publishing, 2017.

[25] Krishnendu Chatterjee, Petr Novotný, and Dorde Zikelic. Stochastic Invariants for Probabilistic Termination. In *POPL44*, pages 145–160, 2017.

[26] Mario Coppo and Mariangiola Dezani-Ciancaglini. A new type assignment for lambda-terms. *Archiv fur mathematische Logik und Grundlagenforschung*, 19(1):139–156, 1978.

[27] Patrick Cousot and Radhia Cousot. Abstract interpretation frameworks. *Journal of logic and computation*, 2(4):511–547, 1992.

[28] Ugo Dal Lago, Marc de Visme, Damiano Mazza, and Akira Yoshimizu. Intersection types and runtime errors in the pi-calculus. *Proc. ACM Program. Lang.*, 3(POPL), January 2019.

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

[30] Ugo Dal Lago and Charles Grellois. Probabilistic termination by monadic affine sized typing. In *Programming Languages and Systems*, pages 393–419. Springer Berlin Heidelberg, 2017.

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

[32] Ugo Dal Lago and Barbara Petit. The geometry of types. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'13, Proceedings*, pages 167–178. ACM, 2013.

[33] Ugo Dal Lago and Paolo Parisen Toldin. A higher-order characterization of probabilistic polynomial time. In *International Workshop on Foundational and Practical Aspects of Resource Analysis*, pages 1–18. Springer, 2011.

[34] Vincent Danos and Jean-Baptiste Joinet. Linear logic and elementary time. *Information and Computation*, 183(1):123–137, 2003.

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

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

[37] Ankush Das, Jan Hoffmann, and Frank Pfenning. Work analysis with resource-aware session types. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '18, page 305–314, New York, NY, USA, 2018. Association for Computing Machinery.

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

[39] Rowan Davies and Frank Pfenning. Intersection types and computational effects. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, page 198–208, New York, NY, USA, 2000. Association for Computing Machinery.

[40] Daniel De Carvalho. Execution time of lambda-terms via denotational semantics and intersection types. *Mathematical Structures in Computer Science*, 28(7):1169–1203, 2018.

[41] Pierpaolo Degano, Fabio Gadducci, and Corrado Priami. A causal semantics for ccs via rewriting logic. *Theoretical Computer Science*, 275(1):259 – 282, 2002.

[42] Pierpaolo Degano, Fabio Gadducci, and Corrado Priami. Causality and replication in concurrent processes. In *Perspectives of System Informatics*, pages 307–318. Springer Berlin Heidelberg, 2003.

[43] Pierpaolo Degano and Corrado Priami. Causality for mobile processes. In *Automata, Languages and Programming*, pages 660–671. Springer Berlin Heidelberg, 1995.

[44] Romain Demangeon, Daniel Hirschkoff, Naoki Kobayashi, and Davide Sangiorgi. On the complexity of termination inference for processes. In *Trustworthy Global Computing, Third Symposium, TGC 2007*, volume 4912 of *Lecture Notes in Computer Science*, pages 140–155. Springer, 2007.

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

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

[47] Mariangiola Dezani-Ciancaglini and Ugo De'Liguoro. Sessions and session types: An overview. In *International Workshop on Web Services and Formal Methods*, pages 1–28. Springer, 2009.

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

[49] Tim Freeman and Frank Pfenning. Refinement types for ML. In *PLDI91*, pages 268–277, 1991.

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

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

[52] Jean-Yves Girard. Light linear logic. *Information and Computation*, 143(2):175–204, 1998.

[53] Dennis Griffith and Elsa L. Gunter. Liquidpi: Inferrable dependent session types. In *NASA Formal Methods*, pages 185–197. Springer Berlin Heidelberg, 2013.

[54] Emmanuel Hainry, Jean-Yves Marion, and Romain Péchoux. Type-based complexity analysis for fork processes. In *Foundations of Software Science and Computation Structures - 16th International Conference, FOSSACS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013*, volume 7794 of *Lecture Notes in Computer Science*, pages 305–320. Springer, 2013.

[55] Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, 2012.

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

[57] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. Resource aware ML. In *Computer Aided Verification - 24th International Conference, CAV 2012*, volume 7358 of *Lecture Notes in Computer Science*, pages 781–786. Springer, 2012.

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

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

[60] Martin Hofmann. A mixed modal/linear lambda calculus with applications to bellantoni-cook safe recursion. In *Computer Science Logic*, pages 275–294. Springer Berlin Heidelberg, 1998.

[61] Martin Hofmann. Linear types and non-size-increasing polynomial time computation. *Information and Computation*, 183(1):57–85, 2003.

[62] Martin Hofmann and Steffen Jost. Static prediction of heap space usage for first-order functional programs. In *Conference Record of POPL 2003: The 30th SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 185–197. ACM, 2003.

[63] Martin Hofmann and Steffen Jost. Type-based amortised heap-space analysis. In *European Symposium on Programming*, pages 22–37. Springer, 2006.

[64] Martin Hofmann, Lorenz Leutgeb, Georg Moser, David Obwaller, and Florian Zuleger. Type-based analysis of logarithmic amortised complexity. *CoRR*, abs/2101.12029, 2021.

[65] Martin Hofmann and Dulma Rodriguez. Automatic Type Inference for Amortised Heap-Space Analysis. In *ESOP22*, volume 7792, pages 593–613, 2013.

[66] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. *J. ACM*, 63(1), March 2016.

[67] 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*, pages 410–423. ACM, 1996.

[68] Steffen Jost. Automated amortised analysis, September 2010.

[69] Steffen Jost, Kevin Hammond, Hans-Wolfgang Loidl, and Martin Hofmann. Static determination of quantitative resource usage for higher-order programs. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, pages 223–236. ACM, 2010.

[70] Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Federico Olmedo. Weakest precondition reasoning for expected run–times of probabilistic programs. In *Programming Languages and Systems*, pages 364–389. Springer Berlin Heidelberg, 2016.

[71] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

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

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

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

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

[76] Naoki Kobayashi, Benjamin C. Pierce, and David N. Turner. Linearity and the pi-calculus. *ACM Trans. Program. Lang. Syst.*, 21(5):914–947, sep 1999.

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

[78] Yves Lafont. Soft linear logic and polynomial time. *Theoretical Computer Science*, 318(1-2):163–180, 2004.

[79] Ugo Dal Lago and Barbara Petit. Linear dependent types in a call-by-value scenario. *Science of Computer Programming*, 84:77–100, 2014. Principles and Practice of Declarative Programming (PPDP 2012).

[80] Cosimo Laneve, Michael Lienhardt, Ka I Pun, and Guillermo Román-Díez. Time analysis of actor programs. *Journal of Logical and Algebraic Methods in Programming*, 105:1 – 27, 2019.

[81] Daniel Leivant. Stratified functional programs and computational complexity. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 325–333, 1993.

[82] Daniel Leivant and Jean-Yves Marion. Lambda calculus characterizations of poly-time. In *Typed Lambda Calculi and Applications*, pages 274–288. Springer Berlin Heidelberg, 1993.

[83] Antoine Madet and Roberto M. Amadio. An elementary affine λ-calculus with multithreading and side effects. In *Typed Lambda Calculi and Applications - 10th International Conference, TLCA 2011*, volume 6690 of *Lecture Notes in Computer Science*, pages 138–152. Springer, 2011.

[84] Jean-Yves Marion. A type system for complexity flow analysis. In *Proceedings of the 26th Annual IEEE Symposium on Logic in Computer Science, LICS 2011*, pages 123–132. IEEE Computer Society, 2011.

[85] Damiano Mazza. Linear logic and polynomial time. *Mathematical Structures in Computer Science*, 16.

[86] John Mitchell, Mark Mitchell, and Andre Scedrov. A linguistic characterization of bounded oracle computation and probabilistic polynomial time. In *Foundations of Computer Science, 1998. Proceedings. 39th Annual Symposium on*, pages 725–733. IEEE, 1998.

[87] Van Chan Ngo, Quentin Carbonneaux, and Jan Hoffmann. Bounded expectations: Resource analysis for probabilistic programs. In *PLDI39*, pages 496–512, 2018.

[88] Benjamin Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. In *[1993] Proceedings Eighth Annual IEEE Symposium on Logic in Computer Science*, pages 376–385. IEEE, 1993.

[89] Milner Robin. A calculus of communicating systems. *Lect. Notes Comput. Sci.*, 1980.

[90] Davide Sangiorgi and David Walker. *The pi-calculus: a Theory of Mobile Processes*. Cambridge university press, 2003.

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

[92] Kazushige Terui. Light affine lambda calculus and polytime strong normalization. In *Logic in Computer Science, 2001. Proceedings. 16th Annual IEEE Symposium on*, pages 209–220. IEEE, 2001.

Appendix A

Additional Results

A.1 Type Inference Procedure for Work

We describe formally a type inference procedure for the work type system described in Section 4.2.3. The work described here has not been implemented yet, but in order to be sure it would be implementable and sound, we give a totally inductive presentation of the procedure. The notations might be cumbersome, as we have to track in particular names of variables. However, with this inductive presentation, we are convinced that the soundness and completeness result given at the end of this section is verified.

Notations for Type Inference

We start by introducing some notations for type inference. We have a particular focus on names and variable renaming, as it could be a source of problems in implementation. We want to design the procedure in such a way that it is easily implementable from the theoretical description, and such that the procedure is correct and complete.

From now on, we may use second order variables for indexes, denoted by $e_1, \ldots, e_n$. This is useful when doing type inference, in the same way that we usually need to add type variables, in a context with sized types we need second-order index variables. The difference between a usual index variable and a second-order index variable is that the second-order index variable comes with a *closure* (a function ν from second order index variables to index variables), that is to say the set of index variables that it can use. Moreover, we can do a substitution $e\{I/i\}$ on those variables. So, formally, we define extended indices on a set φ of first order variables and Ψ of second order variables (with closure ν) by:

$$I, J, K ::= i \in \varphi \mid f(I_1, \ldots, I_n) \mid e \in \Psi$$

We also ask that in the closure ν we have $\nu(e) \subset \varphi$ for all e (usually noted by abuse of notation $\nu \subset \varphi$).

Definition A.1.1 (Canonical Extension)**.** *We say that a canonical intermediate type T_c is a* canonical extension *of a simple type U if they have the same skeleton (that is to say T_c without sizes is U). We can generalize this to context.*

Definition A.1.2 (Constraints and Satisfaction)**.** *A constraint C on a set φ of first order variables and Ψ of second order variables (with closure $\nu \subset \varphi$) is an object of the form $I \leq J$ where I and J are extended indices on φ and Ψ, or an explicit substitution of an extended index. A set of such constraints is usually denoted $\mathcal{C}$.*

We say that ϵ is an instantiation *of Ψ (with a closure ν) if ϵ is a function defined on Ψ, such that $\epsilon(e)$ is an index with no second-order variables, and with first order variables in $\nu(e)$. Given an index I on φ and Ψ such that $\nu \subset \varphi$, we define I_ϵ as an index on φ with no second-order variables given by:*

- $i_\epsilon = i$
- $(f(I^1, \ldots, I^n))_\epsilon = f(I^1_\epsilon, \ldots, I^n_\epsilon)$
- $e_\epsilon = \epsilon(e)$
- $(e\{I/i\})_\epsilon = \epsilon(e)\{I/i\}$

(Note that for the last rule, there is a distinction to make between the syntactic substitution $e\{I/i\}$ and the semantically defined substitution $\epsilon(e)\{I/i\}$)

Finally, we say that a set of constraints $\mathcal{C}$ on φ and Ψ (with closure $\nu \subset \varphi$) is satisfiable if there exists an instantiation ϵ of Ψ such that, for all constraints $I \leq J$ in $\mathcal{C}$, we have:

$$\varphi; \cdot \vDash I_\epsilon \leq J_\epsilon$$

In order to harmonize notations for the following functions, let us give some usual notations we will reuse in the following.

Definition A.1.3 (Notations for index variables)**.** *In this section, we will denote an index variable as i^l_m. To denote the set of variables already used, we use an integer k and a function $\rho : \{1, \ldots, k\} \to \mathbb{N}$. This means that the variables we used are in the set $\{i^l_m \mid 1 \leq l \leq k \wedge 1 \leq m \leq \rho(l)\}$.*

Also, especially when doing type inference, we want to represent the usual φ in the type system. To do that, we use a set $\mathcal{K} \subset \{1, \ldots, k\}$. This denotes the set $\varphi \equiv \{i^l_m \mid l \in \mathcal{K} \wedge 1 \leq m \leq \rho(l)\}$.

Definition A.1.4 (Notation for second-order index variables)**.** *We will usually denote an index variable as e_i. To denote the set of variables already used, we use an integer N and a function $\nu : \{1, \ldots, N\} \to \mathcal{P}(\{1, \ldots, k\})$. This means that the already defined second-order index variables are $\{e_i \mid 1 \leq i \leq N\}$, and the variable e_i uses the index variables $\mathtt{var}(e_i) \equiv \{i^l_m \mid l \in \nu(i) \wedge 1 \leq m \leq \rho(l)\}$.*

Definition A.1.5 (Notation for output)**.** *Using the previous notations, we will usually in a procedure define new variables, thus updating the set of defined variables. For this, we use the notation k', ρ', $\mathcal{K}'$, φ', N', ν', with the expected meaning. When we need to use several times those notations, we will prefer using integers index k_0, k_1, etc.*

Intermediate Functions for Inference

Definition A.1.6 (Creating a Canonical Extension)**.** *We define the function*

$$\mathsf{canon}(T, k, \rho, N, \nu, l) \equiv (T'_c, k', \rho', N', \nu')$$

where T is a simple type and T'_c is a canonical extension of T. The additional integer $l \in \{1, \ldots, k\}$, denotes the current exponent for index variables we are using.

This is defined by induction on T. For simplicity reason, we do not recall the input arguments except T.

- $\mathsf{canon}(\mathsf{Nat}) \equiv \mathtt{let}\ m = \rho(l)\ \mathtt{in}\ (\mathsf{Nat}[0, i^l_{m+1}], k, \rho[l \mapsto m+1], N, \nu)$

- $\mathsf{canon}(\mathsf{List}(\mathcal{B})) \equiv \texttt{let } m = \rho(l) \texttt{ in let } (\mathcal{B}'_c, k', \rho', N', \nu') = \mathsf{canon}(\mathcal{B}, k, \rho[l \mapsto m+1], N, \nu, l) \texttt{ in } (\mathsf{List}[0, i^l_{m+1}](\mathcal{B}'_c), k', \rho', N', \nu')$
- $\mathsf{canon}(\alpha) \equiv (\alpha, k, \rho, N, \nu)$
- $\mathsf{canon}(A) \equiv (A, k, \rho, N, \nu)$
- $\mathsf{canon}(\mathit{ch}(\widetilde{T})) \equiv \texttt{let } (\widetilde{T'_c}, k', \rho', N', \nu') = \mathsf{canon}(\widetilde{T}, k{+}1, \rho[k{+}1 \mapsto 0], N{+}1, \nu[N{+}1 \mapsto \{k{+}1\}], k{+}1)$ $\texttt{in let } m' = \rho'(k+1) \texttt{ in } (\forall i_1^{k+1}, \ldots, i_{m'}^{k+1}.\mathit{ch}^{e_{N+1}}(\widetilde{T'_c}), k', \rho', N', \nu')$
- $\mathsf{canon}(\cdot) = (\cdot, k, \rho, N, \nu)$
- $\mathsf{canon}((U, \widetilde{T})) = \texttt{let } ((U_c), k_1, \rho_1, N_1, \nu_1) = \mathsf{canon}(U, k, \rho, N, \nu, l) \texttt{ in}$ $\texttt{let } (\widetilde{T'_c}, k_2, \rho_2, N_2, \nu_2) = \mathsf{canon}(\widetilde{T}, k_1, \rho_1, N_1, \nu_1, l) \texttt{ in}$ $((U'_c, \widetilde{T'_c}), k_2, \rho_2, N_2, \nu_2).$

Then, we can easily show by induction that the type returned is indeed a canonical type, and it verifies the constraints on the use of variables given by the notations.

We also need a unification procedure for base sized types. In order to do this, let us first define a generalization of types.

Definition A.1.7 (Quantified Type Form)**.** *A quantified type form has the shape $\forall\omega :: \mathcal{B}, T$ where $\mathcal{B}$ is a simple base type and T is an intermediate type that can use the special base type variable ω. Intuitively, this is used to type an expression e such that, for all intermediate type $\mathcal{B}'$ with a skeleton equal to $\mathcal{B}$, e has type $T[\omega ::= \mathcal{B}']$. Such a type $T[\omega ::= \mathcal{B}']$ is called a* particular case *of the type form.*

We say that the skeleton of $\forall\omega :: \mathcal{B}, T$ is the skeleton of T where the special variable ω is replaced by the skeleton of $\mathcal{B}$.

Definition A.1.8 (Unification Procedure)**.** *Given two base types (quantified type form) $\mathcal{B}_1$ and $\mathcal{B}_2$ with the same skeleton, we define the unification procedure*

$$\mathsf{unif}(\mathcal{B}_1, \mathcal{B}_2, k, \rho, \mathcal{K}, N, \nu) \equiv (\mathcal{B}', \mathcal{C}, N', \nu').$$

$\mathcal{B}'$ can be a quantified type form, and $\mathcal{C}$ is a set of constraints on the first-order variables described by k and ρ, and on the second order variables described by N and ν. The intuition is that if the constraints in $\mathcal{C}$ are satisfied by an instantiation ϵ, then there exists particular cases of the type forms $\mathcal{B}_1$ and $\mathcal{B}_2$, with $\varphi; \cdot \vdash (\mathcal{B}_i)_\epsilon \sqsubseteq \mathcal{B}'_\epsilon$ for $i = 1, 2$.

This is defined by induction on the shape of both $\mathcal{B}_1$ and $\mathcal{B}_2$. In practice, we take in input a base type with possibly a special variable ω_1 or ω_2 with its associated type, and it returns a base type with possibly a special variable ω.

- $\mathsf{unif}(\mathsf{Nat}[0, I], \mathsf{Nat}[0, J]) \equiv (\mathsf{Nat}[0, e_{N+1}], \{I \leq e_{N+1}; J \leq e_{N+1}\}, N+1, \nu[e_{N+1} \mapsto \mathcal{K}])$
- $\mathsf{unif}(\mathsf{List}[0, I](\mathcal{B}'_1), \mathsf{List}[0, J](\mathcal{B}'_2)) \equiv$ $\texttt{let } (\mathcal{B}', \mathcal{C}, N', \nu') = \mathsf{unif}(\mathcal{B}'_1, \mathcal{B}'_2, k, \rho, \mathcal{K}, N+1, \nu[e_{N+1} \mapsto \mathcal{K}]) \texttt{ in}$ $(\mathsf{List}[0, e_{N+1}](\mathcal{B}'), \mathcal{C}' \cup \{I \leq e_{N+1}; J \leq e_{N+1}\}, N', \nu')$
- $\mathsf{unif}(\alpha, \alpha) \equiv (\alpha, \emptyset, N, \nu)$
- $\mathsf{unif}(\omega_1 :: \mathcal{B}, \mathcal{B}_2) \equiv (\mathcal{B}_2[\omega_2 ::= \omega], \emptyset, N, \nu)$
- $\mathsf{unif}(\mathcal{B}_1, \omega_2 :: \mathcal{B}) \equiv (\mathcal{B}_1[\omega_1 ::= \omega], \emptyset, N, \nu)$
- Undefined otherwise.

With this, it is easy to show the following lemma.

Lemma A.1.1 (Correction of Unification)**.** *If* $\mathsf{unif}(\mathcal{B}_1, \mathcal{B}_2, k, \mathcal{K}, \rho, N, \nu) = (\mathcal{B}', \mathcal{C}, N', \nu')$ *then* $\mathcal{B}'$ *has the same skeleton as* $\mathcal{B}_1$ *and* $\mathcal{B}_2$*, and for any base type* $\mathcal{B}$*, there exists two base types* $\mathcal{B}'_1$ *and* $\mathcal{B}'_2$ *such that for any instantiation* ϵ *satisfying* $\mathcal{C}$*, we have*

$$\varphi; \cdot \vdash (\mathcal{B}_i(\mathcal{B}_i)')_\epsilon \sqsubseteq (\mathcal{B}'(\mathcal{B}))_\epsilon$$

Proof. By induction on the skeleton of both $\mathcal{B}_1$ and $\mathcal{B}_2$. All cases are direct. □

In the same way that we did for canon, we need a function that renames variables in an intermediate type.

Definition A.1.9 (Renaming a Canonical Type)**.** *We define the function*

$$\mathsf{rename}(T_c, k, \rho, N, \nu, l_0, l_1) \equiv (T'_c, \mathcal{C}, k', \rho', N', \nu')$$

where $\mathcal{C}$ *is a set of constraints. The integers* $l_0, l_1 \in \{1, \dots, k\}$*, denote the current exponent for index variables in* T_c *and in the renaming. We also ask that* $\rho(l_0) = \rho(l_1)$*.*

This is defined by induction on T_c*.*

- $\mathsf{rename}(\mathsf{Nat}[0, i_m^{l_0} + n]) \equiv (\mathsf{Nat}[0, i_m^{l_1} + n], \emptyset, k, \rho, N, \nu)$
- $\mathsf{rename}(\mathsf{List}[0, i_m^{l_0} + n](\mathcal{B}_c)) \equiv \texttt{let}\ (\mathcal{B}'_c, \mathcal{C}, k', \rho', N', \nu') = \mathsf{rename}(\mathcal{B}_c, k, \rho, N, \nu, l_0, l_1)\ \texttt{in}$ $(\mathsf{List}[0, i_m^{l_1} + n](\mathcal{B}'_c), \mathcal{C}, k', \rho', N', \nu')$
- $\mathsf{rename}(\alpha) \equiv (\alpha, \emptyset, k, \rho, N, \nu)$
- $\mathsf{rename}(A) \equiv (A, \emptyset, k, \rho, N, \nu)$
- $\mathsf{rename}(\forall i_1^l, \dots, i_m^l.\, \mathsf{ch}^K(\widetilde{T}_c)) \equiv \texttt{let}\ (\widetilde{T}'_c, \mathcal{C}, k', \rho', N', \nu') = \mathsf{rename}(\widetilde{T}_c, k+1, \rho[k+1 \mapsto m], N+1, \nu[N+1 \mapsto \{k+1\}], l, k+1)$ $\texttt{in}\ (\forall i_1^{k+1}, \dots, i_m^{k+1}.\, \mathsf{ch}^{e_{N+1}}(\widetilde{T}'_c), \mathcal{C} \cup \{e_{N+1} = K\{\widetilde{i}^{k+1}/\widetilde{i^l}\}\}, k', \rho', N', \nu')$
- $\mathsf{rename}(\cdot) = (\cdot, k, \rho)$
- $\mathsf{rename}((U_c, \widetilde{T}_c)) = \texttt{let}\ (U'_c, \mathcal{C}_1, k_1, \rho_1, N_1, \nu_1) = \mathsf{rename}(U_c, k, \rho, N, \nu, l_0, l_1)\ \texttt{in}$ $\texttt{let}\ (\widetilde{T}'_c, \mathcal{C}_2, k_2, \rho_2, N_2, \nu_2) = \mathsf{rename}(\widetilde{T}_c, k_1, \rho_1, N_1, \nu_1, l_0, l_1)\ \texttt{in}$ $((U'_c, \widetilde{T}'_c), \mathcal{C}_1 \cup \mathcal{C}_2, k_2, \rho_2, N_2, \nu_2)$.

To prove correctness of this definition, we will define a relation called *simply equal*, that is a notion of equality between types weaker than α-renaming equality, but easier to implement.

Definition A.1.10 (Simple Equality)**.** *We say that two canonical types* T_c *and* T'_c *are simply equal (denoted* $T_c =_s T'_c$*) if they are syntactically equal when we remove the exponent on the index variables. Formally, this is defined by the rules in Figure A.1. Note that by definition of canonicity (notably because of* btocc *and the ordering of variables), two canonical channel types with the same skeleton are simply equal if and only if all the complexity appearing in those types are simply equal.*

It is then easy to see that if two canonical channels types with no higher-order variables are simply equal then they are equal by α-renaming.

And then, we can prove the following lemma for correctness of renaming:

$$\frac{m_0 = m_1 \qquad n_0 = n_1}{\mathsf{Nat}[0, i^{l_0}_{m_0} + n_0] =_s \mathsf{Nat}[0, i^{l_1}_{m_1} + n_1]} \qquad \frac{m_0 = m_1 \qquad n_0 = n_1 \qquad (\mathcal{B}_c)_0 =_s (\mathcal{B}_c)_1}{\mathsf{List}[0, i^{l_0}_{m_0} + n_0]((\mathcal{B}_c)_0) =_s \mathsf{List}[0, i^{l_1}_{m_1} + n_1]((\mathcal{B}_c)_1)}$$

$$\frac{}{\alpha =_s \alpha} \qquad \frac{}{A =_s A} \qquad \frac{m_0 = m_1 \qquad K_0 =_s K_1 \qquad \widetilde{T_0} =_s \widetilde{T_1}}{\forall i^{l_0}_1, \ldots, i^{l_0}_{m_0}.\mathtt{ch}^{K_0}(\widetilde{T_0}) =_s \forall i^{l_1}_1, \ldots, i^{l_1}_{m_1}.\mathtt{ch}^{K_1}(\widetilde{T_1})}$$

$$\frac{m_0 = m_1}{i^{l_0}_{m_0} =_s i^{l_1}_{m_1}} \qquad \frac{\widetilde{I_0} =_s \widetilde{I_1}}{f(\widetilde{I_0}) =_s f(\widetilde{I_1})} \qquad \frac{}{e_i =_s e_i}$$

Figure A.1: Simple Equality

Lemma A.1.2 (Correctness of Renaming)**.** *If* $\mathsf{rename}(T_c, k, \rho, N, \nu, l_0, l_1) =_s (T'_c, \mathcal{C}, k', \rho', N', \nu')$ *and* ϵ *is an instantiation satisfying* $\mathcal{C}$*, then* $(T'_c)_\epsilon =_s (T_c)_\epsilon$*.*

The proof is done by induction.

Finally, let us say something about naming. We define first well-named types and contexts.

Definition A.1.11 (Well-named Types)**.** *Intuitively, we will say that a type is well-named given a set of variables if it uses all first-order variables at least once, and all the binding of first-order variables always use different variables. Intuitively, to show that a type is well-named, we extract the names in a type and show that this corresponds to our set of names. Formally, this is defined by the partial function:*

$$\mathsf{names}(T, k, \rho, \mathcal{K}, N, \nu) \equiv (\varphi'_f, \varphi'_b)$$

Also defined on extended indexes, with T *an intermediate type, and we define as usual* $\varphi \equiv \{i^l_m \mid l \in \mathcal{K} \mid 1 \leq m \leq \rho(l)\}$*. This set* φ *will correspond to the already bounded variables. Then,* $\varphi'_f \subset \varphi$ *is the set of free variables that we saw in the type (in the end, we would want* $\varphi = \varphi'_f$*), and* $\varphi'_b \subset \{i^l_m \mid 1 \leq l \leq k, 1 \leq m \leq \rho(l)\}$ *is the set of variables bound somewhere in the type, always satisfying* $\varphi \cap \varphi'_b = \emptyset$*. Note that we consider that first-order index variables are always denoted* i^l_m *for some integers* l *and* m *(as it will be the way to denote variables in the implementation). For the sake of simplicity, we define* $\varphi \coprod \varphi'$ *as the partial function returning* $\varphi \cup \varphi'$ *when* $\varphi \cap \varphi' = \emptyset$*, and is undefined otherwise. Then, we define* $(\varphi'_f, \varphi'_b) + (\varphi''_f, \varphi''_b) \equiv (\varphi'_f \cup \varphi''_f, \varphi'_b \coprod \varphi''_b)$*. We will also use* $\mathtt{assert}(b)$ *to say that the function is undefined if* b *is false.*

- $\mathsf{names}(i^l_m) \equiv \mathtt{assert}(i^l_m \in \varphi); (\{i^l_m\}, \emptyset)$
- $\mathsf{names}(f(I_1, \ldots, I_n)) \equiv \sum_{1 \leq i \leq n} \mathsf{names}(I_i, k, \rho, \mathcal{K}, N, \nu)$
- $\mathsf{names}(e_j) \equiv \mathtt{assert}(\nu(j) \subset \mathcal{K}); (\{i^l_m \mid l \in \nu(j) \wedge 1 \leq m \leq \rho(l)\}, \emptyset)$
- $\mathsf{names}(\mathsf{Nat}[0, I]) \equiv \mathsf{names}(I, k, \rho, N, \nu)$
- $\mathsf{names}(\mathsf{List}[0, I](\mathcal{B})) \equiv \mathsf{names}(I, k, \rho, N, \nu) + \mathsf{names}(\mathcal{B}, k, \rho, N, \nu)$
- $\mathsf{names}(\alpha) \equiv \mathsf{names}(A) \equiv (\emptyset, \emptyset)$
- $\mathsf{names}(\forall \widetilde{i}.\mathtt{ch}^K(\widetilde{T})) \equiv \mathtt{assert}(\widetilde{i} = i^l_1, \ldots, i^l_m \text{ for some } 1 \leq l \leq k, l \notin \mathcal{K} \text{ and } m = \rho(l));$
  $\mathtt{let}\ (\varphi_f, \varphi_b) = \mathsf{names}(K, k, \rho, \{l\}, N, \nu)\ \mathtt{in}\ \mathtt{assert}(\varphi_b = \emptyset)$
  $\mathtt{let}\ (\varphi'_f, \varphi'_b) = \mathsf{names}(\widetilde{T}, k, \rho, \mathcal{K} \cup \{l\}, N, \nu)\ \mathtt{in}\ \mathtt{assert}(\{i^l_1, \ldots, i^l_m\} \subset \varphi'_f);$
  $(\varphi'_f \cap \varphi, \varphi'_b \coprod \{i^l_1, \ldots, 1^l_m\})$
- $\mathsf{names}((T_1, \ldots, T_n)) \equiv \sum_{1 \leq i \leq n} \mathsf{names}(T_i, k, \rho, \mathcal{K}, N, \nu)$

And we obtain directly by induction that if $\mathsf{names}(T, k, \rho, \mathcal{K}, N, \nu) \equiv (\varphi'_f, \varphi'_b)$ *is defined, then we have indeed* $\varphi'_f \subset \varphi$ *and* $\varphi'_b \cap \varphi = \emptyset$ *with* $\varphi'_b \subset \{i^l_m \mid 1 \leq l \leq k, 1 \leq m \leq \rho(l)\}$

And then, we say that a type T (or a list of types) is well-named (under $(k, \rho, \mathcal{K}, N, \nu)$) when $\mathsf{names}(T, k, \rho, \mathcal{K}, N, \nu) \equiv (\varphi'_f, \varphi'_b)$ is defined, and we have $\varphi'_f = \varphi$ and $\varphi'_b \coprod \varphi = \{i^l_m \mid 1 \leq l \leq k, 1 \leq m \leq \rho(l)\}$

Now, we show that names are well-used in the functions defined previously.

Lemma A.1.3 (Naming in Intermediate Functions)**.** *We have,*

- *If* $\mathsf{canon}(T, k, \rho, N, \nu, l) \equiv (T'_c, k', \rho', N', \nu')$ *then* $\mathsf{names}(T'_c, k', \rho', \{l\}, N', \nu') \equiv (\varphi'_f, \varphi'_b)$ *is defined and* $\varphi'_f = \{i^l_m \mid \rho(l) < m \leq \rho'(l)\}$, $\varphi'_b = \{i^{l'}_m \mid k < l' \leq k', 1 \leq m \leq \rho'(l')\}$. *Moreover, for* $l' \neq l$, $1 \leq l' \leq k$, *we have* $\rho(l') = \rho'(l')$.
- *If* $\mathsf{rename}(T_c, k, \rho, N, \nu, l_0, l_1) \equiv (T'_c, \mathcal{C}, k', \rho', N', \nu')$, *and the following is defined:* $\mathsf{names}(T_c, k, \rho, \{l_0\}, N, \nu) \equiv (\varphi_f, \varphi_b)$, *then* $\mathsf{names}(T'_c, k', \rho', \{l_1\}, N', \nu') \equiv (\varphi'_f, \varphi'_b)$ *is defined and* $\varphi'_f = \{i^{l_1}_m \mid i^{l_0}_m \in \varphi_f\}$, $\varphi'_b = \{i^{l'}_m \mid k < l' \leq k', 1 \leq m \leq \rho'(l')\}$ *and is isomorphic to* φ_b. *Moreover, for* $1 \leq l \leq k$, *we have* $\rho(l) = \rho'(l)$.

The proof can be done by induction in both cases. Nothing is really difficult, we only need to expand the definitions.

In the same way of unify, we give a procedure to impose (simple) equality between two types. There are two different procedures, for canonical types and for non canonical types.

Definition A.1.12 (Imposing Canonical Equality)**.** *Given two canonical types* T^0_c *and* T^1_c, *we define the partial function:*

$$\mathsf{canoneq}(T^0_c, T^1_c, k, \rho, N, \nu, l_0, l_1) \equiv \mathcal{C}$$

where $1 \leq l_0 \leq k$, $1 \leq l_1 \leq k$ *are indices indicating the variables used in* T^0_c *and* T^1_c, *and we ask that* $\rho(l_0) = \rho(l_1)$. *The procedure is such that if it is defined and* ϵ *satisfies* $\mathcal{C}$, *then we have indeed* $(T^0_c)_\epsilon =_s (T^1_c)_\epsilon$.

- $\mathsf{canoneq}(\mathsf{Nat}[0, i^{l_0}_m + n], \mathsf{Nat}[0, i^{l_1}_m + n], k, \rho, N, \nu, l_0, l_1) \equiv \emptyset$
- $\mathsf{canoneq}(\mathsf{List}[0, i^{l_0}_m + n](\mathcal{B}^0_c), \mathsf{List}[0, i^{l_1}_m + n](\mathcal{B}^1_c), k, \rho, N, \nu, l_0, l_1) \equiv \mathsf{canoneq}(\mathcal{B}^0_c, \mathcal{B}^1_c, k, \rho, l_0, l_1)$
- $\mathsf{canoneq}(\alpha, \alpha, k, \rho, N, \nu, l_0, l_1) \equiv \emptyset$
- $\mathsf{canoneq}(A, A, k, \rho, N, \nu, l_0, l_1) \equiv \emptyset$
- $\mathsf{canoneq}(\forall i^{l'_0}_1, \ldots, i^{l'_0}_m.\, c\hbar^{K_0}(\widetilde{T_c}^0), \forall i^{l'_1}_1, \ldots, i^{l'_1}_m.\, c\hbar^{K_1}(\widetilde{T_c}^1), k, \rho, N, \nu, l_0, l_1) \equiv$
  $\{K_0 = K_1\{\widetilde{i^{l'_0}}/\widetilde{i^{l'_1}}\}\} \cup \bigcup_j \mathsf{canoneq}((T^0_c)_j, (T^1_c)_j, k, \rho, N, \nu, l'_0, l'_1)$

Then, for non-canonical types.

Definition A.1.13 (Imposing Equality)**.** *Given two types* T^0 *and* T^1, *we define the partial function:*

$$\mathsf{eq}(T^0, T^1, k, \rho, \mathcal{K}, N, \nu) \equiv \mathcal{C}$$

where $\mathcal{K}$ *indicates the variables used in* T^0 *and* T^1.

- $\mathsf{eq}(\mathsf{Nat}[0, I_0], \mathsf{Nat}[0, I_1], k, \rho, \mathcal{K}, N, \nu) \equiv \{I_0 = I_1\}$

$\dfrac{v : T \in \Gamma}{\Gamma \vdash v : T}$	$(\Delta(v), \emptyset, N, \nu)$
$\dfrac{}{\Gamma \vdash \underline{0} : \mathsf{Nat}}$	$(\mathsf{Nat}[0,0], \emptyset, N, \nu)$
$\dfrac{\Gamma \vdash e : \mathsf{Nat}}{\Gamma \vdash \mathtt{s}(e) : \mathsf{Nat}}$	$\mathtt{let}\ (\mathsf{Nat}[0,I], \mathcal{C}, N', \nu') = \mathsf{Inf}(\Gamma \vdash e : \mathsf{Nat}, k, \rho, \mathcal{K}, N, \nu, \Delta)\ \mathtt{in}$ $(\mathsf{Nat}[0, I+1], \mathcal{C}, N', \nu')$
$\dfrac{}{\Gamma \vdash [] : \mathsf{List}(\mathcal{B})}$	$(\forall \omega :: \mathcal{B}.\mathsf{List}[0,0](\omega), \emptyset, N, \nu)$
$\dfrac{\Gamma \vdash e : \mathcal{B} \qquad \Gamma \vdash e' : \mathsf{List}(\mathcal{B})}{\Gamma \vdash e :: e' : \mathsf{List}(\mathcal{B})}$	$\mathtt{let}\ (\mathcal{B}_1, \mathcal{C}_1, N_1, \nu_1) = \mathsf{Inf}(\Gamma \vdash e : \mathcal{B}, k, \rho, \mathcal{K}, N, \nu, \Delta)\ \mathtt{in}$ $\mathtt{let}\ (\mathsf{List}[0,I](\mathcal{B}_2), \mathcal{C}_2, N_2, \nu_2) = \mathsf{Inf}(\Gamma \vdash e' : \mathsf{List}(\mathcal{B}), k, \rho, \mathcal{K}, N_1, \nu_1, \Delta)\ \mathtt{in}$ $\mathtt{let}\ (\mathcal{B}', \mathcal{C}_3, N_3, \nu_3) = \mathsf{unif}(\mathcal{B}_1, \mathcal{B}_2, k, \rho, \mathcal{K}, N_2, \nu_2) =\ \mathtt{in}$ $(\mathsf{List}[0, I+1](\mathcal{B}'), \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_3, N_3, \nu_3)$

Figure A.2: Constraints Generation Rules for Expressions

- $\mathsf{eq}(\mathsf{List}[0,I_0](\mathcal{B}^0), \mathsf{List}[0,I_1](\mathcal{B}^1), k, \rho, \mathcal{K}, N, \nu) \equiv \{I_0 = I_1\} \cup \mathsf{eq}(\mathcal{B}^0, \mathcal{B}^1, k, \rho, \mathcal{K}, N, \nu)$
- $\mathsf{eq}(\alpha, \alpha, k, \rho, \mathcal{K}, N, \nu) \equiv \emptyset$
- $\mathsf{eq}(A, A, k, \rho, \mathcal{K}, N, \nu) \equiv \emptyset$
- $\mathsf{eq}(\forall i_1^{l_0'}, \ldots, i_m^{l_0'}.\mathcal{ch}^{K_0}(\widetilde{T_c}^0), \forall i_1^{l_1'}, \ldots, i_m^{l_1'}.\mathcal{ch}^{K_1}(\widetilde{T_c}^1), k, \rho, \mathcal{K}, N, \nu) \equiv$
  $\{K_0 = K_1\{\widetilde{i^{l_0'}}/\widetilde{i^{l_1'}}\}\} \cup \bigcup_j \mathsf{canoneq}((T_c^0)_j, (T_c^1)_j, k, \rho, N, \nu, l_0', l_1')$

Note that because of the last item, eq *is not defined on types with non-canonical subtypes for channels. This is not a problem since in type inference, all channel types will be canonical.*

It is then easy to see that for any instantiation ϵ satisfying $\mathcal{C}$, we have T_ϵ^0 equal to T_ϵ^1 by α-renaming.

Type Inference Procedure

We can now present the procedure associated to the intermediate type system. The goal is, from a process P and a classical π-calculus type derivation for this process (obtained by a standard type inference algorithm) to generate a set of constraints such that if this set has a solution, then we can construct a type derivation with sizes and complexity for the process P.

More precisely, we design two procedures, one for expressions and one for processes. Given a typing $\Gamma \vdash P$, and a context Δ that is a canonical extension of Γ, and that is well-named with respect to $(k, \rho, \mathcal{K}, N, \nu)$. We define:

$$\mathsf{Inf}(\Gamma \vdash P, k, \rho, \mathcal{K}, N, \nu, \Delta) \equiv (K, \mathcal{C}, k', \rho', N', \nu')$$

where K is an expression using index variables in φ' and possibly second-order index variables. The intuition is that, if we are given an instantiation ϵ satisfying $\mathcal{C}$, then, we have a typing $\varphi; \Delta_\epsilon \vdash P \triangleleft K_\epsilon$.

In the same way, we have an inference function for expressions

$$\mathsf{Inf}(\Gamma \vdash e : T, k, \rho, \mathcal{K}, N, \nu, \Delta) \equiv (T', \mathcal{C}, N', \nu')$$

Where T' is a type or a quantified type form $\forall \omega :: \mathcal{B}, U(\omega)$.

The procedure for expressions and processes are given in Figure A.2 and Figure A.3, where the notation for input are given above.

Then, we can show the following lemma on the procedure for expressions:

$\dfrac{}{\Gamma \vdash 0}$	$(0, \emptyset, k, \rho, N, \nu)$
$\dfrac{\Gamma \vdash P \qquad \Gamma \vdash Q}{\Gamma \vdash P \mid Q}$	$\texttt{let}\ (K_1, \mathcal{C}_1, k_1, \rho_1, N_1, \nu_1) =$ $\mathsf{Inf}(\Gamma \vdash P, k, \rho, \mathcal{K}, N+1, \nu[N+1 \mapsto \mathcal{K}], \Delta)\ \texttt{in}$ $\texttt{let}\ (K_2, \mathcal{C}_2, k_2, \rho_2, N_2, \nu_2) =$ $\mathsf{Inf}(\Gamma \vdash Q, k_1, \rho_1, \mathcal{K}, N_1, \nu_1, \Delta)\ \texttt{in}$ $(e_{N+1}, \mathcal{C}_1 \cup \mathcal{C}_2 \cup \{e_{N+1} \geq K_1 + K_2\}, k_2, \rho_2, N_2, \nu_2)$
$\dfrac{\Gamma \vdash a : \mathtt{ch}() \qquad \Gamma, \widetilde{v} : \widetilde{T} \vdash P}{\Gamma \vdash !a(\widetilde{v}).P}$	$\texttt{let}\ (\widetilde{\forall i}.\mathtt{ch}^K(\widetilde{T'_c}), \mathcal{C}_1, k_1, \rho_1, N_1, \nu_1) =$ $\mathsf{rename}(\Delta(a), k, \rho, N, \nu, \_, \_))\ \texttt{in}$ $\texttt{let}\ (K', \mathcal{C}_2, k_2, \rho_2, N_2, \nu_2) =$ $\mathsf{Inf}(\Gamma, \widetilde{v} : \widetilde{T} \vdash P, k_1, \rho_1, \mathcal{K} \cup \{k+1\}, N_1, \nu_1, (\Delta, \widetilde{v} :$ $\widetilde{T'_c}))\ \texttt{in}$ $(0, \mathcal{C}_1 \cup \mathcal{C}_2 \cup \{K' \leq K\}, k_2, \rho_2, N_2, \nu_2)$
$\dfrac{\Gamma \vdash a : \mathtt{ch}() \qquad \Gamma, \widetilde{v} : \widetilde{T} \vdash P}{\Gamma \vdash a(\widetilde{v}).P}$	Same as above.
$\dfrac{\Gamma \vdash a : \mathtt{ch}() \qquad \Gamma \vdash \widetilde{e} : \widetilde{T}}{\Gamma \vdash \overline{a}\langle\widetilde{e}\rangle}$	$\texttt{let}\ \forall i^l_1, \ldots, i^l_m.\mathtt{ch}^K(\widetilde{T_c}) = \Delta(a)\ \texttt{in}$ $\texttt{let}\ (\widetilde{T'}, \mathcal{C}, N', \nu') = \mathsf{Inf}_0(\Gamma \vdash \widetilde{e} : \widetilde{T}, k, \rho, \mathcal{K}, N+m+$ $1, \nu[N' \mapsto \mathcal{K} \mid N < N' \leq N+m+1], \Delta)\ \texttt{in}$ $\texttt{let}\ \mathcal{C}' = \mathsf{eq}(\widetilde{T_c}\{e_{N+1}, \ldots, e_{N+m}/i^l_1, \ldots, i^l_m\}, \widetilde{T'},$ $k, \rho, \mathcal{K}, N', \nu')\ \texttt{in}$ $(e_{N+m+1}, \mathcal{C} \cup \mathcal{C}' \cup \{e_{N+m+1} \geq$ $K\{e_{N+1}, \ldots, e_{N+m}/i^l_1, \ldots, i^l_m\}\}, k, \rho, N', \nu')$
$\dfrac{\Gamma, a : T \vdash P}{\Gamma \vdash (\nu a)P}$	$\texttt{let}\ (T_c, k', \rho', N', \nu') = \mathsf{canon}(T, k, \rho, N, \nu, \_)\ \texttt{in}$ $\mathsf{Inf}(\Gamma, a : T \vdash P, k', \rho', \mathcal{K}, N', \nu', (\Delta, a : T_c))$
$\dfrac{\Gamma \vdash v : \mathsf{Nat} \qquad \Gamma \vdash P \qquad \Gamma, x : \mathsf{Nat} \vdash Q}{\Gamma \vdash \mathtt{match}\ v\ \{\underline{0} \mapsto P; ;\ \mathtt{s}(x) \mapsto Q\}}$	$\texttt{let}\ \mathsf{Nat}[0, i+n] = \Delta(v)\ \texttt{in}$ $\texttt{let}\ (K_1, \mathcal{C}_1, k_1, \rho_1, N_1, \nu_1) =$ $\mathsf{Inf}(\Gamma \vdash P, k, \rho, \mathcal{K}, N+1, \nu[N+1 \mapsto \mathcal{K}], \Delta)\ \texttt{in}$ $\texttt{let}\ (\Delta', e') = \texttt{if}\ (n = 0)$ $\texttt{then}\ ((\Delta\{i/i+1\}, x : \mathsf{Nat}[0, i]), e_{N+1}\{I+1/i\}$ $\texttt{else}\ ((\Delta, x : \mathsf{Nat}[0, i + (n-1)]), e_{N+1})\ \texttt{in}$ $\texttt{let}\ (K_2, \mathcal{C}_2, k_2, \rho_2, N_2, \nu_2) =$ $\mathsf{Inf}(\Gamma, x : \mathsf{Nat} \vdash Q, k_1, \rho_1, \mathcal{K}, N_1, \nu_1, \Delta')\ \texttt{in}$ $(e_{N+1}, \mathcal{C}_1 \cup \mathcal{C}_2 \cup \{K_1 \leq e_{N+1}; K_2 \leq e'\}, k_2, \rho_2, N_2, \nu_2)$
$\dfrac{\Gamma \vdash v : \mathsf{List}(\mathcal{B}) \qquad \Gamma \vdash P \qquad \Gamma, x : \mathcal{B}, y : \mathsf{List}(\mathcal{B}) \vdash Q}{\Gamma \vdash \mathtt{match}\ v\ \{[] \mapsto P; ;\ x :: Q \mapsto\ \}}$	$\texttt{let}\ \mathsf{List}[0, i+n](\mathcal{B}') = \Delta(v)\ \texttt{in}$ $\texttt{let}\ (K_1, \mathcal{C}_1, k_1, \rho_1, N_1, \nu_1) =$ $\mathsf{Inf}(\Gamma \vdash P, k, \rho, \mathcal{K}, N+1, \nu[N+1 \mapsto \mathcal{K}], \Delta)\ \texttt{in}$ $\texttt{let}\ (\Delta', e') = \texttt{if}\ (n = 0)$ $\texttt{then}\ ((\Delta\{i/i+1\}, x{:}\mathcal{B}', y{:}\mathsf{List}[0, i](\mathcal{B}')), e_{N+1}\{i+1/i\})$ $\texttt{else}\ ((\Delta, x : \mathcal{B}', y : \mathsf{List}[0][i + (n-1)][\mathcal{B}']), e_{N+1})\ \texttt{in}$ $\texttt{let}\ (K_2, \mathcal{C}_2, k_2, \rho_2, N_2, \nu_2) =$ $\mathsf{Inf}(\Gamma, x : \mathcal{B}, y : \mathsf{List}(\mathcal{B}) \vdash Q, k_1, \rho_1, \mathcal{K}, N_1, \nu_1, \Delta')\ \texttt{in}$ $(e_{N+1}, \mathcal{C}_1 \cup \mathcal{C}_2 \cup \{K_1 \leq e_{N+1}; K_2 \leq e'\}, k_2, \rho_2, N_2, \nu_2)$
$\dfrac{\Gamma \vdash P}{\Gamma \vdash \mathtt{tick}.P}$	$\texttt{let}\ (K, \mathcal{C}, k', \rho', N', \nu') =$ $\mathsf{Inf}(\Gamma \vdash P, k, \rho, \mathcal{K}, N, \nu, \Delta)\ \texttt{in}$ $(K+1, \mathcal{C}, k', \rho', N', \nu')$

Figure A.3: Constraints Generation Rules for Processes

Lemma A.1.4 (Correction of Inference for Expressions)**.** *If* $\mathsf{Inf}(\Gamma \vdash e : T, k, \rho, \mathcal{K}, N, \nu, \Delta) = (T', \mathcal{C}, N', \nu')$*, then the skeleton of* T' *is* T *and for an instantiation* ϵ *satisfying* $\mathcal{C}$*, we have* $\varphi; \Delta_\epsilon \vdash e : T'_\epsilon$*. Moreover, if* $\mathsf{Inf}(\Gamma \vdash e : T, k, \rho, \mathcal{K}, N, \nu, \Delta) = (\forall \omega :: \mathcal{B}.T'(\omega), \mathcal{C}, N', \nu')$ *then the skeleton of* $T'(\mathcal{B})$ *is* T*, and for all intermediate type* $\mathcal{B}'$ *of skeleton* $\mathcal{B}$*, we have* $\varphi; \Delta_\epsilon \vdash e : T'(\mathcal{B}')_\epsilon$ *for all instantiation* ϵ *satisfying* $\mathcal{C}$*.*

Proof. The proof is done by induction on $\Gamma \vdash e : P$. All cases are direct except for list concatenation. The case for list concatenation is a consequence of Lemma A.1.1. □

Note that in particular, this procedure in the case of channel type is very easy, as channel types can only be typed with the axiom rule. So, in the following we may not use this procedure for channel type and for the sake of simplicity, directly rewrite the case of axiom when working with a channel. We also define simply $\mathsf{Inf}_0(\Gamma \vdash e : T, k, \rho, \mathcal{K}, N, \nu, \Delta)$ as the usual inference but instead of returning a type form $\forall \omega :: \mathcal{B}, T(\omega)$, we return the type $T(\mathcal{B}_0)$ where $\mathcal{B}_0$ is the intermediate type consisting in $\mathcal{B}$ with 0 everywhere for the sizes.

The first thing that we can easily verify with the previous results on names, is that we have indeed the invariant that Δ is a well-named canonical extension of Γ through the computation, as it is the main goal of the function rename and canon to keep this invariant. Moreover, we can easily see that $\rho'(l) = \rho(l)$ for all $1 \leq l \leq k$ and $\nu(n) = \nu'(n)$ for all $1 \leq n \leq N$ from Lemma A.1.3

And finally, we can show the following theorem.

Theorem A.1.1 (Soundness)**.** *If* $\mathsf{Inf}(\Gamma \vdash P, k, \rho, \mathcal{K}, N, \nu, \Delta) \equiv (K, \mathcal{C}, k', \rho', N', \nu')$ *then for any instantiation* ϵ *satisfying* $\mathcal{C}$*, we have* $\varphi; \Delta_\epsilon \vdash P \triangleleft K_\epsilon$ *with* $\varphi \equiv \{i^l_m \mid l \in \mathcal{K}, 1 \leq m \leq \rho(l) = \rho'(l)\}$*.*

Proof. By induction. We present here the important cases.

Case input. Let ϵ an instantiation satisfying $\mathcal{C}_1 \cup \mathcal{C}_2 \cup \{K' \leq K\}$. As $\Delta(a)$ is well-named, it has the shape $\forall i^l_1, \ldots, i^l_m.\mathtt{ch}^K(\widetilde{T_c})$. And, by definition, $\rho(l) = m$. Then, by definition of renaming, in $\forall \widetilde{i}.\mathtt{ch}^K(\widetilde{T})$ we have $\widetilde{i} = i^{k+1}_1, \ldots, i^{k+1}_m$.

By induction hypothesis, we obtain $\varphi_2; \Delta_\epsilon, \widetilde{v} : \widetilde{T}_\epsilon \vdash P \triangleleft K'_\epsilon$ with $\varphi_2 \equiv \{i^l_n \mid l \in \mathcal{K}, 1 \leq n \leq \rho(l) = \rho_1(l) = \rho_2(l)\} \cup \{i^{k+1}_n \mid 1 \leq n \leq m = \rho(l) = \rho_1(l) = \rho_2(l)\}$. By Lemma A.1.2, we have $\Delta(a)_\epsilon =_s (\forall \widetilde{i}.\mathtt{ch}^K(\widetilde{T}))_\epsilon$. Thus, by definition of simple equality, they are equal by α-renaming. We can thus consider Δ' equal to Δ_ϵ everywhere except for $\Delta'(a) \equiv (\forall \widetilde{i}.\mathtt{ch}^K(\widetilde{T}))_\epsilon$. And we obtain directly $\varphi_2; \Delta', \widetilde{v} : \widetilde{T}_\epsilon \vdash P \triangleleft K'_\epsilon$. And we have $K'_\epsilon \leq K_\epsilon$, thus we can do the usual input typing rule and conclude this case.

Case output. Let ϵ be an instantiation satisfying the set of constraints $\mathcal{C} \cup \mathcal{C}' \cup \{e_{N+m+1} \geq K\{e_{N+1}, \ldots, e_{N+m}/i^l_1, \ldots, i^l_m\}\}$. By Lemma A.1.4, we have $\varphi; \Delta_\epsilon \vdash \widetilde{e} : \widetilde{T}'_\epsilon$. Moreover, by definition of eq, we have $\widetilde{T}'_\epsilon$ equal to U_ϵ where $U \equiv \widetilde{T}_c\{e_{N+1}, \ldots, e_{N+m}/i^l_1, \ldots, i^l_m\}$ by α-renaming. Thus, $\epsilon(N+1), \ldots, \epsilon(N+m)$ gives us an instantiation for the output rule.

Case Pattern Matching. The case of pattern matching is direct by induction hypothesis. The only thing that needs some care is if we are going to use the rule for $n = 0$ or $n \neq 0$, but both cases are direct. □

Completeness: Intermediate Results

We now focus on completeness for the type inference procedure. First, remark that by a good use of variables and α-renaming, we can always consider that in a typing $\varphi; \Delta \vdash P \triangleleft K$, Δ is well-named. In order to state completeness, let us first give the following definition.

Definition A.1.14 (Ground Typing)**.** *Given a typing* $\varphi; \Delta \vdash P \triangleleft K$*, we define* $\lfloor \varphi; \Delta \vdash P \triangleleft K \rfloor$ *as the simple typing corresponding to the initial one without sizes.*

And now, what we want to show is the following theorem. Note that we do not have total completeness because of one detail. Indeed, when doing an output rule, we have to choose in the expressions $\widetilde{e}$ the sizes corresponding to the type of the channels. And sometimes, this choice can be totally arbitrary. For example, if $a : \forall i_1, i_2.\mathtt{ch}^K(\mathsf{List}[0, i_1](\mathsf{Nat}[0, i_2]))$, and we have the output $\overline{a}\langle[]\rangle$, then we need to choose a value for i_2, and it can be whatever we want. So, in order to impose some uniformity and have completeness, we ask that this choice is always 0. (When we can take whatever size we want, we consider that the size is 0 as it should lead to a smaller complexity). Such a typing is called a 0-typing.

Theorem A.1.2 (Completeness of Type Inference)**.** *Given a set of index variables described by k, ρ and $\mathcal{K}$ and a 0-typing (without second order variables) $\varphi; \Delta \vdash P \triangleleft K$ with Δ well-named, then $\mathsf{Inf}(\lfloor \varphi; \Delta \vdash P \triangleleft K \rfloor, k, \rho, \mathcal{K}, 0, \emptyset, \Delta) \equiv (K', \mathcal{C}, k', \rho', N', \nu')$ is defined and there exists an instantiation ϵ satisfying $\mathcal{C}$ such that K'_ϵ is equal to K.*

Let us first present some intermediate lemmas before going to the proof, related to the intermediate procedure.

Lemma A.1.5 (Completeness for Canonical Extension)**.** *Given variables decribed by (k, ρ, N, ν). Given a simple channel type T. Suppose also that*

$$\mathsf{canon}(T, k, \rho, N, \nu, \_) \equiv (T'_c, k', \rho', N', \nu')$$

Then, for any instantiation ϵ of (k, ρ, N, ν), for any other canonical channel type T_c with skeleton T, there exists an instantiation ζ of (k', ρ', N', ν') extension of ϵ such that $(T'_c)_\epsilon$ is equal to T_c.

Proof. This relies on several facts. The first one is to see that canon output indeed a canonical type, so the variables are ordered and there are exactly as many variables as there are occurrences of base types (as defined before). Then, we have to remark that two canonical channel types are equal (up-to α-renaming) if and only if their complexities are equal, because the shape is totally restricted for everything else. And then, by definition of canon, it suffices to take the instantiation ζ that gives to second-order variables the complexity in the type T_c. □

For the following lemma, we need the following definition for type form.

Definition A.1.15 (Type Form and Equality)**.** *By an abuse of notation, we say that a type form $\forall \omega :: \mathcal{B}.T(\omega)$ is equal to a type U is there exists a type T' of skeleton $\mathcal{B}$ such that $T(T')$ is equal to U.*

Lemma A.1.6 (Completeness for Unification)**.** *Given variables described by $(k, \rho, \mathcal{K}, N, \nu)$ and two base types (or quantified type form) $\mathcal{B}_1$ and $\mathcal{B}_2$ with the same skeleton, given also two intermediate base types $\mathcal{B}_1^r$ and $\mathcal{B}_2^r$ and an instantiation ϵ such that $(\mathcal{B}_i)_\epsilon$ is equal to $\mathcal{B}_i^r$, then,*

$$\mathsf{unif}(\mathcal{B}_1, \mathcal{B}_2, k, \rho, \mathcal{K}, N, \nu) \equiv (\mathcal{B}', \mathcal{C}, N', \nu')$$

is defined. And, for all intermediate type $\mathcal{B}_r$ such that $\varphi; \cdot \vdash \mathcal{B}_i^r \sqsubseteq \mathcal{B}_r$, we have an instantiation ζ extension of ϵ satisfying $\mathcal{C}$ and such that $\mathcal{B}'_\zeta$ is equal to $\mathcal{B}^r$.

Proof. By induction. The statement is a bit convoluted (notably because of type form and subtyping), but it works well in the proof. For example, suppose we are in the case

$$\mathsf{unif}(\mathsf{List}[0, I](\mathcal{B}_1), \mathsf{List}[0, J](\mathcal{B}_2), k, \rho, \mathcal{K}, N, \nu) \equiv (\mathsf{List}[0, e_{N+1}](\mathcal{B}'), \mathcal{C} \cup \{I \leq e_{N+1}; J \leq e_{N+1}\}, N', \nu')$$

with

$$\mathsf{unif}(\mathcal{B}_1, \mathcal{B}_2, k, \rho, \mathcal{K}, N + 1, \nu[e_{N+1} \mapsto \mathcal{K}]) \equiv (\mathcal{B}', \mathcal{C}, N', \nu')$$

(Recall the special variables ω_1 and ω_2 can appear in $\mathcal{B}_1$ or $\mathcal{B}_2$)

By hypothesis, we have two intermediate base types $\mathsf{List}[0, I^r](\mathcal{B}_1^r)$ and $\mathsf{List}[0, J^r](\mathcal{B}_2^r)$ and an instantiation ϵ. Let $\mathcal{B}_r$ ba a type such that $\varphi; \cdot \vdash \mathsf{List}[0, I^r](\mathcal{B}_1^r) \sqsubseteq \mathcal{B}_r$ and $\varphi; \cdot \vdash \mathsf{List}[0, J^r](\mathcal{B}_2^r) \sqsubseteq \mathcal{B}_r$. So, by definition, $\mathcal{B}_r \equiv \mathsf{List}[0, L](\mathcal{B}_r')$. Let $\epsilon' \equiv \epsilon[N+1 \mapsto L]$. By induction hypothesis, we have the variables described by $(k, \rho, \mathcal{K}, N+1, \nu[e_{N+1} \mapsto \mathcal{K}])$, two types $\mathcal{B}_1$ and $\mathcal{B}_2$ with the same skeleton. Two intermediate types $\mathcal{B}_1^r$ and $\mathcal{B}_2^r$, and instantiation ϵ' such that we have the equality (as ϵ' is an extension of ϵ). So, for the chosen intermediate type $\mathcal{B}_r'$ that is such that $\varphi; \cdot \vdash \mathcal{B}_1^r \sqsubseteq \mathcal{B}_r'$ and $\varphi; \cdot \vdash \mathcal{B}_2^r \sqsubseteq \mathcal{B}_r'$, we have an instantiation ζ extension of ϵ' satisfying $\mathcal{C}$ and such that $\mathcal{B}_\zeta'$ is equal to $\mathcal{B}_r'$. Thus, we obtain directly that ζ is a good instantiation for this case (as $\epsilon'(N+1) = L$ and L_ζ is greater than I_ζ and J_ζ).

□

Lemma A.1.7. *Given variables described by (k, ρ, N, ν). Given a canonical type T_c with free variables in $\{i_m^{l_0} \mid 1 \leq m \leq \rho(l_0)\}$ and an instantiation ϵ, we have,*

$$\mathsf{rename}(T_c, k, \rho, N, \nu, l_0, l_1) \equiv (T_c', \mathcal{C}, k', \rho', N', \nu')$$

defined and there exists an instantiation ζ extension of ϵ satisfying $\mathcal{C}$.

Proof. The proof is direct by induction on T_c. □

Then, we also have the following lemma.

Lemma A.1.8 (Completeness for Equality)**.** *Given variables described by $(k, \rho, \mathcal{K}, N, \nu)$, two types T^0 and T^1 and an instantiation ϵ such that T_ϵ^0 is equal to T_ϵ^1, then*

$$\mathsf{eq}(T^0, T^1, k, \rho, \mathcal{K}, N, \nu) \equiv \mathcal{C}$$

is defined and ϵ satisfies $\mathcal{C}$.

Proof. By induction. The only interesting case is the one for channel types, but it can be done easily by induction again, verifying that $\mathsf{canoneq}$ corresponds in fact to α-renaming in this case. □

And finally the most important lemma is the one for type inference for expressions.

Lemma A.1.9 (Completeness for Type Expressions)**.** *Given a set of first-order index variables described by k, ρ and $\mathcal{K}$ and second order index variables described by N and ν, given a typing (with no second order variables) $\varphi; \Delta \vdash e : T$ with Δ well-named, given also Δ' well-named (possibly using second-order variables) and an instantiation ϵ such that Δ_ϵ' is equal to Δ then*

$$\mathsf{Inf}(\lfloor \varphi; \Delta \vdash e : T \rfloor, k, \rho, \mathcal{K}, N, \nu, \Delta') \equiv (T', \mathcal{C}, N', \nu')$$

is defined and there exists an instantiation ζ extension of ϵ satisfying $\mathcal{C}$ such that T_ζ' (possibly a type form) is equal to T.

Proof. By induction. Almost all cases are direct except for list concatenation, that we detail here. Suppose that we have the typing:

$$\frac{\varphi; \Delta \vdash e : \mathcal{B}_r^1 \qquad \varphi; \Delta \vdash e' : \mathsf{List}[0, I_r](\mathcal{B}_r^2) \qquad \varphi; \cdot \vdash \mathcal{B}_r^1, \mathcal{B}_r^2 \sqsubseteq \mathcal{B}_r}{\varphi; \Delta \vdash e :: e' : \mathsf{List}[0, I_r + 1](\mathcal{B}_r)}$$

Let Δ' be a well-named context. Let ϵ be an instantiation of (k, ρ, N, ν) such that Δ'_ϵ is equal to Δ. By induction hypothesis, we have:

$$\mathsf{Inf}(\lfloor \varphi; \Delta \vdash e : \mathcal{B}^1_r \rfloor, k, \rho, \mathcal{K}, N, \nu, \Delta') \equiv (\mathcal{B}_1, \mathcal{C}_1, N_1, \nu_1)$$

is defined, and there exists an instantiation ζ_1 extension of ϵ satisfying $\mathcal{C}_1$ such that $(\mathcal{B}_1)_\zeta$ is equal to $\mathcal{B}^1_r$.

Thus, again by induction hypothesis,

$$\mathsf{Inf}(\lfloor \varphi; \Delta \vdash e' : \mathsf{List}[0, I](\mathcal{B}^2_r) \rfloor, k, \rho, \mathcal{K}, N_1, \nu_1, \Delta') \equiv (\mathsf{List}[0, I](\mathcal{B}_2), \mathcal{C}_2, N_2, \nu_2)$$

is defined and there exists an instantiation ζ_2 extension of ζ_1 satisfying $\mathcal{C}_2$ and such that (I_{ζ_2}) is equal to I_r and $(\mathcal{B}_2)_{\zeta_2}$ is equal to $\mathcal{B}^2_r$.

Moreover, we have that $\mathcal{B}_1$ and $\mathcal{B}_2$ have the same skeleton, and $(\mathcal{B}_1)_{\zeta_2}$ is equal to $\mathcal{B}^1_r$ and $(\mathcal{B}_2)_{\zeta_2}$ is equal to $\mathcal{B}^2_r$. So, by Lemma A.1.6,

$$\mathsf{unif}(\mathcal{B}_1, \mathcal{B}_2, k, \rho, \mathcal{K}, N_2, \nu_2) \equiv (\mathcal{B}', \mathcal{C}_3, N_3, \nu_3)$$

is defined. And, as we have $\varphi; \cdot \vdash \mathcal{B}^1_r, \mathcal{B}^2_r \sqsubseteq \mathcal{B}_r$, we also have an instantiation ζ extension of ζ_2 satisfying $\mathcal{C}_3$ and such that $\mathcal{B}'_\zeta$ is equal to $\mathcal{B}_r$.

So, ζ is indeed an extension of ϵ, satisfying $\mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_3$, and we have $(\mathsf{List}[0, I+1](\mathcal{B}'))_\zeta$ equal to $\mathsf{List}[0, I_r + 1](\mathcal{B}_r)$. This concludes this case. □

Proof of Theorem A.1.2

In order to show that, we need a more general property that can be used in induction hypothesis. Namely:

Lemma A.1.10 (Completeness of Type Inference)**.** *Given a set of first-order index variables described by k, ρ and $\mathcal{K}$ and second order index variables described by N and ν. Given a typing (with no second order variables) $\varphi; \Delta \vdash P \triangleleft K$ with Δ well-named, given also Δ' well-named (possibly using second-order variables) and an instantiation ϵ such that Δ'_ϵ is equal to Δ then*

$$\mathsf{Inf}(\lfloor \varphi; \Delta \vdash P \triangleleft K \rfloor, k, \rho, \mathcal{K}, N, \nu, \Delta') \equiv (K', \mathcal{C}, k', \rho', N', \nu')$$

is defined and there exists an instantiation ζ extension of ϵ satisfying $\mathcal{C}$ such that K'_ζ is equal to K. (Remark that this concept of extension makes sense only because both ρ,ρ' and ν,ν' agree on the initial set of variables)

Proof. We prove here Lemma A.1.10. By induction on the typing $\varphi; \Delta \vdash P \triangleleft K$.

- **Case** 0**.** Thus, $K = 0$. The inference procedure is defined, and it returns $(0, \emptyset, k, \rho, N, \nu)$. Thus, we can define $\zeta \equiv \epsilon$, it satisfies indeed $\emptyset$ and we have $0_\zeta = 0$.
- **Case Parallel Composition**. We have the following typing:

$$\frac{\varphi; \Delta \vdash P \triangleleft K_1 \qquad \varphi; \Delta \vdash Q \triangleleft K_2 \qquad \varphi; \cdot \vDash K_1 + K_2 \leq K}{\varphi; \Delta \vdash P \mid Q \triangleleft K}$$

  Let us define $\epsilon' \equiv \epsilon[N+1 \mapsto K]$. We have directly that ϵ' is an instantiation for $(k, \rho, N+1, \nu[N+1 \mapsto \mathcal{K}]$, and $\Delta'_{\epsilon'}$ is Δ. Thus, by induction hypothesis, we know that

$$\mathsf{Inf}(\lfloor \varphi; \Delta \vdash P \triangleleft K_1 \rfloor, k, \rho, \mathcal{K}, N+1, \nu[N+1 \mapsto \mathcal{K}], \Delta') \equiv (K'_1, \mathcal{C}_1, k_1, \rho_1, N_1, \nu_1)$$

is defined, and we have an instantiation ζ_1 extension of ϵ' satisfying $\mathcal{C}_1$ and such that $(K'_1)_{\zeta_1}$ is equal to K_1.

Thus, ζ_1 is by definition an instantiation for $(k_1, \rho_1, N_1, \nu_1)$, and we have Δ'_{ζ_1} equal to Δ as it is an extension of ϵ'. So, by induction hypothesis,

$$\mathsf{Inf}(\lfloor \varphi; \Delta \vdash Q \triangleleft K_2 \rfloor, k_1, \rho_1, \mathcal{K}, N_1, \nu_1, \Delta') \equiv (K'_2, \mathcal{C}_2, k_2, \rho_2, N_2, \nu_2)$$

is defined, and we have an instantiation ζ_2 extension of ζ_1 satisfying $\mathcal{C}_2$ and such that $(K'_2)_{\zeta_2}$ is equal to K_2.

Thus, ζ_2 is by definition an instantiation for $(k_2, \rho_2, N_2, \nu_2)$, ζ_2 satisfies $\mathcal{C}_1$ as it is an extension of ζ_1, it satisfies also $\mathcal{C}_2$ and it also satisfies $e_{N+1} \geq K'_1 + K'_2$ as we have (since ζ_2 is an extension of ϵ'):

$$(e_{N+1})_{\zeta_2} \equiv K \qquad (K'_1 + K'_2)_{\zeta_2} \equiv K_1 + K_2 \qquad \varphi; \cdot \vDash K \geq K_1 + K_2$$

This concludes this case.

- **Case Input.** Suppose we have the following typing:

$$\frac{\varphi; \Delta \vdash a : \forall \widetilde{j}.\mathtt{ch}^{K_0}(\widetilde{T_c}) \qquad \varphi, \widetilde{j}; \Delta, \widetilde{v} : \widetilde{T_c} \vdash P \triangleleft K_1 \qquad (\varphi, \widetilde{j}); \cdot \vDash K_1 \leq K_0}{\varphi; \Delta \vdash a(\widetilde{v}).P \triangleleft 0}$$

By definition of canonical type, $\Delta'(a)$ has no free variables. By Lemma A.1.7,

$$\mathsf{rename}(\Delta'(a), k, \rho, N, \nu, \_, \_) \equiv (\forall \widetilde{i}.\mathtt{ch}^K(\widetilde{T_c}'), \mathcal{C}_1, k_1, \rho_1, N_1, \nu_1)$$

is defined and there exists an instantiation ζ_1 extension of ϵ satisfying $\mathcal{C}_1$.

Moreover, by Lemma A.1.2, $\Delta'_{\zeta_1}(a)$ and $(\forall \widetilde{i}.\mathtt{ch}^K(\widetilde{T_c}'))_{\zeta_1}$ are equal by α-renaming. Thus, $(\Delta', \widetilde{v} : \widetilde{T_c}')_{\zeta_1}$ is equal to Δ by α-renaming. In particular, we have K_{ζ_1} equals to $K_0\{\widetilde{i^{k+1}}/\widetilde{j}\}$ So, by induction hypothesis,

$$\mathsf{Inf}(\lfloor \varphi, \widetilde{i}; \Delta, \widetilde{v} : \widetilde{T_c} \vdash P \triangleleft K_1 \rfloor, k_1, \rho_1, \mathcal{K} \cup \{k+1\}, N_1, \nu_1, (\Delta', \widetilde{v} : \widetilde{T_c}') \equiv (K', \mathcal{C}_2, k_2, \rho_2, N_2, \nu_2)$$

is defined, and there exists an instantiation ζ_2 extension of ζ_1 satisfying $\mathcal{C}_2$ such that K'_{ζ_2} is equal to $K_1\{\widetilde{i^{k+1}}/\widetilde{j}\}$. Moreover, recall that $K'_{\zeta_2} \leq K_{\zeta_2}$ because $(\varphi, \widetilde{i}); \cdot \vDash K_1 \leq K_0$. Thus, this concludes this case.

- **Case Output.** Suppose we have the following typing:

$$\frac{\varphi; \Delta \vdash a : \forall \widetilde{j}.\mathtt{ch}^{K_0}(\widetilde{T_c}) \qquad \varphi; \Delta \vdash \widetilde{e} : \widetilde{T_c}\{\widetilde{J}/\widetilde{j}\} \qquad \varphi; \cdot \vDash K_0\{\widetilde{J}/\widetilde{j}\} \leq K_1}{\varphi; \Delta \vdash \overline{a}\langle \widetilde{e} \rangle \triangleleft K_1}$$

As Δ' is well named, $\Delta'(a)$ has the shape $\forall i^l_1, \ldots, i^l_m.\mathtt{ch}^K(\widetilde{T_c}')$. Moreover, Δ'_ϵ is equal to Δ, so we obtain directly:

$$K_0\{\widetilde{i^l}/\widetilde{j}\} = K_\epsilon \qquad \widetilde{T_c}\{\widetilde{i^l}/\widetilde{j}\} = \widetilde{T_{c_\epsilon}}'$$

Let us define ϵ' extension of ϵ with, for all $1 \leq i \leq m$, $\epsilon'(N+i) \equiv J_i$ and $\epsilon'(N+m+1) \equiv K_1$. Then, ϵ' is an instantiation for $(k, \rho, N+m+1, \nu[N' \mapsto \mathcal{K} \mid N < N' \leq N+m+1])$. So, by Lemma A.1.9,

$$\mathsf{Inf}_0(\lfloor \varphi; \Delta \vdash \widetilde{e} : \widetilde{T_c}\{\widetilde{J}/\widetilde{j}\} \rfloor, k, \rho, \mathcal{K}, N+m+1, \nu[N' \mapsto \mathcal{K} \mid N < N' \leq N+m+1], \Delta) \equiv (\widetilde{T'}, \mathcal{C}, N', \nu')$$

is defined and there exists an instantiation ζ extension of ϵ' satisfying $\mathcal{C}$ and such that $\widetilde{T}'_\zeta$ is equal to $\widetilde{T}_c\{\widetilde{J}/\widetilde{j}\}$. Indeed, we can easily generalize Lemma A.1.9 to lists of expressions, and as we have a 0-typing, we know that replacing the special variable ω by a 0-type give exactly the initial type.

By the previous equality between $\widetilde{T}'_\zeta$ and $\widetilde{T}_c\{\widetilde{J}/\widetilde{j}\}$, and the fact that $\widetilde{T}_c\{\widetilde{i^l}/\widetilde{j}\} = \widetilde{T}_{c\epsilon}{}'$, we obtain that $(\widetilde{T}_c{}'\{e_{N+1}, \ldots, e_{N+m}/i^l_1, \ldots, i^l_m\})_\zeta$ is equal to $\widetilde{T}'_\zeta$. So, by Lemma A.1.8,

$$\mathsf{eq}(\widetilde{T}_c\{e_{N+1}, \ldots, e_{N+m}/i^l_1, \ldots, i^l_m\}, \widetilde{T}', k, \rho, \mathcal{K}, N', \nu') \equiv \mathcal{C}'$$

is defined and ζ satisfies $\mathcal{C}'$. Moreover, as previously, ζ also satisfies the constraint $\{e_{N+m+1} \geq K\{e_{N+1}, \ldots, e_{N+m}/i^l_1, \ldots, i^l_m\}\}$, so we can conclude this case.

- **Case ν.** This case is rather direct using induction hypothesis on the instantiation given by Lemma A.1.5.
- **Case `tick`.** This case is direct by induction hypothesis.
- **Case Pattern Matching.** Again, this case is direct by induction hypothesis and looks a lot like the case for parallel composition. The only thing that needs to be done is separate in two cases depending on whether the initial typing was with a $n = 0$ or not.

□

So, we have indeed a sound and complete procedure for type inference for intermediate types, we have reduced the problem of type inference to solving a set of constraints.