Concurrency, references and linear logic

Yann Hamdaoui

Université Sorbonne Paris Cité

université PARIS DIDEROT PARIS 7

THÈSE DE DOCTORAT DE
L'UNIVERSITÉ SORBONNE PARIS CITÉ

Préparée à l'Université Paris Diderot

École Doctorale 386 – Sciences Mathématiques Paris-Centre

Institut de Recherche en Informatique / Équipe PPS

Concurrency, References and Linear Logic

Préparée et présentée par **Yann HAMDAOUI**

Thèse de doctorat d'Informatique

Dirigée par Claudia Faggian

Présentée et soutenue publiquement à l'IRIF le 25 septembre 2018

Président du Jury		
Laurent Regnier	Professeur	Univ. Aix-Marseille
Rapporteurs		
Lorenzo Tortora de Falco	Professeur	Univ. Roma 3 (Italie)
Ian Mackie	Professeur (Reader)	Univ. of Sussex (UK)
Examinateurs		
Christine Tasson	Maitre de conférence	Univ. Paris Diderot
M. Thomas Ehrhard	Directeur de recherche	CNRS – Univ. Paris Diderot
Daniele Varacca	Professeur	Univ. Paris Est Créteil
Directrice de thèse		
Claudia Faggian	Chargée de recherche	Univ. Paris Diderot
Co-directeur		
M. Benoit Valiron	Maitre de conférence	Univ. Paris-Saclay

Titre: Concurrence, Références et Logique Linéaire

Résumé

Le sujet de cette thèse est l'étude de l'encodage des références et de la concurrence dans la Logique Linéaire. Notre motivation est de montrer que la Logique Linéaire est capable d'encoder des effets de bords, et pourrait ainsi servir comme une cible de compilation pour des langages fonctionnels qui soit à la fois viable, formalisée et largement étudiée. La notion clé développée dans cette thèse est celle de zone de routage. C'est une famille de réseaux de preuve qui correspond à un fragment de la logique linéaire différentielle, et permet d'implémenter différentes primitives de communication. Nous les définissons et étudions leur théorie. Nous illustrons ensuite leur expressivité en traduisant un λ-calcul avec concurrence, références et réplication dans un fragment des réseaux différentiels. Pour ce faire, nous introduisons un langage semblable au λ-calcul concurrent d'Amadio, mais avec des substitutions explicites à la fois pour les variables et pour les références. Nous munissons ce langage d'un système de types et d'effets, et prouvons la normalisation forte des termes bien typés avec une technique qui combine la réductibilité et une nouvelle technique interactive. Ce langage nous permet de prouver un théorème de simulation, et un théorème d'adéquation pour la traduction proposée.

Mots-clés: logique linéaire, sémantique, concurrence, références, parallélisme, réseaux de preuve, programmation fonctionnelle

Title: Concurrency, References and Linear Logic

Abstract

The topic of this thesis is the study of the encoding of references and concurrency in Linear Logic. Our perspective is to demonstrate the capability of Linear Logic to encode side-effects to make it a viable, formalized and well studied compilation target for functional languages in the future. The key notion we develop is that of routing areas: a family of proof nets which correspond to a fragment of differential linear logic and which implements communication primitives. We develop routing areas as a parametrizable device and study their theory. We then illustrate their expressivity by translating a concurrent λ-calculus featuring concurrency, references and replication to a fragment of differential nets. To this purpose, we introduce a language akin to Amadio's concurrent λ-calculus, but with explicit substitutions for both variables and references. We endow this language with a type and effect system and we prove termination of well-typed terms by a mix of reducibility and a new interactive technique. This intermediate language allows us to prove a simulation and an adequacy theorem for the translation.

Keywords: linear logic, semantics, concurrency, references, parallelism, proof nets, functional programming

À mon grand père.

Acknowledgements

Je remercie en premier lieu mes parents, sans qui je n'aurais pas pu être là aujourd'hui, ni même les jours précédents d'ailleurs. Ils m'ont soutenu psychologiquement et matériellement plus que je ne pourrais probablement jamais leur rendre. Pour cela, je leur suis infiniment reconnaissant. Je remercie également mes soeurs, Anna et Neijma, dont l'amour et l'humour caustique (juste l'humour) ont été un soutient constant.

Je suis profondément reconnaissant à Claudia d'avoir pris en thèse il y a 4 ans maintenant cet étudiant ignare et dans la lune que j'étais et de l'avoir supporté jusque là. Je remercie Benoît de l'avoir rejointe un peu plus tard. J'ai toujours pu compter sur votre suivi et votre soutient, même dans les moments difficiles. C'est probablement une formule convenue, mais qui n'en reste pas moins valide: je n'en serais pas là aujourd'hui sans vous.

Je remercie Lorenzo Tortora de Falco et Ian Mackie, d'avoir accepté d'être les rapporteurs de ma thèse, et ce malgré des contraintes matérielles et temporelles difficiles.

Je remercie les chercheurs qui m'ont acceuilli de près ou de loin, en Angleterre, en Italie ou au Japon au cours de cette thèse: Dan, Ugo, Ichiro et Kazushige notamment.

PPS aura été un cadre de travail exceptionnel et stimulant: je connais peu de lieux concentrant autant de gens brillants et passionnés. Je remercie également tous ceux avec qui j'ai pu avoir des discussions éclairantes, ou simplement de qui j'ai beaucoup appris, Beniamino, Paolo, Michele, Paul-André, Thomas, Ugo, Ichiro, et bien d'autres. Je remercie particulièrement Alexis, psychotérapeute attitré des thésards, dont l'écoute et les conseils dans des périodes difficiles ont été d'un grand secours.

Je remercie également Odile, qui outre son efficacité irréelle qui a longtemps fait peser sur elle des soupçons de dopage, est également l'une des personnes plus adorables que je connaisse.

Je suis fier d'avoir appartenu à l'institution millénaire du bureau des autistes, établissement médical rassemblant divers transfuges linéaires et catégoriques. Clément, chef à ses heures perdues et qui m'aura accompagné dans nos mésaventures thésardesques variées. Charles, partenaire de dégustation de vins et bières de prédilection et grand amateur de pipe. Marie, consoeur bretonne. Amina, partenaire de double master de qualité, merci pour tes cours de running et de danse lors de cet ETAPS mythique à Londres. Un jour nous serons traders

ou gérerons notre narco-empire en Colombie, comme promis, Patròn. Gabriel, encyclopédie vivante d'oeuvres de science-fiction et de jeux vidéos, merci pour tes lumières sur OCaml et la programmation en général. Pierre, grimpeur de son état, merci pour tes calembours intempestifs et surtout d'avoir téléchargé l'intégralité de ncatlab dans ton cortex, ce qui s'est révélé très pratique pour tes cobureaux. Léo, merci pour rien. Entre les pauses thés, la dénigration systématique des Danois, de LaTeX, de la blockchain et du monde en général, j'apprécie bien trop ta compagnie pour avoir eu la moindre minute de travail à moi. Rémi, merci pour ton enthousiasme débordant et ta bienveillance, et pour l'exegèse collaborative de l'oeuvre de Girard. J'espère sincèremet que tu parviendra un jour à surmonter ton addiction à l'installateur Debian. Paulina, je te souhaite bien du courage pour la suite avec de tels énergumènes.

Dans l'étrange contrée de thésardie, je remercie le peuple autochtone qui a été aujourd'hui (presque) remplacé par une nouvelle génération: Etienne, PM et notre goût partagé pour les bons thés et les détournements scabreux, Gabriel S., Matthieu B., Pierre et Cyril pour leurs culture impressionante et dans des domaines parfois déroutants, Raphaëlle, Hadrien, Thibaut, et probablement une foultitude d'autres que ma mémoire défaillante n'est pas capable de se rappeler dans l'instant.

Je remercie aussi la nouvelle génération dont la compagnie aura été tout autant agréable: l'équipe gâteau, Victor et Cédric dirigés par le Général de Brigade Zeinab, pour votre gestion efficace de cette cérémonie et votre créativité sans limite dans les annoncements. Merci à Théo, Antoine, Léonard, Nicolas, Tommaso, Jules, Axel et les autres.

Vient le tour de mes autres amis, que je coremercie plutôt que remercie, au sens où j'aurais probablement pu finir ma thèse en 6 mois s'ils n'avaient pas été là. Mais ces 6 mois auraient été bien ternes et ennuyeux.

Les irréductibles de l'Essonne pour commencer, Geoffroy, Thib, Fabze, Jerem, Adeline et Rémy, dont certains m'ont supporté depuis si longtemps que j'ai arrêté de compter. Je pense que j'ai tellement de remerciements à vous faire qu'il me faudrait probablement écrire une seconde thèse pour qu'ils puissent y tenir. Ne changez rien.

Merci à Marianne, de m'avoir soutenu et encouragé, et de toujours avoir prêté une oreille à mes problèmes. Merci à Julie, de m'avoir supporté, voire porté parfois, et d'avoir été aussi présente pour moi. Vous avez été indispensables.

Merci à Quentin, que j'ai traîné dans les bas-fonds parisiens, et qui a toujours représenté avec dignité le clan familial.

Les GGs ensuite, pour avoir partagé l'antre du bonheur pendant presque

un an. C'est dans cette ambiance d'émulation intellectuelle intense que j'ai établi ma toute première équation mathématique : "deux fois plus de cours = deux fois plus de temps libre". Gerblé, mon GG attitré et compère de toujours, homme-cheval-poisson, avec qui j'ai traversé des déserts de servitude et des champs de soleil. T,tjb. No12, co-anniversereux et déclaré meilleur rampeur d'anniversaires 2016, dit le Jawad de Strasbourg-Saint-Denis et adoré de tous ses voisins, merci de m'avoir accueilli comme je suis d'innombrable fois et pour ces diverses projections cinématographiques de qualité. Tu es une source d'inspiration inépuisable pour l'ensemble d'entre nous. Pie XII, vénérable catholique et milliardaire, grand gardien de la feuille de présence, tu m'as appris qu'il ne faut jamais cesser de se remettre en question dans la vie, et en particulier se demander: qui suis-je ? (rires). Jules, the defusable man, jamais aussi splendide qu'échoué en peignoir sur le canapé, merci d'avoir écouté mes élucubrations mathématiques et informatiques avec grand intérêt.

Puis l'équipée de Bois-Colombes et la nébuleuse qui gravite autour. La Meum's, depuis cette aventure du Dekmantel au tout début de ma thèse, on ne s'est plus vraiment quittés. Grand maître des afters squares et du n'importe quoi en général, srab d'escalade et soudé dans l'adversité capillaire, merci pour tout. Poupours, douce comme un ageneau (sic), Agatha-chan rockstar et artiste martial à ses heures perdues, la Mache, le meilleur protostagiaire que j'ai eu l'occasion de former, la Valj ou la nonchalance, Goldyboy le champion de course de brouettes, la Franche, Amiral de la Narine et Capitaine de chaloupe inégalé dans toute la mer d'Iroise, Eliott et sa fine connaissance du rap et des belles tapisseries, la couls, la Glache, Coralie, Sarah, Bap, la Basse, la Masse, et tous les autres que j'ai pu oublier.

Merci aux Michels, Lulu d'am a.k.a Lucifer, petite Margot, Léa mon sujet préféré de toute la Rabzouzie, Tangouze no limit, Fedi la plus belle des princesses Niglo, Fourmarie, femme de goût aussi bien vestimentairement que gastronomiquement parlant, Beber, Marie, Hélo, Blondie car qu'est-ce qu'un Picon sans Ducon, Tinmar, Felix, Thibax, et tous les autres pour avoir égayé ces années de vos picons chouffe. Merci d'être vous.

Merci aux faisands et assimilés, nouveaux ou anciens, Jonath, Jess, Jerem & Jerem, Nico, Hélène, Tom, Marion, Jim, Romain, Ben, Baddri, Eliot, Mike, Chloé, Léo, Paula, Adriana, Morgane, Jaimito, Clara, Charlie et tous les autres. Merci pour votre bonne humeur et vos talens musicaux. Je suis toujours ressorti ressourcé de votre belle maison.

Merci aux Centraliens, Doc d'amoume, Baza l'affreux, Foox le fooxien, La Croute, Sancho, Manus, Denver, Passpass, Gob, Billou, Babass, et tous les autres que je n'ai pas la place de citer, pour ces repas odieux au japonais.

Enfin, merci aux Originaux, Anti, Adri, Mariche, Eugé, Ben, Louis et toutes les autres belles personnes qui gravitent autour de vous pour avoir rendue un peu plus douce ma période de rédaction et de préparation de soutenance. Paradoxalement, le FOMO qui a découlé de l'Origine s'est transformée en un puissant moteur pour la rédaction de mon manuscrit.

Contents

1 Introduction **12**
- 1.1 Goals and motivations . . . 16
- 1.2 Context . . . 17
- 1.3 Related works . . . 20
- 1.4 Contributions . . . 21
- 1.5 Plan . . . 23

2 Nets **26**
- 2.1 Syntax . . . 27
- 2.2 Reduction . . . 31
  - 2.2.1 Base reduction $\rightarrow$ . . . 31
  - 2.2.2 Lifting reduction to nets . . . 34
  - 2.2.3 About closed and surface reduction . . . 35
- 2.3 Confluence . . . 35
- 2.4 Normalizing and non-normalizing nets . . . 37
- 2.5 Summary . . . 44
- 2.6 Discussion . . . 44

3 Routing Areas **46**
- 3.1 Routing Areas . . . 47
  - 3.1.1 Multirelations . . . 48
  - 3.1.2 (Co)contraction trees . . . 50
  - 3.1.3 Correctness criterion . . . 51
  - 3.1.4 Routing areas . . . 52
- 3.2 Operations on Routing Areas . . . 54
- 3.3 Routing Nets . . . 58
  - 3.3.1 Paths in Routing Nets . . . 58
  - 3.3.2 Normal forms . . . 61
  - 3.3.3 The Routing Semantics . . . 63
- 3.4 Summary . . . 64
- 3.5 Discussion . . . 64

4 The concurrent λ-calculus with explicit substitutions $\lambda_{\mathbf{cES}}$ 67
4.1 A Concurrent λ-calculus with Explicit Substitutions 70
4.1.1 Syntax 70
4.1.2 Reduction 72
4.1.3 Weak confluence 77
4.1.4 Preorder on terms 82
4.2 Stratification and Type System 86
4.2.1 The Type System of λ_{cES} 86
4.2.2 Subject reduction 89
4.2.3 Progress 96
4.3 Termination 97
4.3.1 Technical Definitions 99
4.3.2 Strong Normalization for λ_{cES} 101
4.3.3 Proof of Proposition 4.3.9 104
4.4 λ_{C} and λ_{cES} 109
4.4.1 The concurrent λ-calculus λ_{C} 110
4.4.2 Simulation 112
4.4.3 Adequacy 118
4.5 Summary 120
4.6 Discussion 120

5 Encoding a concurrent λ-calculus in nets 122
5.1 The Translation 123
5.1.1 Translating types and effects 123
5.1.2 Combining effects 124
5.1.3 Translating terms 125
5.2 Simulation 129
5.2.1 Nets contexts 129
5.2.2 Variable substitutions reductions 131
5.2.3 Downward reference substitutions reductions 133
5.2.4 Upward reference substitutions reduction 135
5.3 Termination and Adequacy 136
5.3.1 Termination 136
5.3.2 Proof of Proposition 5.3.1 137
5.3.3 Adequacy 141
5.4 Summary 142
5.5 Discussion 143
5.6 Perspective 144

Bibliography **153**

Chapter 1

Introduction

> Computer science is no more about computers than astronomy is about telescopes, biology is about microscopes or chemistry is about beakers and test tubes.
>
> Michael R. Fellow, Ian Parberry

Computations have existed long before computers and computer science. Babylonians had, for example, precise descriptions of algorithms to compute the solutions to various geometric problems more than one millennium before the beginning of formal algebra. What is maybe less intuitive, partly because of the English conflation of the word computer in "computer science" with modern electronic computers, is that computer science was also born before computers as we know them today, roughly in the 1930's. Alan Turing may be considered as the father of modern computer science. He proposed the concept of a Turing machine [58], a mathematical model that answers the question "What is a computation ?" and the not less fundamental question of what can be computed, and what can not be. While his work is a cornerstone of computer science, mathematicians - or rather computer scientists - have found that Turing machines are very far from being the only answer to this question. On the contrary, an astonishing number of different systems are able to express computations in different forms, from λ-calculus to graph rewriting systems. A lot of these models have exactly the same computing power: they can do as much as a Turing machine, and no more, that is they are Turing-complete. The Church-Turing thesis precisely claims that the notion of computation is captured by Turing machines. This is illustrated nowadays by the large zoology of general purpose programming languages available to developers which are all capable of expressing the same algorithms, but not in the same way, not with the same performance, or not as easily, depending on the task to be performed and the language considered.

The Curry-Howard Isomorphism In the 50s and 80s, Curry and Howard made the fundamental observation that there is a deep connection between mathematical proofs and computer programs [29]. Basically, the mathematical activity of proving a theorem is analogous to the one of programming. Programmers split their programs into small independent and reusable blocks, called functions. They take arguments, their input, and return a result, the output. One can take for example a function that takes a list of textual data as an argument and returns the same list but sorted alphabetically. Then, whenever needed, this function is *applied* to given arguments and produces the desired output. This function may then be used in diverse softwares, such as the contacts manager of a smartphone or an online dictionary. The implementation is independent from the usage: all this function has to know is the list it takes as an argument to provide a result. Mathematicians proceed exactly in the same way. Due to the complexity and the breadth of the mathematical knowledge, results are often broken down into multiple pieces: lemmas, theorems, definitions, propositions, properties, *etc.*. They share the same purpose and characteristics as functions in programs. Application then corresponds to a *cut*, or its more intuitive emanation *modus ponens*: from a proof of $A \implies B$ – or rather, in our new interpretation, a function from A to B – and a proof of A (a data of type A), one deduces B (one gets a data of type B by applying the function to the argument). This intuition can be framed in a very precise framework, both on the logic side and the programming side: proofs in propositional intuitionistic logic are in correspondence with programs of the simply typed λ-calculus, associating the following objects [29]:

Logic	Programming
Propositions	Types
Proofs	Programs
Cut-elimination	Execution

This realization has spawned a substantial corpus of research and results. People started to import concepts of logic inside computer science, and concepts of computer science inside logic. The fact that proofs carry a computational result led to *program extraction*, a technique that synthesizes a program from a proof, for example a program implementing an algorithm from the proof of its termination. Type theory has led to the development of proof assistants, giving an unified environment to write both a program, properties about this program, and proofs of these properties all in the same language. Girard [22] (and independently from a purely programming approach, Reynolds [49]) extended the Curry-Howard isomorphism by introducing polymorphic λ-calculus System

F, which corresponds to second order intuitionistic logic. System F is the foundation of many modern functional programming languages such as Haskell or OCaml. In general, the Curry-Howard isomorphism gave a totally new status to types in programs. From a simple tag addressed to the compiler specifying how generic binary data should be handled, as are types in C for example, they have become first class entities expressed in their own language, which are able to attach a logical specification to a piece of code, to drive optimization, static analysis, *etc.*

Linear Logic Girard introduced Linear Logic (LL) in his seminal paper [23]. The distinctive feature of LL is to be a resource-aware logic, which explains its success as a tool to study computational processes. Indeed, in LL, the usage of resources is controlled and explicit: using an hypothesis once is not the same as using it twice. The native language for LL proofs - or rather, *programs* - are proof nets, a graph representation endowed with a local and asynchronous cut-elimination procedure. The fine-grained computations and the explicit management of resources in LL make it an expressive target to translate various computational primitives. Girard provided two translations of the λ-calculus to LL in its original paper, later clarified as respectively a call-by-value and call-by-name translation of the λ-calculus [42].

Parallelism An important feature of languages based on the λ-calculus, i.e. functional languages, is that they tend to expose the intrinsic parallelism of programs. This has been responsible in part for the dramatic increase in interest in functional languages and functional paradigms over the last years, which have percolated through mainstream programming languages: lambdas, closures, monads, algebraic effects, immutability, type inference, *etc.*. And for a very good reason: processor technology have bumped into physical limits that prevent any further increase in frequency or transistor density. To continue to improve performances, manufacturers resort to rather delivering more and more cores that all run in parallel on a single CPU. On a different scale, the development of cloud computing, coupled with a high internet speed, gives the possibility of performing computations on huge grids of thousands and thousands of processing units. This has caused a total paradigm shift in program development: to take advantage of this new speed up capability, classical imperative programs must often be totally rethought and rewritten. Indeed, mutability, encouraged in imperative programming, affect the whole state of the computer. There is no obvious way to automatically parallelize such programs written with the mental model that they would be executed sequentially and alone on a CPU. Even

when one wants to write parallel programs in such languages, one must resort to specific and error prone mechanisms of synchronization to avoid data races and deadlocks. On the other hand, writing a program which assures the same functionality in a pure functional language forces, by design, to write code that is easily parallelizable. This property is shared by programs expressed in LL, thanks to the local and asynchronous graph-reduction of proof nets.

Concurrency and Effects Automatic parallelization is not the only challenge of modern programming. Either via the internet, or even on a single computer, today programs live in a concurrent world. They are exchanging information with other programs and in the same time competing for resources. Programming models must incorporate this dimension and offer primitives to perform and control this kind of interactions. While λ-calculus is a fundamental tool in the study and design of functional programming languages, mainstream programming languages are pervaded with features that enable productive development such as support for parallelism and concurrency, communication primitives, imperative references, *etc.* Most of them imply side-effects, which are challenging to model, to compose, and to reason about. Some effects, such as imperative references, are non commutative: the order of evaluation matters and may change the result of a computation. Together with parallelism, this induces a typical consequence of concurrency: *non-determinism*. Non determinism means that the same program, executed in the same environment, may give different and incompatible results. One **simple example** to illustrate this is a program composed of three threads in parallel accessing the same memory cell which holds an integer value. The first one tries to read the memory cell, the second one tries to write the value 0 in it, while the third one tries to write the value 1 in it. Depending on which of the writers is executed first, the reader may end up getting either 0 or 1.

Concurrent Curry-Howard The Curry-Howard isomorphism has been extended in many ways since its inception. As already mentioned, Girard extended it to second order with System F. Surprisingly, Curry-Howard is not restricted to constructive logics: Griffin discovered that Pierce's law, whose addition to intuitionistic logic gives back classical logic, corresponds to a construction known long before by Scheme programmers, *call-with-current-continuation* [26]. Krivine extends the computational interpretation to non-logical axioms, such as axioms of set theory in his classical realizability [33]. A long standing research direction has been the extension of Curry-Howard to concurrency, that is, find what could be the logical counter part of concurrent constructions of programming

languages. Since the first similarities between processes calculi and proof nets have been observed, a lot of effort have been put in founding a concurrent version of the Curry-Howard isomorphism, but with limited success. To quote Damiano Mazza, "linear logic proofs are processes, but processes are far from being linear logic proofs and, what is more important, it is unclear at this time whether what is missing has any natural proof-theoretic counterpart" [44].

1.1 Goals and motivations

This thesis motivation is the belief that functional programming languages, and in particular LL - when seen as a programming language through the lens of the Curry-Howard isomorphism - are particularly amenable to parallelization and distributed computing. Moreover, the strong theoretical foundations of LL and the consequent research effort made since its inception also brings various extensions, semantics, abstract machines, complexity analysis techniques, implementation schemes, *etc.* Even when the settings does not exactly match the standard framework of LL, many ideas and notions are still usable or adaptable.

We think that the fine-grained and asynchronous dynamic of LL and its ability to express many computational paradigms make it a good candidate to serve as a common target for compiling languages, i.e. a backend. In this regard, one may consider LL and derived systems as "functional assembly languages". Various translations of calculi to LL have already been investigated. Yet, they are often limited to a very austere setting (pure λ-calculus), do not necessarily accommodate side effects or concern rather modeling languages than actual programming language (cf Section 1.3). Our goal is to go one step further by modeling a language featuring at the same time concurrency, references and replication in LL. Our long-term intent is to exploit the ability of nets to enable independent computations to be done in parallel without breaking the original operational semantics. Concurrency and references are challenging as they lead to non-determinism and non commutativity which are two phenomena alien to standard LL. The approach to translate a concurrent calculus in nets that we propose in Chapter 5 can be seen as a compilation from a *global shared memory* model to a *local message passing* one, in line with proof nets philosophy.

To do so, we introduce a net system that is able to express concurrent behaviors. We develop a key tool for implementing communication primitives in this net system, *routing areas*, which are flexible and compositional. We illustrate their use through the encoding of a concurrent λ-calculus in nets, but we believe that they may be used in various future translations.

We tried to give a picture of what this thesis is about. Let us briefly mention

what it is not. While the language we use to illustrate our tools through the translation features side-effects, a type system, and a usual λ-calculus syntax, it does not come close to something a developer would use in a day-to-day job. It lacks fundamental practical features. But it contains core constructs of concurrency, references, and function calls. We believe it is a convincing proof of concept for the ideas we develop.

We do not claim to propose something like a concurrent Curry-Howard isomorphism either. Our nets are not *correct*, and as such do not correspond to a clear notion of proof. The dynamics is not the same in the source language and in nets, as a reduction step in the source language may be reflected by either zero or several steps in nets. However, given our goals and motivations, we do not require a correspondence as tight as one would expect from a variant of the Curry-Howard isomorphism. What is important is rather the preservation of the operational semantics of the source language.

1.2 Context

Linear Logic The creation of LL by Girard in [23] is a foundational work for this thesis. The idea of proof nets as a graphical parallel representation for proofs is already present in [23], and developed in [25, 10], the latter introducing the Danos-Regnier criterion discriminating correct nets. Ehrhard and Regnier introduced Differential Linear Logic in [15], from which we borrow some rules to handle non-determinism. In his thesis, Tranquilli studies the rewriting system of differential proof nets with exponential boxes [55].

Geometry of Interaction and applications Together with Linear Logic, Girard initiated a program named Geometry of Interaction (GoI), which aims at giving a syntax-free representation of algorithms as independent mathematical objects [24]. He gives an interpretation of proofs of linear logic as endomorphisms on an Hilbert space, and an operator on these endomorphisms, *the execution formula*, which corresponds to the process of cut-elimination. Although this seems to be far away from practical considerations, GoI was later understood in a more concrete manner, as being about interaction paths in proof nets [7]. This gave a very operational presentation of GoI, as a reversible abstract machine which runs by moving a token along a proof net. Mackie saw in GoI a promising and novel approach for compilation [39]. Ghica also uses GoI to enable programming in a high level functional language for integrated circuits [18, 19, 20, 21]. Ghica also demonstrates how, with GoI-style machine, it is trivial to split and distribute computations over multiple processing units [16]. In [51], Schöpp shows that

execution through GoI-like interactive semantics corresponds to well known compiler transformations: CPS-translation and defunctionalization. In [36], Dal Lago *and al.* introduce a multi-token version of the GoI abstract machine for PCF. Multi-tokens do not only enable massive parallelism, but enable the same machine to support both call-by-value and call-by-name depending on the translation of terms, while single token GoI seems to be irremediably call-by-name without extension (Mackie gives a single-token call-by-value GoI compilation for λ-calculus, by allowing jumps). In [46], Ghica and Muroya present a mixed graph rewriting/token machine that achieve call-by-need evaluation. In [28, 47], the authors use categorical constructions to automatically derive GoI machines for various algebraic effects, such as first-order references or non deterministic choice. While we do not give a GoI abstract machine for the nets used in this thesis, these works - and especially the ambition of extending [36] to a richer source language - were a strong motivation for extending known translations in linear logic to new computational paradigms.

Interaction nets and parallelism The research on interactions nets and parallel evaluation is strongly linked to proof nets and geometry of interaction, although it was somehow led in parallel. In [34], Lafont abstracts away the key features of proof nets from the particular setting of LL: what he get is the generic graph rewriting system of Interaction Nets (INs). INs share the same locality and asynchronicity properties as proof nets, which naturally leads to parallel implementation. In [48], Pinto gives a parallel abstract machine for interaction nets. In [30], Jiresch proposes to exploit the massive parallelism of modern GPUs to execute interaction nets, while Kahl gives another parallel implementation in Haskell [31]. In his thesis, Mazza studies the semantics of interaction nets and propose a non-deterministic variant, multiport interaction nets, able to encode concurrent calculi [43]. Dal Lago, Tanaka and Yoshimizu gives a GoI for multiport interaction nets in [35].

Encoding of references and concurrency in proof nets A part of this thesis (Chapter 4 and Chapter 5) is dedicated to the translation of a concurrent calculus to nets. This illustrates the expressivity of routing areas and the nets system. Our work is built on a series of previous works. Girard provided two translations of the simply typed λ-calculus in its original paper [23], later clarified as respectively a call-by-value and call-by-name translation of the λ-calculus [42]. Other works have tackled the intricate question of modeling side-effects. State has been considered in a λ-calculus with references [56]. Another direction which has been explored is concurrency and non-determinism. In [27], Honda and

Laurent gives an exact correspondence between a typed π-calculus polarized proof nets. Ehrhard and Laurent then gives a translation of a fragment of π-calculus, and of acyclic solos (both without replication) in differential interaction nets [14, 13]. In [40], Mackie give an encoding of Kahn process networks to interaction nets. Kahn process networks are a model of computation based on a collection of sequential, deterministic processes that communicate by sending messages through unbounded channels.

Explicit substitutions A specially useful tool in the design of such abstract machines is the notion of explicit substitution, a refinement over β-reduction. The β-reduction of the λ-calculus is a meta-rule where substitution is defined inductively and performed all at once on the term. But its implementation is a whole different story: to avoid size explosion in presence of duplication, mechanisms such as sharing are usually deployed. Abstract machines implement various specific strategies that may either be representable in pure λ-calculus (call-by-value or call-by-name) or for which the syntax needs to be augmented with new objects (e.g. call-by-need or linear head reduction). The mismatch between β-reduction and actual implementations can make the proof of soundness for an evaluator or a compiler a highly nontrivial task. The heart of the theory of explicit substitutions, introduced in [1], is to give substitutions a first class status as objects of the syntax to better understand the dynamics and implementation of β-reduction. It consists in decomposing a substitution into explicit atomic steps. The main ingredient is to modify the β rule so that $(\lambda x.M)N$ reduces to $M[N/x]$, where $[N/x]$ is now part of the syntax. Additional reduction rules are then provided to propagate the substitution $[N/x]$ to atoms.

Studied for the last thirty years [1, 2, 3, 5, 4, 17, 32, 38, 50, 52], explicit substitution turns out to be a crucial device when transitioning from a formal higher-order calculus to a concrete implementation. It has been considered in the context of sharing of mutual recursive definitions [50], higher-order unification [38], algebraic data-types [17], efficient abstract machines [2, 52], cost-model analysis [5], *etc.* The use of explicit substitutions however comes at a price [32]. Calculi with such a feature are sensitive to the definition of reduction rules. If one is too liberal in how substitutions can be composed then a strongly normalizing λ-term may diverge in a calculus with explicit substitutions [45]. If one is too restrictive, confluence on metaterms is lost [9]. The challenge is to carefully design the language to implement desirable features without losing fundamental properties. Several solutions have been proposed to fix these defects [4, 32] for explicit substitutions of term variables.

1.3 Related works

We discuss more in detail work that is closely related to this thesis and has similar objectives. As a matter of fact, our thesis builds on the following papers:

1. In [14], Ehrhard and Laurent proposes a translation of a fragment of π-calculus in differential nets. The π-calculus is designed to be a model of concurrent processes. As such, it features non-determinism, and the authors show that a possible answer to the problem of encoding it in nets is to switch to differential linear logic. The limit of their work is that they interpret a π-calculus without sums nor replication. Correspondingly, they consider a promotion free version of differential nets. The authors introduce the notion of *communication area*, a device introduced to implement communication in nets. Communication areas are specific nets that enable communication between the processes (actually, their encoding as nets) that are plugged in it. In this thesis, we present and study a generalization: *routing areas* (Chapter 3). We discuss how routing areas extend communication areas in Section 3.5. Our approach allows the interpretation of a calculus with replication.

2. In [56], Tranquilli introduces an encoding of a λ-calculus with higher order references and a types and effects system to multiplicative exponential linear logic proof nets. He observes that the translation make explicit the independence of subprograms that operate on distinct references and allow their evaluation to potentially happen in parallel. In order to do so, he uses a monadic translation that allows to encode references inside a pure calculus, which is then translated through the usual call by value translation [42]. This is what Haskellers do, for example, when they use the state monad to emulate references. We build on the ideas of Tranquilli and propose the translation of a calculus with references and parallelism. However the combination of parallelism and references leads to non-determinism and possibly complex interleaved executions. It is not clear at all that this has a natural monadic encoding.

Let us clarify how we relate to these work.

- [56] interprets a deterministic sequential calculus with references inside proof nets. We present a translation of a concurrent and non-deterministic calculus.

- [14] gives a translation from a replication-free π-calculus to differential nets, and introduces communication areas. We generalize communication areas

and introduce routing areas. The calculus we propose features replication. More generally, while π-calculus is a fundamental tool adapted to the study of concurrent processes, it abstracts away too much detail of processes to resemble a programming language. It is rather a modelling tool. On the other hand, the calculus we use is based on the λ-calculus which, with few additions, is arguably a concrete programming language.

We use the concurrent calculus introduced by Amadio in [6] (actually the version described in [41]). It is a λ-calculus enriched with a parallel operator and references that can be read or modified through get and set operations. It is endowed with a types and effects system which ensures termination of well-typed terms. There is a lot of concurrent languages in the literature to chose from. This one is simple yet support core concurrent constructs.

1.4 Contributions

Routing areas In a concurrent imperative language, references are a means to exchange information between threads. In order to implement communication primitives in proof nets, we use and extend the concept of *communication areas* introduced in [14]. A *communication area* is a particular proof net whose external interface, composed of wires, is split between an equal number of inputs and outputs. Inputs and outputs are grouped by pairs representing a plug on which other nets can be connected. They are simple yet elegant devices, whose role is similar to the one of a network switch which connects several agents. Connecting two *communication areas* yields a *communication area* again: this key feature enables their use as modular blocks that can be combined into complex assemblies. In this paper, we introduce *routing areas*, which allows a finer control on the wiring diagram. They are parametrized by a relation which specifies which inputs and outputs are connected. Extending our network analogy, *communication areas* are rather *hubs*: they simply broadcast every incoming message to any connected agent. On the other hand, *routing areas* are more like switches: they are able to choose selectively the recipients of messages depending on their origin. *Routing areas* are subject to atomic operations that decompose the operation of connecting *communication areas*. These operations also have pure algeabric counterparts directly on relations.

We show that *routing areas* are sufficient to actually describe all the normal forms of the fragment of proof nets composed solely of structural rules. The algebraic description of *routing areas* then provides a semantic for this fragment.

Computational content of nets We define and study a system of nets, derived from differential interaction nets, that is powerful enough to express concurrent primitives. However the translation of concurrent processes seems to unavoidably produce nets that do not respect the correctness criterion (as this was already the case in [14]). Without a correctness criterion, the standard full-fledged reduction is neither confluent nor terminating. We observe however that imposing some constraints on this reduction allows to recover interesting properties. Notably, we argue that we may make up for abandoning global properties like termination by proving that particular nets are well-behaved: for example, we prove that if a net is weakly normalizing (without erasing reduction), then it is strongly normalizing. It suffices then to show that nets of interest are weakly normalizing to obtain strong normalization for them, even if general, there are non terminating nets. In the paper introducing differential nets [15], a foot notes precises that "[...] incorrect nets might be interesting from a purely computational point of view". However, to our knowledge, the study of such systems in the literature is very limited.

Expressivity of routing areas : a translation of a concurrent λ-calculus to nets We illustrate the use of routing areas and our nets system by encoding the concurrent λ-calculus with explicit substitutions that we introduce. This translation combines the approach of [56] to accommodate references and [14] to accommodate concurrency. The first one only considers state, and the second one a π-calculus without replication: we propose a translation of a language featuring higher-order references, concurrency and replication. We prove a simulation and an adequacy theorem.

Explicit substitutions for a concurrent λ-calculus In order to define and study the translation from a concurrent λ-calculus to nets, we introduce an intermediate language between the calculus and the nets. While the syntax and the structure of terms are still close to the original language, the dynamic is rather the one of nets, with a local and small-step reduction which distills the information through the whole term step by step. Our contributions are:

1. The definition of a system of *explicit substitutions* for a concurrent λ-calculus with references, both for variables and references.

   The problem we address is the bidirectional property of assignment of references within a term. An assignment for a term variable in a redex only diffuses inward: in $(\lambda x.M)V$, the assignment $x \mapsto V$ only concerns the

subterm M. Instead, a reference assignment $\mathrm{set}(r, V)$ is potentially *global*: it concerns all the occurrences of the subterm $\mathrm{get}(r)$.

Our first contribution is to propose an explicit substitution mechanism to be able to express reference assignment step-wise, as for term-variables.

2. A proof of strong normalization for a typed fragment using a novel *interactive* property.

   Akin to [6], the language we propose is typed and the type-system is enforcing strong-normalization. In the proof of [6] the infinitary structure of terms is restricted to top-level stores. In our setting, this would require infinite explicit substitutions which are subject to duplication, erasure, composition, *etc.*

   Our proof only uses *finite* terms. It has a Game Semantics flavor which we find of interest on its own. Indeed, we use the idea of abstracting the context in which a subterm is executed as an opponent able to interact by sending and receiving explicit substitutions. Moreover, we believe that the finite, interactive technique we develop in this second contribution may be well-suited for different settings such as proof nets or other concurrent calculi.

1.5 Plan

Chapter 2 In Chapter 2, we introduce a graph programming language which is a fragment of differential nets. Our goal is to have a language which is both well-behaved and expressive enough to encode concurrent constructs. We do not impose any correctness criterion. We prove confluence of the reduction, and a conservation theorem, that is if we exclude erasing reduction steps (the one reducing to $\mathbf{0}$), then weakly normalizing nets are strongly normalizing. Section 2.1 defines of simple nets and nets. Section 2.2 defines the reduction rules on simple nets and show how to lift the reduction to formal sums of simple nets, that is nets. The next sections are dedicated to study various properties that the reduction satisfies. Section 2.3 shows confluence via the definition of a parallel reduction inspired by the proof of confluence of the λ-calculus. At last, Section 2.4 studies the zoology of terminating and non terminating nets. While the reduction we define is not terminating in general, we show that removing the erasing rule $\xrightarrow{\mathbf{0}}$ allows to show a property which imposes a strong condition on nets to be non terminating, limiting this possibility.

Chapter 3 In Chapter 3, we introduce routing areas, an extension of communication areas [14]. In order to do so, various technical tools are introduced first. Section 3.1.1 introduces the notion of multirelation, a quantitative extension of relations between sets, which will be used to give an algebraic description of a routing area. Section 3.1.2 defines (co)contraction trees, which can be seen as generalized n-ary (co)contractions, and derives corresponding reduction rules. (Co)contraction trees will be the basic ingredients for building routing areas. Section 3.1.3 briefly reviews the correctness criterion which has been mentioned in Chapter 2. It will be used for various proofs of this chapter, as routing areas does satisfy the correctness criterion. Then, everything is in place to construct routing areas, which is done in Section 3.1.4.

Section 3.2 describes constructions that allow to combine simple routing areas into more complex ones. We give two fundamental operations, juxtaposition and trace, and define the composition of routing areas from these two.

In Section 3.3, we observe that the interpretation of routing areas as communication devices extends to a larger class of nets, *routing nets*, which include routing areas. The necessity of recasting the notion of connection between endpoints in this setting leads us to give a formal treatment of path in Section 3.3.1. This tool allows to prove in Section 3.3.2 that the normal form of a routing net is a routing area. This provides a semantics that associates the multirelation describing its normal form to a routing nets. We study the properties of this semantics in Section 3.3.3, and show that it is closely related to paths.

Chapter 4 In Chapter 4, we introduce the concurrent λ-calculus with explicit substitutions λ_{cES}, with both substitutions for variable and references. The syntax and the operational semantics are given in Section 4.1. In Section 4.2, we introduce a type and effect system for λ_{cES}, adapted from the one of λ_{C} [6]. In presence of higher-order references, simple types does not suffice to entail strong normalization. On must use additional constraints: we use the idea of stratification, proposed by Boudol [8]. It is nonetheless not trivial to extend the proof of strong normalization of well typed terms to the calculus with explicit substitutions. This is done in Section 4.3, where we use an interactive technique that is reminiscent of Game Semantics. At last, Section 4.4 focuses on the relation between λ_{cES} and the language that inspired it, λ_{C}. While λ_{cES} should be thought of as a version of λ_{C} with explicit substitutions, the latter is not a proper sublanguage of the former. We can however define an embedding of λ_{C} in λ_{cES}. We prove a simulation for this embedding in Section 4.4.2. In Section 4.4.3, we give an adequacy theorem, which is complementary to the simulation result. It states that the values than can be computed by a term of λ_{C} and the ones

that can be computed by its translation in λ_{cES} are essentially the same.

Chapter 5 In Chapter 5, we illustrate the ideas we set up in previous chapter and propose an encoding of our concurrent calculus with explicit substitutions λ_{cES} to nets. In Section 5.1, we give the definition of the translation. Section 5.1.1 reviews the monadic translation of [56] and explain how we adapt it to our setting. In Section 5.1.2, we detail how and why routing areas are used to design the translation. Then, Section 5.1.3 gives the encoding of λ_{cES} terms in nets.

Section 5.2 is dedicated to the proof of simulation. It introduces some practical tools in Section 5.2.1, while the three following sections, Section 5.2.2, Section 5.2.3 and Section 5.2.4 treat the all the different cases.

Section 5.3 shows that the translation of a normal form is a strongly normalizing net. This is almost enough to show that the translation of any well-typed term is strongly normalizing, but we still miss an intermediate result to prove this claim (cf Section 5.5). We also show an adequacy property akin to the one between λ_{cES} and λ_{C} given in Chapter 4.

Chapter 2

Nets

In this chapter, we introduce a graph language which is a fragment of differential nets. Our goal is to define a fragment which is both well-behaved and expressive enough to encode concurrent constructs. By well-behaved, we mean that it satisfies a set of properties one would reasonably expect from a nets system. Concurrency, here in the form of the coexistence of both references and parallelism, leads to non-determinism. State is a non-commutative effect, as the order of evaluation matters and may change the result of a computation. On the other hand, parallelism allows multiple subprograms - in the form of threads - to be executed in an arbitrary order, whence the non-determinism. One simple example to illustrate this is a program composed of three threads in parallel accessing the same memory cell which holds an integer value. The first one tries to read the memory cell, the second one tries to write the value 0 in it, while the third one tries to write the value 1 in it. Depending on which of the writer is executed first, the reader may end up getting either 0 or 1. However, non-determinism seems to be outside of the realm of traditional LL: confluence enforces that the final outcome of a computation is unique and independent from the path of reduction. Fortunately, there is a well defined extension of LL, Differential LL, which supports non-determinism by operating not on terms but rather on formal sums of terms, where summands represent different and incompatible outcomes.

Correctness We will not require nets to verify the correctness criterion of Differential LL here: it turns out that the translation of concurrent calculi seems to naturally produce nets that do not respect the correctness criterion, which was already the case in [14] for example (cf Section 2.6 for an extended discussion). A world without correctness is a wild world: in this setting, the full fledged reduction of (differential) LL is neither confluent (Figure 2.1) nor terminating (Figure 2.3, Figure 2.2). The goal of this chapter is the definition and the study

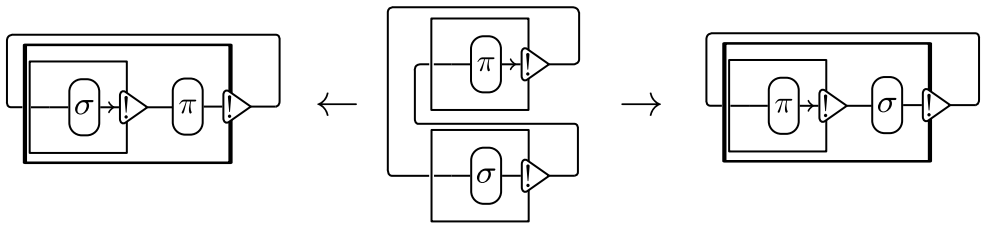


Figure 2.1: Example of non-confluence

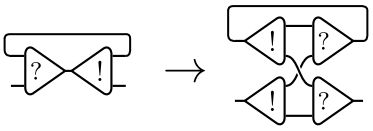


Figure 2.2: Self replicating pattern using a cocontraction

of a manageable system in this context.

Overview Section 2.1 defines simple nets and nets. Section 2.2 defines the reduction rules on simple nets and shows how to lift the reduction to formal sums of simple nets, that is nets. The next sections are dedicated to study the various properties that the reduction satisfies. Section 2.3 shows confluence via the definition of a parallel reduction inspired by the proof of confluence of the λ-calculus. At last, Section 2.4 studies the zoology of terminating and non terminating nets. While the reduction we define is not terminating in general, we show that removing one problematic rule allows to show a property which imposes a strong condition on nets to be non-terminating, limiting this possibility.

2.1 Syntax

We can decompose our system into three layers (fragments):

Multiplicative The multiplicative fragment is composed of the conjunction $\otimes$ and the dual disjunction $⅋$. These connectors can express the linear implication $A \multimap B$ as $A^\perp ⅋ B$ and this fragment is sufficient to encode a

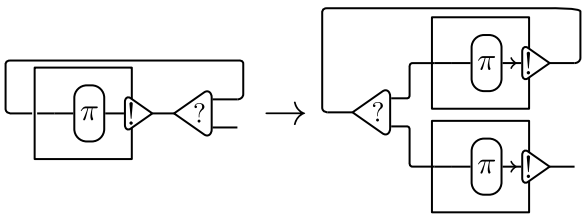
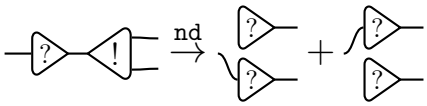


Figure 2.3: Self replicating pattern using a box

linear λ-calculus, where all bound variables must occur exactly once in the body of an abstraction.

Exponential The exponential fragment enables structural rules to be applied on particular formulas distinguished by the ! modality. Structural rules correspond to duplication (contraction) and erasure (weakening) : the multiplicative exponential fragment regain the power to use an argument an arbitrary number of times. This is the setting of LL to interpret the λ-calculus.

Differential Non-determinism is expressed by using two rules from Differential LL: cocontraction and coweakening. Semantically, contraction is thought of as a family of diagonal morphisms $cntr_A : !A \to !A \otimes !A$, where $cntr_A$ takes a resource $!A$ and duplicates it into a pair $!A \otimes !A$. Dually, cocontraction is a morphism going in the opposite direction, packing two resources of the same type into one : $cocntr_A : !A \otimes !A \to !A$. What happens when the resulting resource is to be consumed ? There are two incompatible possibilities : either the left one is used and the right one is erased, or vice-versa. This corresponds to the following rule $\overset{\mathtt{nd}}{\to}$ of Table 2.2:



Here, the reduction produces the non-deterministic sum of the two outcomes. Cocontraction will be used as an internalized non-deterministic choice. While weakening $weak_A : !A \to \mathbf{1}$ erases a resource, the dual coweakening produces a resource ex nihilo: $coweak_A : \mathbf{1} \to !A$. This is like a Pandora box: it can be duplicated or erased, but any attempt to consume it will turn the whole summand to $\mathbf{0}$, the neutral element of the non-deterministic sum. Coweakening is the neutral element of cocontraction.

Notation 2.1.1. We recall here some vocabulary of rewriting theory. An *abstract rewriting system* (ARS) is a pair $(A, \to)$ where A is a set and $\to \subseteq A \times A$ is a binary relation on A, called a *rewriting relation* or a *reduction*. For an ARS $(A, \to)$, we write $\to^+$ for the transitive closure and $\to^*$ for the reflexive transitive closure.

Let $t \in A$, a *reduction sequence* of t is a finite or infinite sequence $(t_0, t_1, \ldots, t_i, \ldots)_{i<N}$ (where $N \in \mathbb{N} \cup \{\infty\}$) of elements of A such that:

- $t_0 = t$
- $t_i \to t_{i+1}$

Let $t \in A$, we say that t is:

- *a normal form* if there exists no $t' \in A$ such that $t \to t'$.
- *weakly normalizing* if there exists a normal form n such that $t \to^* n$
- *strongly normalizing* if there exists no infinite reduction sequence of t. Note that a strongly normalizing element is in particular weakly normalizing.
- *confluent* if for all u, u' such that $u \,{}^*\!\leftarrow t \to^* u'$, there exists v such that $u \to^* v \,{}^*\!\leftarrow u'$

A rewriting relation is confluent (resp. weakly normalizing, strongly normalizing) if all elements $t \in A$ are.

Definition 2.1.2. Simple nets
Given a countable set, whose elements are called ports, a simple net is given by

1. A finite set of *ports*
2. A finite set of *cells*. A cell is a finite non-empty sequence of pairwise distinct ports, and two cells have pairwise distinct ports. The first port of a cell c is called the *p*rincipal port and written $p(c)$, and the $(i+1)$th the ith auxiliary port, noted $p_i(c)$. The number of auxiliary ports is called the *a*rity of the cell. A port is free if it does not occur in a cell.
3. A labelling of cells by symbols amongst $\{1, \otimes, \parr, ?, !, !_p\}$ where p is a strictly positive integer. We ask moreover that the arity respects the following table:

Symbol	1	$\parr$	$\otimes$	$?$	$!$	$!_p$
Arity	0	2	2	$0, 1$ or 2	$0, 1$ or 2	p

4. A partition of the set of ports into pairs called *w*ires. A wire with one (resp. two) free port is a free (resp. floating) wire.
5. A labelling of wires, which is a map from wires to the following fragment of LL: $F ::= 1 \mid \bot \mid F \parr F \mid F \otimes F \mid !F \mid ?F$. A wire (p_1, p_2) labelled by F is identified with the reversed wire (p_2, p_1) labelled by $F^\perp$.
6. To each $!_p$ nodes - called an *exponential box* - is associated a simple net in an inductive manner. Let $!_{p+1}$ be a box where the wires attached to its ports (oriented outwards) are labelled by $!A, ?B_1, \ldots, ?B_p$. The associated simple net S must have no floating wires and precisely $n \geq 1$ free wires $w_1, \ldots, w_p$. Assuming the orientation to be toward their free port, w_1 is labelled by A and for $1 < i \leq p$, w_i is labelled by $?B_i$.

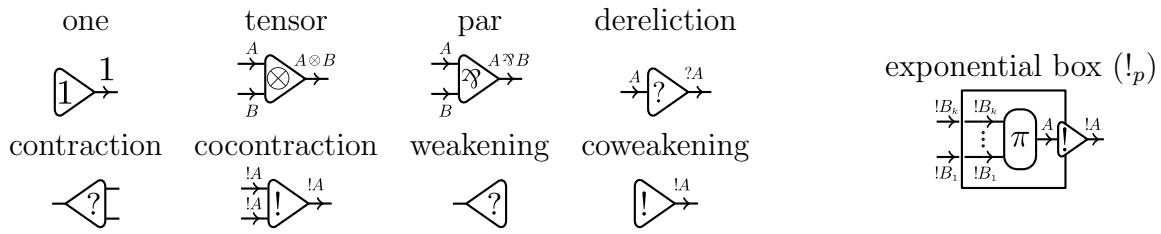


Table 2.1: Cells and box

The set of simple nets is written $\mathcal{S}_N$.

Definition 2.1.3. Depth
Let S be a simple net. The depth of a port p in S is defined to be:

- 0 if p is a port of S
- $n+1$ if it is at depth n in a simple net associated to a box of S

Similarly, we define the depth of a cell or a wire as the depth of its ports. A port, cell or wire occurring at depth 0 is said to be at the *surface*.

Remark 2.1.4. As the simple nets are defined inductively, the depth of a port is a well defined positive integer.

The different kind of cells are illustrated in Table 2.1. We directly represent $!_p$ cells as their associated simple net delimited by a rectangular shape. The well-founding condition ensure that such a representation is always possible. We impose that labels of wires connected to a cell respect the one given in Table 2.1, i.e that nets are *well-typed*.

We define the set of nets $\mathcal{P}_N$ as the set of finite formal sums of simple nets, where each summand represents a different outcome in a non-deterministic context.

Definition 2.1.5. Nets
The set $\mathcal{P}_N$ of nets is defined as formal sums of simple nets:

$$\mathcal{P}_N = \{\sum_{i=1}^{n} \mathcal{S}_i \mid n \geq 0,\ \mathcal{S}_i \text{ is a simple net}\}$$

The empty sum is denoted by $\mathbf{0}$.

Let us now give the reduction on nets.

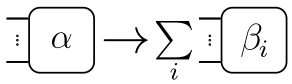
2.2 Reduction

In this section, we define the rewriting system $(\mathcal{P}_N, \Rightarrow)$ on nets. To do so, we proceed in two stages:

1. We define (Section 2.2.1) a base reduction from simple nets to nets id est a relation $\rightarrow \subseteq \mathcal{S}_N \times \mathcal{P}_N$. We start from atomic rules describing how an elementary pattern may be rewritten , and then extend the reduction by allowing patterns to be reduced in an arbitrary context (Definition 2.2.1).
2. In Section 2.2.2, we lift $\rightarrow$ to reduction on nets, id est a relation $\Rightarrow \subseteq \mathcal{P}_N \times \mathcal{P}_N$ (Definition 2.2.3).

2.2.1 Base reduction $\rightarrow$

We define the atomic rules as relations between $\mathcal{S}_N$ and $\mathcal{P}_N$. By convention, we write them as one or several symbols over an arrow, such as $\xrightarrow{\texttt{er}}$ for example. These are the base cases that describe how an atomic pattern can be rewritten in a simple net. In a non-deterministic context, it can produces several outcomes, hence the result is a net and not just a simple net. Atomic rules are illustrated in Table 2.2. They have the following general form:



where the interface of α and each β_i are the same. The pattern on the left hand side of an atomic rule is called a reducible expression, or a *redex*. The right hand side of a rule is a *reduct*.

From these atomic rules, the reduction is then extended - still as a relation between $\mathcal{S}_N$ and $\mathcal{P}_N$ - by allowing redexes to be reduced inside an arbitrary context, that is a larger simple net that contains the redex. Reduction inside boxes is also permitted but only for a few rules (see Section 2.2.3).

We explain and detail the rules hereafter.

Multiplicative rules We have the standard multiplicative rule of LL

- $\xrightarrow{\texttt{m}}$: eliminate a tensor facing a par.

Exponential rules There are four rules operating on boxes:

- $\xrightarrow{\texttt{e}}$: open a box facing a dereliction
- $\xrightarrow{\texttt{d}}$: duplicate a box facing a contraction

- $\xrightarrow{\texttt{er}}$: erase a box facing a weakening
- $\xrightarrow{\texttt{c}}$: compose a box linked to the auxiliary door of another one, by putting it inside

They are like the standard exponential rules of LL, but the difference is that the box that is opened, duplicated, erased or put inside another one is required to be *closed*, meaning that it must not have auxiliary ports. The reason for this restriction is motivated in Section 2.2.3.

Differential rules The last group of rules concerns the differential part: cocontraction and coweakening. The main rule is the non-deterministic rule $\xrightarrow{\texttt{nd}}$. The others specify how the differential part interacts with the other cells.

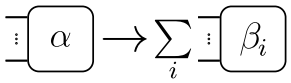
- $\xrightarrow{\texttt{nd}}$: this is the non-deterministic reduction, which is the only one (beside the degenerate case of the $\mathbf{0}$) to produce an actual sum. When a dereliction tries to consume a resource composed of two resources packed through a cocontraction, two different outcome are produced. In the first summand, the left resource in consumed and the other is erased by a weakening. In the second summand, the inverse choise is made.
- $\xrightarrow{\texttt{ba}}$: the bialgebra rule express a commutation property between contraction and cocontraction. An operationnal interpretation is that gathering two resources non-deterministically through a cocontraction, and then copying this pack, is the same as first copying the resources individually and then packing the copy.
- $\xrightarrow{\texttt{s}_1}$: this reduction expresses the interaction between coweakening and contraction. Coweakening is copied by a contraction just as a regular resource.
- $\xrightarrow{\texttt{s}_2}$: this is the dual rule that expresses the interaction between weakening and cocontraction. It says that erasing a non-deterministic packing amount to individually erases each component.
- $\xrightarrow{\epsilon}$: this rules says that coweakening is erased by a weakening, as a regular resource.
- $\xrightarrow{\mathbf{0}}$: finally, this rules states than trying to consume a resource produced by a coweakening turns the whole simple net to $\mathbf{0}$.

Atomic rules only allow to reduce specific redexes. We want to be able to reduce these redexes whenever they appear in a larger simple net, by allowing

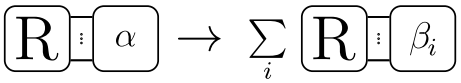
the reduction to be performed up to an arbitrary context. We treat differently the extension of the reduction at surface and inside boxes, where only $\xrightarrow{\mathtt{e}}$ and $\xrightarrow{\mathtt{er}}$ may be performed.

Definition 2.2.1. Reduction $\rightarrow$
We define the reduction relation $\rightarrow \subseteq \mathcal{S}_N \times \mathcal{P}_N$ by:

- (Surface) If

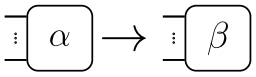


  by an atomic rule, then

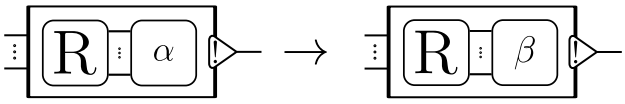


  Note that R may be empty, which implies that atomic rules are included in $\rightarrow$.

- (Inner) If



  by the $\xrightarrow{\mathtt{e}}$ or $\xrightarrow{\mathtt{er}}$ rule, then



When one translates a type derivation of an intuitionistic system (take simply-typed λ-calculus for example) to nets, the explicit appearance of structural rules involves arbitrary choices. Indeed, there are many possbilities for the order of contractions of variables, the location of weakenings, *etc.* Not only these choices are irrelevant (the different representations behave the same way), but they are often not stable by reduction as the simulation of a reduction step in nets may end up with a different representation of the result. To avoid these issues, we quotient the nets by associativity and commutativity of (co)contraction (Table 2.3). The fact that this quotient is compatible with reduction confirms the irrelevance of these differences.

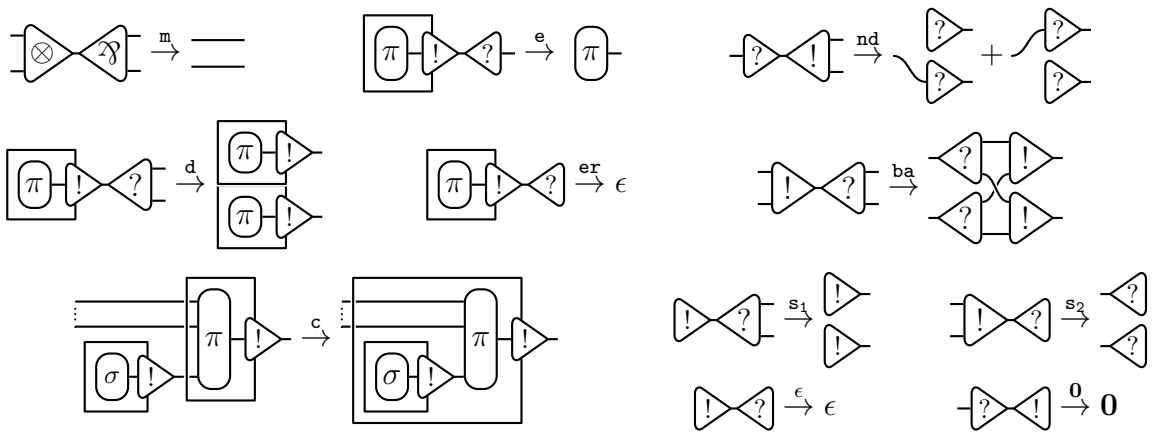


Table 2.2: Reduction rules

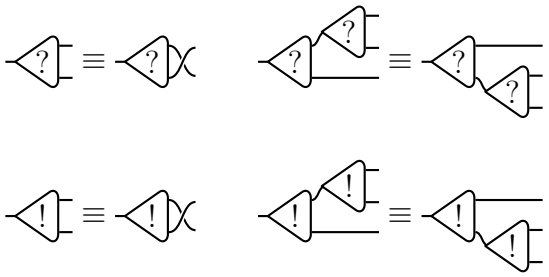


Table 2.3: Equivalence relation

2.2.2 Lifting reduction to nets

The reduction is lifted to nets by allowing the reduction of an arbitrary number of summands at once (Definition 2.2.2). As we consider different summands as different results living in parallel universes, this make sense to be able to reduces several of them at once.

Definition 2.2.2. Sum reduction
Let $\to_a \subseteq \mathcal{S}_N \times \mathcal{P}_N$ be a reduction from simple nets to nets. We define the ARS $(\mathcal{P}_N, \Rightarrow_a)$ by: $\mathcal{R} \Rightarrow_a \sum_i^n \mathcal{T}_i$ if

1. Either $S_i = \mathcal{T}_i$ or $S_i \to_a \mathcal{T}_i$
2. There is at least one i such that $S_i \to_a \mathcal{T}_i$

Definition 2.2.3. The $\Rightarrow$ reduction on nets
The reduction $(\mathcal{P}_N, \Rightarrow)$ on nets is defined as the sum lifting of $\to$.

We define a second notion of reduction on sums, which requires all summands that are not a normal form to perform a reduction step. This will prove useful when discussing properties about termination.

Definition 2.2.4. Total reduction
Let $\rightarrow_a \subseteq \mathcal{S}_N \times \mathcal{P}_N$ be a reduction from simple nets to nets. We define the rewriting system $(\mathcal{P}_N, \Rightarrow_a^{tot})$ by: $\mathcal{R} \Rightarrow_a^{tot} \sum_i^n \mathcal{T}_i$ if either S_i is a normal form and $S_i = \mathcal{T}_i$, or $S_i \rightarrow_a \mathcal{T}_i$.

The total reduction corresponding to $\rightarrow$ is $(\mathcal{P}_N, \Rightarrow^{tot})$.

2.2.3 About closed and surface reduction

Most of the rules are only allowed at the surface, meaning that they can not be performed inside boxes. This constraint is necessary to be able to encode weak reduction strategies of the λ-calculus. $\xrightarrow{\mathtt{e}}$ and $\xrightarrow{\mathtt{er}}$ are however allowed as they are harmless and do not correspond to any real computational step important inside λ-calculus. The closeness constraint, requiring boxes to not have auxiliary ports, fixes the non confluence of Figure 2.1 and the non-termination of Figure 2.3. But it does not change the fact that Figure 2.2 is looping, which seems harder to avoid without a correctness criterion. We will see in the section about termination that the closeness constraint still allows to recover termination properties weaker than strong normalization for some fragments, but sufficient for our purposes.

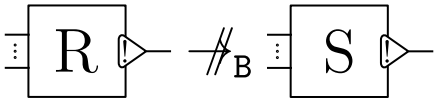
2.3 Confluence

In this section we prove that $\Rightarrow$ is confluent (Theorem 2.3.5). We use the same approach as in the proof of confluence of λ-calculus: we define a parallel reduction $\Rrightarrow$, which allows to perform an arbitrary number of steps in parallel. Parallel means that we can reduce several redexes at once, but only those that were originally present, as opposed to those that are created during the reduction. We have to treat separately reductions that are allowed anywhere, and those that are only allowed at the surface. This the reason why we first have to define a parallel reduction system only for the reductions allowed everywhere, $(\mathcal{S}_N, \not\Rrightarrow_{\mathtt{B}})$. From this, we extend this system to a reduction from $\mathcal{S}_N$ to $\mathcal{P}_N$, $\not\Rrightarrow$. We finally lift it to sums of net to define the complete parallel reduction $(\mathcal{P}_N, \Rrightarrow$.

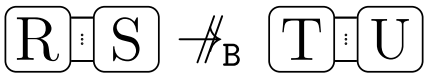
Definition 2.3.1. $\not\Rrightarrow_{\mathtt{B}}$
Let $(\mathcal{S}_N, \rightarrow_{\mathtt{B}})$ be the reduction system on $\mathcal{S}_N$ defined by the atomic rules that are allowed inside and outside boxes, that is $\rightarrow_{\mathtt{B}} = \xrightarrow{\mathtt{e}} \cup \xrightarrow{\mathtt{er}}$. We define the reduction system $(\mathcal{S}_N, \not\Rrightarrow_{\mathtt{B}})$ by:

- $R \not\Rrightarrow_{\mathtt{B}} R$
- If $R \rightarrow_{\mathtt{B}} S$, then $R \not\Rrightarrow_{\mathtt{B}} S$.

- If $R \not\to_{\mathtt{B}} S$, then



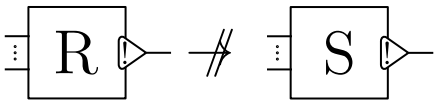
- If $R \not\to_{\mathtt{B}} T$ and $S \not\to_{\mathtt{B}} U$, then



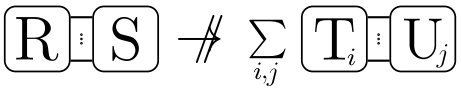
We can now define the complete parallel reduction on simple nets.

Definition 2.3.2. $\not\Rightarrow$
We define the reduction $\not\Rightarrow$ from simple nets to nets by:

- $R \not\Rightarrow R$
- If $R \to \mathcal{S}$, then $R \not\Rightarrow \mathcal{S}$
- If $R \not\to_{\mathtt{B}} S$, then



- If $R \not\Rightarrow \sum_i T_i$ and $S \not\Rightarrow \sum_j U_j$, then



Definition 2.3.3. Parallel reduction
$(\mathcal{P}_N, \not\Rightarrow)$ is defined as the sum lifting of $\not\Rightarrow$.

Proposition 2.3.4. Properties of parallel reduction
The parallel reduction $(\mathcal{P}_N, \not\Rightarrow)$ verifies:

1. $\Rightarrow \subseteq \not\Rightarrow \subseteq \Rightarrow^*$
2. $\not\Rightarrow$ has the diamond property

Proof. The proof is performed by induction on the size of simple nets for $\not\Rightarrow$. The main point is that there is no actual critical pair. Thanks to the reflexivity of $\not\Rightarrow$, the diamond property is preserved when lifting the reduction to $\mathcal{P}_N$. □

Theorem 2.3.5 is an immediate corollary of Proposition 2.3.4. After confluence, we treat the issue of termination.

Theorem 2.3.5. Confluence
The system $(\mathcal{P}_N, \Rightarrow)$ is confluent.

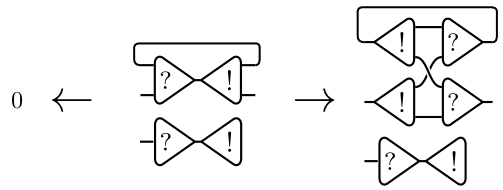


Figure 2.4: Counter example for (T_1)

2.4 Normalizing and non-normalizing nets

Since we do not enforce any correctness criterion, we lose the guarantee of strong normalization (cf Figure 2.2 and Figure 2.3). Nonetheless some fragments of nets obtained by removing some reduction rules verify strong normalization. Others verify the (T_1) property (see Definition 2.4.3) which is not per se a termination property but which seriously limits the candidates for non-normalizing nets. This properties states that either a net is strongly normalizing, or it does not have a normal form at all. In this case, the mere existence of a normal form of a net is sufficient to deduce the finiteness of all reduction sequences of this net.

There is no hope for the full system $(\mathcal{P}_N, \Rightarrow)$ to satisfy (T_1), because the $\xrightarrow{\mathbf{0}}$ rule may erase a looping summand. See Figure 2.4 for a counter-example to (T_1) which is a slight modification of the looping example: at any moment, this net can either duplicate the looping pattern or reduce in one step to $\mathbf{0}$. However, Theorem 2.4.5 shows that removing the $\xrightarrow{\mathbf{0}}$ rule is sufficient to recover (T_1).

The proof is carried out in three steps:

1. We first consider the subsystem of $(\mathcal{P}_N, \Rightarrow^{tot})$ composed only of reduction rules applied at the surface: $(\mathcal{P}_N, \rightharpoonup)$. We show that $(\mathcal{P}_N, \rightharpoonup)$ has the diamond property (Proposition 2.4.11), from which we deduce that it verifies (T_1) (Corollary 2.4.12).

2. We consider the remaining rules performed only inside boxes gathered in the subsystem $(\mathcal{P}_N, \rightsquigarrow)$. We show that it is strongly normalizing (Proposition 2.4.10) and has a commutation property with $(\mathcal{P}_N, \rightharpoonup)$ (Proposition 2.4.7)

3. This commutation property allows us to show that (T_1) can be transported from $(\mathcal{P}_N, \rightharpoonup)$ to $(\mathcal{P}_N, \rightharpoonup \cup \rightsquigarrow)$, which is nothing but $(\mathcal{P}_N, \Rightarrow^{tot})$.

4. Finally, we show that the fact that $(\mathcal{P}_N, \Rightarrow^{tot})$ is (T_1) implies that $(\mathcal{P}_N, \Rightarrow)$ is (T_1).

Let us lay out the definitions of the reductions and the properties we use.

Definition 2.4.1. $(\mathcal{P}_N, \rightharpoonup)$
Let $\overset{\texttt{surf}}{\to}$ be the union of all atomic reduction rules. Then $(\mathcal{P}_N, \rightharpoonup)$ is the total lifting $(\mathcal{P}_N, \Rightarrow^{tot})$ of the surface extension of $\overset{\texttt{surf}}{\to}$ up to context. By surface extension, we mean that we ignore the second point of the definition of the extension up to context and do not allow any reduction inside boxes.

Definition 2.4.2. $(\mathcal{S}_N, \rightsquigarrow)$
Let $\overset{\texttt{inner}}{\to} = \overset{\texttt{e}}{\to} \cup \overset{\texttt{er}}{\to}$. Then $(\mathcal{P}_N, \rightsquigarrow)$ is the total lifting $(\mathcal{P}_N, \Rightarrow^{tot})$ of the inner extension of $\overset{\texttt{inner}}{\to}$ up to context. By inner extension, we mean that we ignore the first point of the definition of the extension up to context and only allow reductions inside boxes.

Definition 2.4.3. (T_1)
A rewriting system $(\mathcal{P}_N, \hookrightarrow)$ on nets verifies (T_1) if, whenever a net $\mathcal{R}$ is $\hookrightarrow$-weakly normalizing, then $\mathcal{R}$ is $\hookrightarrow$-strongly normalizing.

Definition 2.4.4. Diamond
A rewriting system $(\mathcal{P}_N, \hookrightarrow)$ is said to be *diamond* if whenever $\mathcal{R}_1 \hookleftarrow \mathcal{R} \hookrightarrow \mathcal{R}_2$, then either $\mathcal{R}_1 = \mathcal{R}_2$, or there exists $\mathcal{T}$ such that $\mathcal{R}_1 \hookrightarrow \mathcal{T} \hookleftarrow \mathcal{R}_2$.

We can state the main theorem of this section:

Theorem 2.4.5. $(\mathcal{P}_N, \Rightarrow)$ verifies (T_1).

The next proposition states the commutation property between $(\mathcal{P}_N, \rightharpoonup)$ and $(\mathcal{P}_N, \rightsquigarrow)$. The proposition actually defines a transformation of reduction sequences: given a reduction composed of two consecutive blocks of respectively $\rightharpoonup$ and $\rightsquigarrow$ reductions, we can produce a new reduction sequence with two new blocks in the reverse order ($\rightsquigarrow$ followed by $\rightharpoonup$). Moreover, if there is at least one step in the $\rightharpoonup$ block, then the transformed sequence will also have at least one step in the $\rightharpoonup$ block. This last point is fundamental for the lifting of the (T_1) property from $\rightharpoonup$ to $\rightharpoonup \cup \rightsquigarrow$: it would not work anymore without it.

Remark 2.4.6. In principle, the different properties that we show about $\rightharpoonup$ and $\rightsquigarrow$ should be proved on arbitrary sums of simple nets in regard to the definition of these reductions. This would however unnecessarily make for verbose proofs, which are already rather technical. This is why we will implicitly restrict ourselves in the proofs to the base case where the starting nets are simple nets (for example, $\mathcal{S}$ and $\mathcal{R}$ in the proof of Proposition 2.4.7). Thanks to the definition of the total reduction on sums, the base case is sufficient to extends all these properties to the total sum reduction.

Proposition 2.4.7. Commutation
Let $\mathcal{S} \rightsquigarrow^* \mathcal{R}$ and $\mathcal{R} \multimap^+ \mathcal{R}'$. Then there exists $\mathcal{S}'$ such that

$$\begin{array}{ccc} \mathcal{R} & \multimap^+ & \mathcal{R}' \\ {\scriptstyle *}\wr & & {\scriptstyle *}\wr \\ \mathcal{S} & \multimap^+ & \mathcal{S}' \end{array}$$

Proof. Proposition 2.4.7
We proceed by induction on both the length of the reduction $\mathcal{S} \rightsquigarrow^* \mathcal{R}$ and $\mathcal{R} \to^+ \mathcal{R}'$. We consider first the following diagram where both reductions are one-step :

$$\begin{array}{ccc} \mathcal{R} & \multimap & \mathcal{R}' \\ \wr & & \\ \mathcal{S} & & \end{array}$$

We observe that the $\multimap$-redexes of $\mathcal{R}$ and $\mathcal{S}$ are in a one-to-one correspondence. Indeed, as $\rightsquigarrow$ reduction happens inside boxes, it can not create nor erase any $\multimap$-redex. We refer to the redex involved in $\mathcal{S} \rightsquigarrow \mathcal{R}$ (resp. $\mathcal{R} \multimap \mathcal{R}'$) as $\text{red}^{\rightsquigarrow}$ (resp. $\text{red}^{\multimap}$), and to the rule applied as $\text{rule}^{\rightsquigarrow}$ (resp. $\text{rule}^{\multimap}$). We consider different cases:

(a) If $\text{rule}^{\multimap}$ does not involve a box (it is a $\xrightarrow{\mathtt{e}}$,$\xrightarrow{\mathtt{c}}$,$\xrightarrow{\mathtt{er}}$ or $\xrightarrow{\mathtt{d}}$ step), then the diagram can be closed as a square with a one step reduction in both directions. This is because in this case, $\text{rule}^{\multimap}$ and $\text{rule}^{\rightsquigarrow}$ do not interact.

(b) If $\text{rule}^{\multimap}$ is $\xrightarrow{\mathtt{er}}$ (box erasing), we can erase the corresponding one in $\mathcal{S}$: $\mathcal{S} \multimap \mathcal{S}'$. If $\text{red}^{\rightsquigarrow}$ was located in this precise box, it is deleted, and $\mathcal{S}' = \mathcal{R}'$. Otherwise, the two reductions are independent and commute. In any case we can reduce $\mathcal{S}$ to $\mathcal{S}'$ in one step by erasing this box and $\mathcal{S}'$ to $\mathcal{R}'$ by at most one $\rightsquigarrow$ step.

(c) If $\text{rule}^{\multimap}$ is $\xrightarrow{\mathtt{e}}$ (box opening), we can open the corresponding box in $\mathcal{S}$. We have three different situation for $\text{red}^{\rightsquigarrow}$:

1. It is in in a different box
2. It is at depth 1 in the opened box
3. It is at depth $d > 1$ in the opened box

In cases 1 and 3, the two reduction do not interact and thus commute in one step. In case 2, if we open first the box through the step $\text{rule}^{\rightharpoonup}$, then it turns $\text{red}^{\rightsquigarrow}$ into a $\rightharpoonup$-redex. We close the diagram by performing this additional step, in which case $\mathcal{S} \rightharpoonup^2 \mathcal{S}' = \mathcal{R}'$.

In cases (a),(b) and (c), the length of the reduction between $\mathcal{R}'$ and $\mathcal{S}'$ never exceeds one, hence we can always fill the diagram as as in Figure 2.5, where the bottom line does not involve duplication. $\mathcal{S}' \underline{\rightsquigarrow} \mathcal{R}'$ means $\mathcal{S}' = \mathcal{R}'$ or $\mathcal{S}' \rightsquigarrow \mathcal{R}'$.

$$\begin{array}{ccc} \mathcal{R} & \rightharpoonup & \mathcal{R}' \\ \updownarrow & & \updownarrow| \\ \mathcal{S} & \rightharpoonup^+ & \mathcal{S}' \end{array}$$

Figure 2.5: Cases (a),(b), and (c)

(d) If $\text{rule}^{\rightharpoonup}$ is $\xrightarrow{\mathtt{d}}$ (box duplication), performing the duplication on the corresponding box in $\mathcal{S}$ may also duplicate $\text{red}^{\rightsquigarrow}$ if it is located in the same box. In this case, we have to perform two $\rightsquigarrow$ step to recover $\mathcal{R}' : \mathcal{S} \rightharpoonup \mathcal{S}' \rightsquigarrow^2 \mathcal{R}'$. If it is in a different box, the two reductions commute in one step. In any case, we can fill the diagram as in Figure Figure 2.6.

$$\begin{array}{ccc} \mathcal{R} & \rightharpoonup & \mathcal{R}' \\ \updownarrow & & {}^{*}\!\updownarrow \\ \mathcal{S} & \rightharpoonup & \mathcal{S}' \end{array}$$

Figure 2.6: Case (d)

Let us now prove the lemma. We will fill the following diagram by induction on k:

$$\begin{array}{ccc} \mathcal{R} & \rightharpoonup & \mathcal{R}' \\ {}^{k}\!\updownarrow & & \\ \mathcal{S} & & \end{array}$$

We can do the same case analysis on $\mathcal{R} \rightharpoonup \mathcal{R}'$ as previously. We consider the case (d) separately.

Case 1: situation (a),(b) or (c) We perform an induction on the length of the reduction $\mathcal{S} \rightsquigarrow^k \mathcal{R}$. The induction hypothesis (IH) is that we can fill the diagram in the following way

$$\begin{array}{ccc} \mathcal{R} & \rightharpoonup & \mathcal{R}' \\ {\scriptstyle k}\,\wr & & \wr\,{\scriptstyle k'} \\ \mathcal{S} & \rightharpoonup^+ & \mathcal{S}' \end{array}$$

such that $k' \leq k$, and the bottom reduction does not contain any duplication.

The base case $k = 0$ is trivially true. Now, assume $\mathcal{S} \rightsquigarrow^k \mathcal{T} \rightsquigarrow \mathcal{R}$. The upper square of the figure below is obtained by filling the diagram $\mathcal{T} \rightsquigarrow \mathcal{R} \rightharpoonup \mathcal{R}'$ as in Figure 2.5 to get the middle line $T \rightharpoonup^* T_p$. We apply the IH on each step $T_i \rightharpoonup T_{i+1}$, which we can do precisely because the IH states that the length k_i of the reduction $S_i \rightsquigarrow^{k_i} T_i$ decreases with i. Pasting all the diagrams, we get:

$$\begin{array}{ccccccc} \mathcal{R} & & \rightharpoonup & & & & \mathcal{R}' \\ \wr & & & & & & \wr| \\ \mathcal{T} & \rightharpoonup & \mathcal{T}_1 & \rightharpoonup & \ldots & \rightharpoonup & \mathcal{T}_p \\ {\scriptstyle *}\,\wr & & {\scriptstyle *}\,\wr & & & & {\scriptstyle *}\,\wr \\ \mathcal{S} & \rightharpoonup^+ & \mathcal{S}_1 & \rightharpoonup^+ & \ldots & \rightharpoonup^+ & \mathcal{S}_p \end{array}$$

Case 2: situation (d) The case of duplication is simpler as the step $\mathcal{R} \rightharpoonup \mathcal{R}'$ is reflected by just one step $\mathcal{S} \rightharpoonup \mathcal{S}'$ in the diagram of Figure 2.6. The IH is that we can fill the following diagram:

$$\begin{array}{ccc} \mathcal{R} & \rightharpoonup & \mathcal{R}' \\ {\scriptstyle k}\,\wr & & \wr\,{\scriptstyle *} \\ \mathcal{S} & \rightharpoonup & \mathcal{S}' \end{array}$$

We use the diagram of Figure 2.6 to get the upper square in the figure below. Then we apply the IH to get the bottom part $\mathcal{S}'$:

$$\begin{array}{ccc} \mathcal{R} & \multimap & \mathcal{R}' \\ \wr & & \overset{*}{\wr} \\ \mathcal{T} & \multimap & \mathcal{T}' \\ \overset{k}{\wr} & & \overset{*}{\wr} \\ \mathcal{S} & \multimap & \mathcal{S}' \end{array}$$

Once this statement is proved, we perform a second induction on the length of the reduction $\mathcal{R} \to^+ \mathcal{R}'$ to obtain the desired result. □

Writing a reduction $\mathcal{R} \to^* S$ as blocks $\mathcal{R} \rightsquigarrow^* R_1 \multimap^* R_2 \rightsquigarrow^* \ldots \multimap^* R_n$, we can iterate Proposition 2.4.7 to form a new reduction sequence with only two distinct blocks:

Proposition 2.4.8. Postponement
Let $\mathcal{R} \to^* \mathcal{S}$. Then $\mathcal{R} \multimap^* \mathcal{R}' \rightsquigarrow^* \mathcal{S}$. Moreover, if the original reduction contains at least one $\multimap$ step, then $\mathcal{R} \multimap^+ \mathcal{R}'$.

Proof. Proposition 2.4.8. By induction
We decompose the reduction $\mathcal{R} \to^* S$ as alternating blocks $\mathcal{R} \rightsquigarrow^* R_1 \multimap^* R_2 \rightsquigarrow^* \ldots \multimap^* R_n$, and iterate Proposition 2.4.7 to gather the reductions into only two distinct blocks. □

Lemma 2.4.9 states that $\rightsquigarrow$ reduction steps preserve and reflect the fact of being a $\multimap$-normal form. As $\multimap$ only acts on surface and $\rightsquigarrow$ inside boxes, the latter can neither create nor destruct redexes of the former.

Lemma 2.4.9. Neutrality of $\rightsquigarrow$
Let $\mathcal{R} \rightsquigarrow^* \mathcal{R}'$. Then $\mathcal{R}$ is $\multimap$-normal if and only if $\mathcal{R}'$ is.

Proof. Lemma 2.4.9
$\rightsquigarrow$ can not create nor erase $\multimap$-redexes. □

We state the termination properties of the two reductions $\multimap$ and $\rightsquigarrow$. $\multimap$ verifies (T_1) while $\rightsquigarrow$ is strongly normalizing.

Proposition 2.4.10. Strong normalization for $\rightsquigarrow$
$\rightsquigarrow$ is strongly normalizing.

Proof. Opening or deleting a box strictly decreases the total number of boxes in the net. □

Proposition 2.4.11. Diamond for $\multimap$
$(\mathcal{P}_N, \multimap)$ has the diamond property.

Proof. Proposition 2.4.11
This can be checked that the surfaceness and closeness constraints enforces the diamond property. The choice of taking the total lifting $\Rightarrow^{tot}$ is necessary for this proposition to be true, as well as the exclusion of the **0** rule, which would allow counter examples ("triangles") to exist. □

Corollary 2.4.12. $(\mathcal{P}_N, \rightharpoonup)$ verifies (T_1).

Proof. Corollary 2.4.12
The surface reduction satisfies the diamond property, which precludes the existence of a term with both a normal form and an infinite reduction. □

We can finally now prove Theorem 2.4.5. The fact that we prove is that $(\mathcal{P}_N, \rightharpoonup \cup \rightsquigarrow) = (\mathcal{P}_N, \Rightarrow^{tot})$ is (T_1). The last step consist in showing that $(\mathcal{P}_N, \Rightarrow)$ and $(\mathcal{P}_N, \Rightarrow^{tot})$ have the same class of weakly and strongly terminating terms.

Proof. Theorem 2.4.5
We prove two auxiliary properties:

(a) $\to$-weak normalization implies $\rightharpoonup$-weak normalization
Let $\mathcal{R}$ be $\to$-weakly normalizing and $\mathcal{R} \to^* \mathcal{N}$ a reduction to its normal form. By Proposition 2.4.8, we can write $\mathcal{R} \rightharpoonup^* \mathcal{S} \rightsquigarrow^* N$. N being a $\to$-normal form, it is also a $\rightharpoonup$-normal form, and by Lemma 2.4.9 so is $\mathcal{S}$. $\mathcal{R}$ is thus $\rightharpoonup$-weakly normalizing.

(b) an infinite $\to$-reduction gives an infinite $\rightharpoonup$-reduction
Let $\mathcal{R}$ be a net with an infinite $\to$-reduction, written $\mathcal{R} \to \infty$. We will build by induction a reduction sequence $\mathcal{R} = \mathcal{R}_0 \rightharpoonup \mathcal{R}_1 \ldots \rightharpoonup \mathcal{R}_n \rightharpoonup \ldots$ such that for any n, $\mathcal{R}_n \to \infty$.

Base case We just take $\mathcal{R}_0 = \mathcal{R}$

Inductive case If $\mathcal{R} \rightharpoonup^n \mathcal{R}_n \to \infty$, we take an infinite $\to$-reduction starting from $\mathcal{R}_n$. If the first step is $\mathcal{R}_n \rightharpoonup \mathcal{S}$, then we take $\mathcal{R}_{n+1} = \mathcal{S}$. Otherwise, the first step is a $\rightsquigarrow$ step, and we take the maximal block of $\rightsquigarrow$ reductions starting from $\mathcal{R}_n$. By Proposition 2.4.10, this block must be finite and we can write $\mathcal{R}_n \rightsquigarrow^* \mathcal{S} \rightharpoonup \mathcal{S}' \to \infty$. By Proposition 2.4.7, we can swap the two blocks such that $\mathcal{R}_n \rightharpoonup \mathcal{R}' \rightharpoonup^* \mathcal{R}'' \rightsquigarrow^* \mathcal{S}' \to \infty$, and we take $\mathcal{R}_{n+1} = \mathcal{R}'$.

From these two points, it follows that $\to$-weak normalization implies $\to$-strong normalization. If a net is $\to$-weakly normalizing, then by (a) it is $\rightharpoonup$-weakly normalizing. By Corollary 2.4.12, it is also $\rightharpoonup$-strongly normalizing. But by (b) it must be also $\to$-strongly normalizing. □

2.5 Summary

In this chapter, we gave a definition of a non-deterministic nets system, a reduction on this system, and studied the properties of this reduction.

We first introduced simple nets that are defined as graph-like structures, whose cells - the analog of vertices - corresponds to rules of LL and wires to LL formulas. The rules we use are those of the multiplicative exponential fragment of LL, enriched with two rules coming from Differential LL, cocontraction and coweakening. The differential rules enable the expression of non-determinism. The non-determinism of the reduction is then accommodated by defining nets as formal sums of simple nets, where each summand represent a different outcome of the current computation. We then proceed to study some properties of this system.

We show that, although our system handles non-determinism, the introduction of formal sums allows to have confluence. In order to prove this, we constructed a parallel reduction $\nrightarrow$ which we showed to be confluent in a very strong sense: it verifies a diamond property. Then we show that its transitive reflexive closure coincide with the one of the initially defined reduction on nets. This show the latter reduction is also confluent.

We then studied normalization in our system. Since we dropped the correctness criterion, all nets are not strongly normalizing, as illustrated by an example of a looping pattern (Figure 2.2). However we prove that in the fragment without the $\xrightarrow{\mathbf{0}}$ rule, nets that are not strongly normalizing are constrained: they can not reduce to any normal form. To prove this, we split the reduction in two: an inner reduction (which happens inside boxes) and a surface reduction. We studied these fragments independently, and how they interact. This allows us to prove that their union, which is the full reduction on nets (except the $\xrightarrow{\mathbf{0}}$ rule) verifies the property mentioned above.

2.6 Discussion

A distinctive feature of our system is that we do not require a correctness criterion. This has important consequences, as the traditional reduction of Differential LL for our fragment is neither confluent nor terminating, as demonstrated by counter examples. This motivated us to this restrict this reduction in two ways.

Reduction is close Only boxes without auxiliary ports may be subject to reduction

Reduction is surface A very limited set of rules may be performed inside boxes

The close constraint allows us to recover confluence, by a technique inspired by the standard proof of confluence for the λ-calculus. On the other hand, being close and surface is not sufficient to ensure strong normalization. This is why we rather prove the property that if we exclude the $\xrightarrow{\mathbf{0}}$ rule, then any weakly normalizing net is strongly normalizing. This ensure that as long as a net have a normal form in this $\xrightarrow{\mathbf{0}}$-free fragment, then any path of reduction will eventually lead to this normal form.

This completes the study of the abstract rewriting system of nets that has been carried in this chapter. The next chapter focuses on a specific family of nets that plays a key role in the expression of concurrency-related features.

Chapter 3

Routing Areas

Concurrency involves threads and a means of communication between them, be it a shared memory (references), messages (channels) or something else. Such a means of communication, take for example a memory cell in a shared memory, may both have multiple sources (threads writing in this cell) and multiple targets (threads reading this cell). The expected behavior in case of concurrent accesses is dictated by the operational semantic of the source language. In the case of a message channel for example, a read operation could retrieve at random one value amongst all sent messages, or rather the last message sent if channels act as a stack.

To interpret a fragment of π-calculus in nets, [14] introduces *communication areas*. For an integer n, the n-communication area is a simple net with $2n + 2$ free ports, corresponding to inputs and outputs grouped in pairs, that enables a bidirectional communication between the $n + 1$ processes - or rather, their representation as a net - that are plugged on each input/output pair. In this chapter, we introduce *routing areas*, which generalize the design of communication areas. The motivation is similar: we aim at crafting special nets to be used as building blocks for implementing communication primitives. While a communication area connects (by default) all processes, a routing area is parametrized by an algebraic object (a *multirelation*) that allows a finer control over the implemented communication scheme (cf Section 2.6 for a detailed comparison). For example, take three processes P, Q and R communicating through a channel. With a routing area, we can statically enforce various restrictions, such as banning P from sending messages to Q, or banning R from receiving messages from P, depending on the chosen multirelation. One can imagine implementing ownership constraints, in the style of the Rust programming language, where only one process - say P - is able to send messages and every other ones can only listen. Routing areas allow to connect multiple agents with a parametrizable behavior.

Overview Section 3.1 introduces routing areas. In order to do so, various technical tools are introduced first. Section 3.1.1 introduces the notion of multirelation, a quantitative extension of relations between sets, which will be used to give an algebraic description of a routing area. Section 3.1.2 defines (co)contraction trees, which can be seen as generalized n-ary (co)contractions, and derives a corresponding reduction rule. Co(contraction) trees will be the basic ingredients for building routing areas. Section 3.1.3 briefly reviews the correctness criterion which has been mentioned in Chapter 2. It will be used for various proofs of this chapter, as routing areas does satisfy the correctness criterion. Then, everything is in place to construct routing areas, which is done in Section 3.1.4.

Section 3.2 describes constructions that allow to combine simple routing areas into more complex ones. We give two fundamental operations, juxtaposition and trace, and define the composition of routing areas from these two.

Section 3.3 observe that the interpretation of routing areas as communication devices extends to a larger class of nets, *routing nets*, which include routing areas. The necessity of recasting the notion of connection between endpoints in this setting leads us to give a formal treatment of path in Section 3.3.1. This tool allows to prove in Section 3.3.2 that the normal form of a routing net is a routing area. This provides a semantics that associates the multirelation describing its normal form to a routing nets. We study the properties of this semantics in Section 3.3.3, and show that it is closely related to paths.

3.1 Routing Areas

Contraction, weakening, cocontraction and coweakening act as resource dispatchers (a resource designates a closed exponential box in the following). Structural rules are a natural choice for the basic components of routing areas.

- –
  A wire acts as the identity. It passively forwards a resource that is connected on the *input* (the left port) to the *output* (the right port).
- 

  A contraction is a broadcaster with one input and two outputs. A resource connected on the left will be copied to both outputs on the right. A weakening is a degenerate case of a broadcaster with zero outputs as broadcasting something to no one is the same as erasing it.
- 

  Dually, a cocontraction is a packer with two inputs and one output. A packer

aggregates its two inputs non deterministically. When a dereliction is connected to the output to consume two packed resources, a non-deterministic sum of the two possible choices for the resource to be provided is produced. Similarly, coweakening is seen as a degenerate packer with no inputs.

A routing area can be seen as a wiring between inputs and outputs. Inputs are connected to contractions which broadcast the resources they receive to cocontractions. Cocontractions may gather resources from multiple such sources. The conclusions of these cocontractions form the outputs.

3.1.1 Multirelations

In order to describe a routing area, we rely on a natural generalization of a relation between sets, a *multirelation*. Its role is to define the *wiring diagram*, which specifies which inputs and outputs will be connected in the routing area. A multirelation between two sets is a relation with multiplicity. Instead of just indicating if two elements x and y are related or not, a multirelation associates an integer $n \in \mathbb{N}$ to the couple (x, y). If the value at (x, y) is 0, then x and y are not in relation. Otherwise, the value $n \geq 1$ quantifies how much, or rather how many times, x and y are in relation.

Definition 3.1.1. Multirelation
Let A and B be two sets, a *multirelation* R between A and B is a map $R : A \times B \to \mathbb{N}$.

Remark 3.1.2. Relations and multirelations
A relation between A and B can be seen as a multirelation whose value at a pair (x, y) is either 0, meaning "not in relation", or 1, meaning "in relation". Formally, the map that takes a relation R and returns its characteristic function $\rho : R \mapsto \mathbb{1}_R$ is an injection from relations between A and B to multirelations between A and B. Conversely, we can forget the multiplicity of a multirelation S and simply take 0 to mean not in relation and $n \geq 1$ to mean in relation. Doing so, we recover a relation from a S through the map $\nu : S \mapsto \{(x, y) \in A \times B \mid S(x, y) \geq 1\}$. Note that ν is the left inverse of ρ : $\nu \circ \rho = id$.

From now on, we only consider multirelations between finite sets.

Definition 3.1.3. Arity
Let R be a multirelation between (finite) sets E and F. We define the arity of $e \in E$ as the total number of times it is in relation with an element of F, with multiplicity: $\mathbf{ar}(e) = \sum_{f \in F} R(e, f)$. Similarly, the arity of $f \in F$ is defined by $\mathbf{ar}(f) = \sum_{e \in E} R(e, f)$. The set of elements of F (resp. E) connected

to $e \in E$ (resp. $f \in F$) is defined by $\mathbf{co}(e) = \{f \in F \mid R(i,o) > 0\}$ (resp. $\mathbf{co}(f) = \{e \in E \mid R(e,f) > 0\}$).

Remark 3.1.4. For $x \in E \cup F$, $\mathbf{ar}(x) \geq |\mathbf{co}(x)|$. $\mathbf{ar}(x) = |\mathbf{co}(x)|$ for all $x \in E \cup F$ if and only if R is a relation.

A composition operation can be defined on multrelations, which computes all the ways to go from an element to another with multiplicities.

Definition 3.1.5. Composition of multirelations
Let R, S be multirelations respectively between A and B, and B and C, we define the multirelation $S \circ R$ by

$$(S \circ R)(x,z) = \sum_{y \in B} R(x,y)S(y,z)$$

This composition is associative, coincides with the usual one for relations, and has the identity relation (seen as a multirelation) for neutral. One can view a multirelation as a $|B| \times |A|$ matrix with integers coefficients. Lines (resp. columns) are indexed by elements of B (resp. A). For $(a,b) \in A \times B$, the coordinate $R_{b,a}$ is equals to $R(a,b)$. Under this interpretation, the composition previously defined on multirelations corresponds to the matrix multiplication.

Proposition 3.1.6. The **FMRel** category
Finite sets and multirelations between them form a category **FMRel** with the composition defined above, which contains the category **FRel** of finite sets and relations as a subcategory. **FMRel** has finite coproducts.

Proof. Proposition 3.1.6
This can be showed directly from the defintions, but under the interpretation of multirelations between finite set as finite matrices with positive integer coefficients, one realizes that **FMRel** is actually the category of integer matrices, which has direct sums as coproducts. □

Multirelations are used to describe routing areas. In order to build a routing area from this description, we need to build dispatchers (resp. packers) with an arbitrary number of outputs (resp. inputs). The notion of contraction (resp. cocontraction) tree - consisting in several contraction (resp. cocontraction) stacked together - is the natural construction for this role. We define them and give the laws governing their reduction.

3.1.2 (Co)contraction trees

Remark 3.1.7. In this whole chapter, we consider two additional atomic rules for the reduction accounting for the neutrality of (co)weakening for (co)contraction:

As discussed in the coming Section 3.1.3, all the nets considered in this chapter satisfy a correctness criterion. In this context, confluence and strong normalization of nets is ensured and remains true when one also include the $\xrightarrow{\mathtt{n}}$ rules [55, 57]. Without these rules, one would have to consider a notion of (co)contraction trees, and consequently of routing areas, with infinitely many distinct representatives. While these rules are not strictly necessary to our developments, it surely comes in handy by forcing the uniqueness of several objects introduced in the following.

A n-ary (co)contraction tree is a stack of (co)contraction defined by induction, represented in the following way:

We will sometimes omit the arity, but always represent the dots between auxiliary ports to differentiate from (co)contraction and (co)weakening cells.

Definition 3.1.8. (Co)contraction tree
Let $n \in \mathbb{N}$, we define an n-ary (co)contraction tree as a simple net with $n+1$ free ports, split between a *principal* port and *auxiliary* ports. All wires are labelled by the same formula $!A$ (or $?A^{\perp}$ depending on the orientation). (Co)contraction trees are defined by induction:

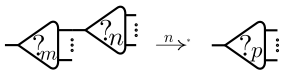
- $n = 0$: a 0-ary tree is a (co)weakening, the principal port being the principal port of this (co)weakening.

- $n = 1$: a unary tree is a wire. If it has ports p_1 and p_2, the principal port is set to be the p_j such that the label of (p_j, p_{2-j}) is $?A^{\perp}$ (resp. $!A$ in the case of cocontraction tree).

- $n > 1$: a n-ary tree for $n > 1$ is built from a $(n-1)$-ary tree by connecting one of its auxiliary port to a (co)contraction. Commutativity and associativity of (co)contractions ensure that the result does not depend on the choice of the auxiliary port.

On (co)contraction trees, we define merging and reduction, as follows:

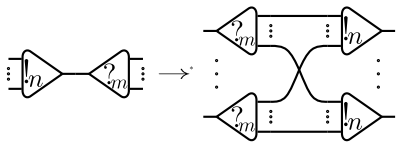
Proposition 3.1.9. Tree merging
Let $n \geq 0$ and $m \geq 1$. Connecting the main port of a (co)contraction tree of arity n to an auxiliary port of a (co)contraction tree of arity m yields a (co)contraction tree of arity $p = n + m - 1$:



Proof. By easy induction on n. □

When a contraction tree and cocontraction tree are connected through their principal port, one can derive the following generalized reduction rule:

Proposition 3.1.10. Tree reduction
Let $n \geq 0$, $m \geq 0$. We have:



Proof. Proposition 3.1.10
By double induction on n and m, using $\xrightarrow{\mathtt{s}_1}$, $\xrightarrow{\mathtt{s}_2}$, $\xrightarrow{\epsilon}$ and $\xrightarrow{\mathtt{ba}}$. □

The last tool we introduce is the correctness criterion.

3.1.3 Correctness criterion

The correctness criterion is a property of simple nets which, when verified, ensures strong normalization. In Chapter 2, we studied nets that may not satisfy this correctness criterion. Instead, the communication devices that we define in this chapter are correct nets. Let us review the correctness criterion, which we have adapted for our case of nets only composed of contraction, cocontraction, weakening and coweakening.

Definition 3.1.11. $G(R)$ and switching graphs
Let R be a simple net, we define the graph $G(R) = (V, E)$ by:

- The set of vertices V is the disjoint union of the set of cells of and the set of free ports of R.
- For each wire w of S, we add an edge $\{v_1, v_2\}$ in E where v_1 (resp. v_2) is either the cell containing $\mathbf{src}(w)$ (resp. $\mathbf{tgt}(w)$), or simply $\mathbf{src}(w)$ itself (resp. $\mathbf{tgt}(w)$) if it is a free port.

A *switching graph* $S = (V, E_s)$ is a graph obtained from $G(R)$ by removing, for each contraction, exactly one of the two wires (formally one of the two corresponding edges in E) connected to its two auxiliary ports.

Definition 3.1.12. Correctness criterion
A simple net R verifies the correctness criterion if all its switching graphs are acyclic. In this case, R is said to be *correct*.

Theorem 3.1.13. [14, Ehrhard-Regnier] Strong normalization of correct nets
A correct net is strongly normalizing.

We are finally ready to introduce routing areas.

3.1.4 Routing areas

A routing area is a simple net which is described by a multirelation between its inputs and outputs, which are a partition of its free port. The value of the multirelation at the pair (i, o) indicates how many times the input i is connected to the output o.

Definition 3.1.14. Routing area
Let $\mathcal{L}_i$ and $\mathcal{L}_o$ be two finite sets called the input labels and the output labels. Let R be a multirelation between $\mathcal{L}_i$ and $\mathcal{L}_o$. The routing area described by the triplet $(\mathcal{L}_i, \mathcal{L}_o, R)$ is a pair $(\mathbf{R}, \sigma)$ where $\mathbf{R}$ is a simple net R and σ - the labelling map - is a bijection from the set of free ports of R to $\mathcal{L}_i + \mathcal{L}_o$, partitioning the free ports between *inputs* - whose labels are in $\mathcal{L}_i$ - and outputs, whose label are in $\mathcal{L}_o$. R is constructed in the following way:

- Each input i is connected by a wire to the principal port of a contraction tree of arity $\mathbf{ar}(\sigma(i))$.
- Each output o is connected by a wire to the principal port of a cocontraction tree of arity $\mathbf{ar}(\sigma(o))$.

- Let $(i, o) \in \mathcal{L}_i \times \mathcal{L}_o$ and $n = R(i, o)$ (remember that n is an integer). Then there are exactly n distinct wires (p_i, p'_i) connecting auxiliary ports of the contraction tree of i to auxiliary ports of the cocontraction tree of o.

Remark 3.1.15. Notations
For a routing area $(\mathbf{R}, \sigma)$ described by $(\mathcal{L}_i, \mathcal{L}_o, R)$, σ is a bijection between the free ports of $\mathbf{R}$ and $\mathcal{L}_i + \mathcal{L}_o$. We will freely assume that the free ports of $\mathbf{R}$ are actually $\mathcal{L}_i + \mathcal{L}_o$, and do not bother with σ anymore. Up to isomorphism and equivalence, the net $\mathbf{R}$ is uniquely defined by its description. When unambiguous, we will abuse the notation and write $\mathbf{R} = (\mathcal{L}_i, \mathcal{L}_o, R)$ to mean that there exists σ such that $(\mathbf{R}, \sigma)$ is the routing area described by $(\mathcal{L}_i, \mathcal{L}_o, R)$.

Remark 3.1.16. Communication Areas
The communication areas defined in [14] are a special case of routing areas: for $n \leq 1$, the n-communication area is the routing area $(\{1, \ldots, n\}, \{1, \ldots, n\}, R)$ where $x \; R \; y \iff x \neq y$.

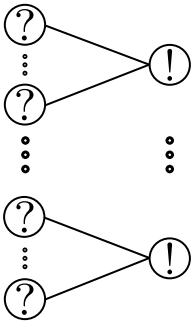
Remark 3.1.17. Canonicity of the description of a routing area
The multirelation defining an area is not unique from a set-theoretic point of view: indeed, for $\mathbf{R} = (\mathcal{L}_i, \mathcal{L}_o, R)$, then the class of multirelations describing $\mathbf{R}$ is $\{\tau^{-1} \circ R \circ \sigma : E \to F \mid \sigma : E \to \mathcal{L}_i, \tau : F \to \mathcal{L}_o, \sigma \text{ and } \tau \text{ bijective } \}$. Even though this is a proper class, all these multirelations are isomorphic (for example, in the arrow category of **FMRel**). Finite deterministic automata also suffer this kind of subtlety for example, which is not relevant in practice.

We can make the following basic observations about routing areas:

Proposition 3.1.18. Let $\mathbf{R}$ be a routing area, then:

- $\mathbf{R}$ is a normal form.
- $\mathbf{R}$ is correct.

Proof. Proposition 3.1.18
The fact that a routing area is a normal form is immediate from its shape : contraction and cocontraction trees are only connected through auxiliary ports. For correctness, observe that in switching graphs, only one branch of each contraction tree may survive. Switching graphs have the following shape:



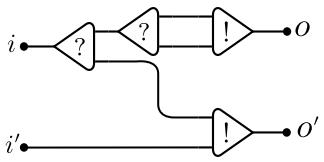
We represented trees as just one multi-ary node, and long branches corresponding to contraction trees as just one node. We see that this is a forest, a disjoint union of trees, which is acyclic. □

We represent routing areas as rectangular boxes, with the inputs appearing on the left and the outputs on the right.

Example 3.1.19. Let R be the multi relation between $\mathcal{L}_i = \{i, i'\}$ and $\mathcal{L}_o = \{o, o'\}$ defined by :

$$\begin{array}{lcl|lcl} R(i,o) & = & 2 & R(i,o') & = & 1 \\ R(i',o) & = & 0 & R(i',o') & = & 1 \end{array}$$

Then the routing area described by $(\mathcal{L}_i, \mathcal{L}_o, R)$ is:



3.2 Operations on Routing Areas

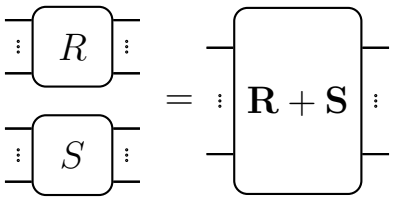
In this section, we introduce elementary operations on routing areas which allow to combine them:

1. Juxtaposition
2. Trace

These two operations allows to implement composition, in a way which is similar to the composition of Game Semantic or Geometry of Interaction whose motto is "composition = parallel composition plus hiding".

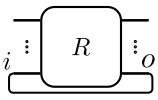
Juxtaposition The first operation, juxtaposition, amounts to put side by side two routing areas. The result is immediately seen as a routing area itself, described by the coproduct of the two multirelations:

Definition 3.2.1. Juxtaposition
Let $\mathbf{R} = (\mathcal{L}_i, \mathcal{L}_o, R)$ and $\mathbf{S} = (\mathcal{L}'_i, \mathcal{L}'_o, S)$, we define the juxtaposition $\mathbf{R} + \mathbf{S}$ by $(\mathcal{L}_i + \mathcal{L}'_i, \mathcal{L}_o + \mathcal{L}'_o, R + S)$. The corresponding net is obtained by juxtaposing the nets of $\mathbf{R}$ and $\mathbf{S}$:



Trace The second operation, trace, consists in connecting an input to an output under the condition We ask that the given input and output are not connected, to avoid the creation of a cycle. Doing so, we remove this output and input from the external interface, and create new connections between remaining inputs and outputs. The resulting net reduces to a routing area whose multirelation is defined by Equation (3.1).

Definition 3.2.2. Pretrace
Let $\mathbf{R}$ be a routing area and $(i, o) \in \mathcal{L}_i \times \mathcal{L}_o$ such that $R(i, o) = 0$. The *pretrace* of $\mathbf{R}$ at (i, o), $\mathtt{PreTr}_{(i,o)}(\mathbf{R})$, is the simple net obtained from R by connecting the input i to the output o.



Proposition 3.2.3. Let $\mathbf{R}$ be a routing area and $(i, o) \in \mathcal{L}_i \times \mathcal{L}_o$ such that $R(i, o) = 0$. Then:

- $\mathtt{PreTr}_{(i,o)}(\mathbf{R})$ satisfies the correctness criterion
- $\mathtt{PreTr}_{(i,o)}(\mathbf{R}) \to^* \mathbf{S}$ where $\mathbf{S}$ is a routing area defined by $(\mathcal{L}_i - \{i\}, \mathcal{L}_o - \{o\}, S)$ and
$$S(x, y) = S(x, y) + S(x, o)R(i, y) \tag{3.1}$$

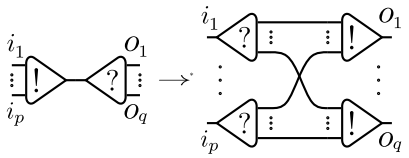
Definition 3.2.4. Trace
Let $\mathbf{R} = (\mathcal{L}_i, \mathcal{L}_o, R)$ be a routing area, and $(i, o) \in \mathcal{L}_i \times \mathcal{L}_o$ such that $R(i, o) = 0$. We define the *trace* at (i, o) of $\mathbf{R}$, $\mathtt{Tr}_{(i\to o)}(\mathbf{R})$, by the routing area obtained by reducing $\mathtt{PreTr}_{(i,o)}(\mathbf{R})$ to a normal form.

The consistence of Definition 3.2.4 is ensured by Proposition 3.2.3. As routing areas are normal forms, when one forms $\mathtt{PreTr}_{(i,o)}(\mathbf{R})$, there can only be one potential redex, created by the connection of i to o. This redex is composed of a contraction tree facing a cocontraction tree, that we know how to reduce thanks to Proposition 3.1.10. We then check that the resulting net is a routing area whose multirelation verifies the equation (Equation (3.1)).

Proof. Proposition 3.2.3
The fact that the pretrace is a correct net is easily deduced from the shape of switching graphs as forests illustrated in the proof of Proposition 3.1.18. It suffices to note that the condition $R(i, o) = 0$ entails that in any switching graph, the input i and the output o are in distinct trees. The connection of i and o then amounts to connect a leaf of one of those trees to the root of another one,

which preserves the forest structure of the switching graph. Thus the switching graphs of the pretrace at (i, o) are also ayclic.

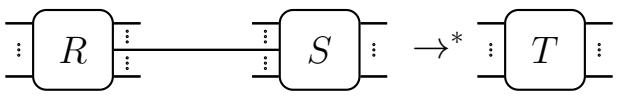
By forming the pretrace at (i, o), we connect the contraction tree of i to the cocontraction tree of o. We apply Proposition 3.1.10 on the introduced redex to get:



where the ports $i_1, \ldots, i_p$ are connected to auxiliary ports of trees of the inputs in $\mathbf{co}(o)$ and $o_1, \ldots, o_q$ to the trees of the outputs in $\mathbf{co}(i)$. By Proposition 3.1.9, we can merge all these trees with the one they are connected to such that the reduced net has the shape of a routing area $\mathbf{S} = (\mathcal{L}'_i, \mathcal{L}'_o, S)$.

To explicit the multirelation S, we have to determine the number of connections between any input $x \in \mathcal{L}'_i$ and output $y \in \mathcal{L}'_o$. The free ports of $\mathbf{S}$ are inherited from $\mathbf{R}$, excepted that i and o are now gone: thus $\mathcal{L}'_i = \mathcal{L}_i \setminus \{i\}$ and $\mathcal{L}'_o = \mathcal{L}_o \setminus \{o\}$. The direct connections between x and y in $\mathbf{R}$ when $x \neq i$ and $y \neq o$ are inherited by $\mathbf{S}$, as they are left unchanged by the reduction. But any pair of connections in $\mathbf{R}$ between x and o arriving at some i_k, and between i and y arriving at some o_l, yields exactly one new connection between x and y in $\mathbf{S}$ after the tree reduction. By definition, there are $R(x, o)$ connections between x and o and $R(i, y)$ connection between i and y : there are $R(x, o)R(i, y)$ such couples. The total number of connections between x and y is then $S(x, y) = R(x, y) + R(x, o)R(i, y)$. □

Composition Composition consists in connecting the output of an area to an input of another area and reducing the result to a routing area. Let $(\mathcal{L}_i, \mathcal{L}_o, R)$ and $(\mathcal{L}'_i, \mathcal{L}'_o, S)$ be two routing areas, $o \in \mathcal{L}_o$, $i \in \mathcal{L}'_i$, the composition of $\mathbf{R}$ and $\mathbf{S}$ at (i, o) is defined as the routing area obtained by first taking the juxtaposition of $\mathbf{R}$ and $\mathbf{S}$, and then the trace at (i, o).



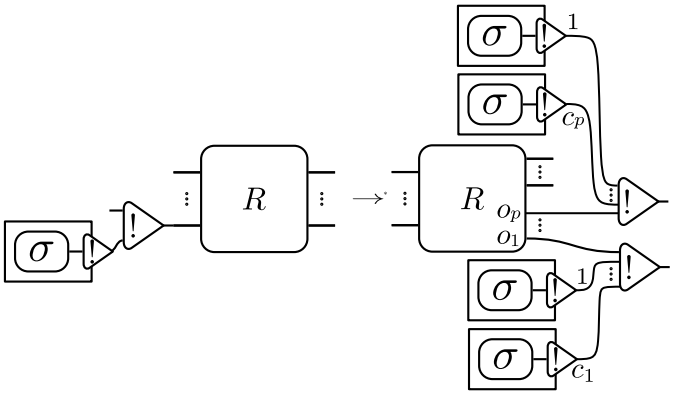
Definition 3.2.5. Composition
Let $\mathbf{R} = (\mathcal{L}_i, \mathcal{L}_o, R)$ and $\mathbf{S} = (\mathcal{L}'_i, \mathcal{L}'_o, S)$ be two routing areas, $o \in \mathcal{L}_o$ and $i \in \mathcal{L}'_i$. The composition of $\mathbf{R}$ and $\mathbf{S}$ at (i, o) is the routing area $\mathbf{R} \circ_{i \to o} \mathbf{S} = (\mathcal{L}_i + \mathcal{L}'_i \setminus \{i\}, \mathcal{L}_o + \mathcal{L}'_o \setminus \{o\}, T)$ defined by $\mathbf{R} \circ_{i \to o} \mathbf{S} = \mathtt{Tr}_{(i \to o)}(\mathbf{R} + \mathbf{S})$.

Remark 3.2.6. This operation can be generalized to the connection of n outputs of R to n inputs of S. When $n = |\mathcal{L}_o| = |\mathcal{L}'_i|$, the multirelation T describing the resulting routing area is the composed $S \circ R$.

Juxtaposition, trace and composition are fundamental features of routing areas. This is what makes them modular, allowing to build complex routing areas by connecting simple blocks.

Transit We conclude this section with a property that reflects the high level operational behavior of a routing area. It supports our interpretation of routing areas as resource dispatchers. Given a closed exponential box, we connect it to the auxiliary port of a cocontraction to obtain a module which can then be connected to an input i. Through reduction, the box will traverse the area and be duplicated $R(i, o)$ times to each output o. The role of the additional cocontraction is to preserve the area and allow future connections to the same input. We would get a similar transit property by connecting directly the exponential box to i, but this process is destructive as it would erase the input wire of i and prevents any future usage.

Proposition 3.2.7. Transit
Let σ be a closed exponential box, $\mathbf{R} = (\mathcal{L}_i, \mathcal{L}_o, R)$ a routing area, $i \in \mathcal{L}_i$. Let $\{o_1, \ldots, o_p\} = \mathbf{co}(i)$ and for $1 \leq k \leq p$, $c_k = R(i, o_k)$. Then :



Proof. Proposition 3.2.7
If we consider the added cocontraction as a 2-ary tree, we can apply Proposition 3.1.10. We aggregate them with the cocontraction trees of outputs using Proposition 3.1.9. Then we duplicate the exponential boxes now facing cocontraction tree several times using □

The last section of this chapter is dedicated to generalize the ideas and the approach developed for routing areas to a larger class of nets, *routing nets*.

3.3 Routing Nets

Routing areas are specific normal forms of nets composed of structural cells, which are contraction, cocontraction, weakening and coweakening. We saw (in Section 3.1) that these cells have an interpretation as simple communication devices. We also proposed ways of combining these simple devices into more involved ones, and that the communication interpretation - practically, the description as a multirelation between inputs and outputs - apply to these combinations. This leads us to wonder if this interpretation can be extended to more general assemblies of structural cells: can any net constituted of structural cells, not necessarily in normal form and not necessarily a combination of routing areas, be seen as a communication device and described as such by a multirelation in the same way as a routing area ? The answer given in this section is yes, as long as nets are correct. More precisely, we show that correct nets composed of structural cells, *routing nets*, reduce to a routing area. We can thus associate to any such net the multirelation describing its normal forms.

Section 3.3.1 does a formal treatment of paths in simple net, an ubiquitous concept in the section. They allow us to speak and quantify connections between free ports in the more general setting of routing nets, instead of just routing areas. We show that the correctness of routing nets exclude the existence of a cyclic path, which is a cornerstone of the proof that the normal form of a routing net is a routing area (Theorem 3.3.7). This theorem is proved in Section 3.3.2. Section 3.3.3 ends the section by exploring the consequence of the fact that routing nets reduce to routing area, which is that we can define a semantics that associate the multirelation describing its normal form to a routing net. Its properties are studied, and we give an alternative formulation in terms of paths that is independent from routing areas.

Definition 3.3.1. Routing nets
A routing net $\mathcal{R}$ is a net whose cells are weakenings, coweakenings, contractions or cocontractions that satisfies the correctness criterion. Moreover, we ask that all wires are labelled with the same formula $!A$, fixing de facto their orientation.

Note that routing area are a special case of routing nets.

3.3.1 Paths in Routing Nets

Simple nets can be seen as graphs where vertices are cells and edges are wires: this is the graph $G(R)$ introduced in Definition 3.1.11. The notion of path defined below corresponds the standard notion of a path in this graph, with the additional restriction that paths can not "bounce back" on a cell through

the principal port or through two auxiliary ports. When a path enters a node through an auxiliary port, it must exit through the principal port, and vice-versa. Similarly, it can not bounce back on a free port.

Let R be a simple net. We recall a few notations. For a cell c of $\mathcal{R}$, we write $p_i(c)$ for its i-th auxiliary port if it exists and $p(c)$ for its principal port. For a wire $w = (p, p')$, $\mathbf{src}(w) = p$ designates the source port of w while $\mathbf{tgt}(w) = p'$ is its target port. The undirected graph $G(R) = (V, E)$ is constructed as follows:

- The set of vertices V is the disjoint union of the set of cells of and the set of free ports of R.
- For each wire w of S, we add an edge (v_1, v_2) where v_1 (resp. v_2) is either the cell containing $\mathbf{src}(w)$ (resp. $\mathbf{tgt}(w)$), or $\mathbf{src}(w)$ (resp. $\mathbf{tgt}(w)$) directly if it is free.

We remind that cells, ports, wires, *etc.* are defined in Definition 2.1.2.

Definition 3.3.2. Non bouncing path
Let R be a simple net. Let π be a finite sequence $(v_1, e_1, v_2, e_2, \ldots, v_n, e_n, v_{n+1})$ where v_i is a vertice of $G(R)$ and e_i is an edge between v_i and v_{i+1}. We can associate to π a corresponding sequence $(w_1, \ldots, w_n)$ of wires of R oriented such that $\mathbf{src}(w_i) = v_i$ and $\mathbf{tgt}(w_i) = v_{i+1}$. A *non bouncing path*, or just *path* in the following, is such a sequence π such that for $i < n$, $\mathbf{tgt}(w_i)$ and $\mathbf{src}(w_{i+1})$ must not be either:

- Both auxiliary ports of the same cell.
- Both principal ports of the same cell.
- Both free ports.

We define $\mathbf{src}(\pi) = v_1$ and $\mathbf{tgt}(\pi) = v_{n+1}$. For two free ports p_1 and p_2, $\mathtt{Paths}_{p_1 \to p_2}(\mathcal{R}) = \{\pi \mid \pi \text{ path }, \ \mathbf{src}(\pi) = p_1, \ \mathbf{tgt}(\pi) = p_2\}$ is the set of paths starting at p_1 and ending at p_2.

There is a relation between correctness and the paths we have defined. We show that a routing net, because it is correct, does not contain cyclic paths.

Proposition 3.3.3. Acyclicity of correct nets
A routing net is acyclic, that is there is no path π such that $\mathbf{src}(p) = \mathbf{tgt}(p)$.

The proof requires the following lemma:

Lemma 3.3.4. Path invariant
Let R be a simple net, $G(R) = (V, E)$ be the associated graph, $\sigma = (v_1, e, v_2)$ a path, $w = (p_1, p_2)$ the corresponding wire in R and F its label. We define the *polarity* $\mathbf{pol}(\sigma)$ by:

- If $F = ?A$, $\mathbf{pol}(\sigma) = -$.
- If $F = !A$, $\mathbf{pol}(\sigma) = +$.

Then, for a path $\pi = (v_1, e_1, \ldots, e_n, v_{n+1})$, the polarity is constant along π, meaning that for $1 \leq i, j \leq n$, $\mathbf{pol}((v_i, e_i, v_{i+1})) = \mathbf{pol}((v_j, e_j, v_{j+1}))$.

Proof. Lemma 3.3.4
The non bouncing condition enforces that two consecutive subpaths (v_i, e_i, v_{i+1}) and $(v_{i+1}, e_{i+1}, v_{i+2})$ must have the same polarity. We conclude by induction on the length of the path. □

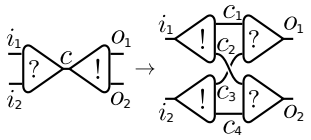
Proof. Proposition 3.3.3
We prove that the existence of a cycle implies the existence of a cycle in a switching graph. More precisely, we show that a minimal cycle never visits the two auxiliary ports of a contraction and that it is thus contained in some switching graph.

Assume the existence of a cycle and select one of minimal length, σ. We will rather work on the associated sequence of oriented wires $(w_1, \ldots, w_n)$. Assume that the two wires connected to the auxiliary ports of some contraction c, f and f', are both visited by σ. There must be two indices i_0 and i_1 such that $w_{i_0} = f$ and $w_{i_1} = f'$. We can assume $i_0 < i_1$, such that $\sigma = (w_1, \ldots, w_{i_0-1}, f, w_{i_0+1}, \ldots, w_{i_1-1}, f', w_{i_1+1}, \ldots, w_n)$. By the invariance of polarity along a path (Lemma 3.3.4), f and f' are visited in the same direction, either both exiting c or both entering c. Moreover, they can not be consecutive in σ by definition of a non bouncing path. Let us extract from the subpath $(f, w_{i_0+1}, \ldots, w_{i_1-1}, f')$ a cycle strictly smaller than σ. If both f and f' are exiting c, then the path must enter c by the principal port, thus w_{i_1-1} must correspond to the wire that contains the principal port of c. But then $(f, w_{i_0+1}, \ldots, w_{i_1-1})$ is a cycle strictly smaller than σ. Similarly, if both are entering c, then $(w_{i_0+1}, \ldots, f')$ is a cycle strictly smaller than σ. Hence, a minimal cycle σ does not visit both auxiliary ports of a contraction cell, and is contained in at least one switching graph. □

We now give an important property relating reduction to paths. It states that paths that are *long enough* - here paths that start and end at a free port - are preserved by the reduction.

Proposition 3.3.5. Path preservation
Let R be a routing net, p_1 and p_2 be free ports, and assume $R \to R'$. Then $\texttt{Paths}_{p_1 \to p_2}(R)$ and $\texttt{Paths}_{p_1 \to p_2}(R')$ are in bijection, which we write $\texttt{Paths}_{p_1 \to p_2}(R) \simeq \texttt{Paths}_{p_1 \to p_2}(R')$.

Proof. Proposition 3.3.5
For a path π, the principal port p of a weakening or a coweakening constitute a dead end: if π visits p, p must be either the source or the target of π. As a path in $\texttt{Paths}_{p_1 \to p_2}(R)$ starts and ends at a free port, it can not visit a (co)weakening. The reductions implying (co)weakenings thus leave such paths unchanged. The only reduction affecting paths in $\texttt{Paths}_{p_1 \to p_2}(R)$ is the $\xrightarrow{\texttt{ba}}$ rule:



Let us build the bijection $\tau : \texttt{Paths}_{p_1 \to p_2}(R) \to \texttt{Paths}_{p_2 \to p_2}(R')$. Let $\pi \in \texttt{Paths}_{p_1 \to p_2}(R)$ be a path that does not visit the contraction or the cocontraction, it is left unchanged and we set $\tau(\pi) = \pi$. Otherwise, we replace any subsequence of π appearing in the left column by the corresponding one in the right column:

Subsequence	Image by τ
i_1, c, o_1	i_1, c_1, o_1
i_1, c, o_2	i_1, c_2, o_2
i_2, c, o_1	i_2, c_3, o_1
i_2, c, o_2	i_2, c_4, o_2

We omitted the four dual possibilities which can be deduced from this table by reversing both the subsequence and its image. It is easily seen that τ is bijective. □

We can now reformulate the link between a routing area $\mathbf{R}$ and its defining relation R in terms of paths: $R(i, o)$ is the number of distinct paths linking i to o.

Proposition 3.3.6. Paths in routing areas
Let $\mathbf{R} = (\mathcal{L}_i, \mathcal{L}_o, R)$ be a routing area and $(i, o) \in \mathcal{L}_i \times \mathcal{L}_o$. Then $R(i, o) = |\texttt{Paths}_{i \to o}(\mathbf{R})|$.

Paths allow to generalize the notion of an input and an output being connected in a routing area. We can replace it by the existence of a path between any two free ports in a routing net. This allows us to prove in the next subsection that routing nets are actually, up to normalization, routing areas.

3.3.2 Normal forms

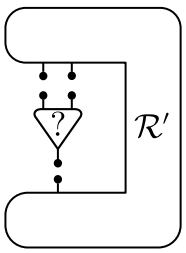
In this section, we show that routing areas are the normal forms of routing nets. That means that an arbitrary routing net always reduces to a routing area. Observe that the basic components of routing nets, (co)contractions, wires and

(co)weakenings are all routing areas. From there, we show that we can combine them into an arbitrary routing net using pretrace and juxtaposition. Thanks to Proposition 3.2.3, this implies that the resulting net reduces to a routing area that can be expressed as trace and juxtaposition of basic cells. One of the key ingredient of the proof is to justify that the trace operations are legal, as $\mathtt{Tr}_{(i \to o)}(.)$ is defined only when i and o are not connected: this is where we make use of paths, which enable to speak about connection formally.

Theorem 3.3.7. Routing area characterization
The normal form of a routing net S is a routing area $\mathbf{R} = (\mathcal{L}_i, \mathcal{L}_o, R)$.

Proof. Theorem 3.3.7
We prove the result by induction on the number n of cells of R.

- $(n = 0)$ R is only composed of free ports and wires. We take $\mathcal{L}_i = \{\mathbf{src}(w) \mid w \text{ wire } \}$, $\mathcal{L}_o = \{\mathbf{tgt}(w) \mid w \text{ wire } \}$ and R is the relation defined by $i \; R \; o \iff \exists w, \; i = \mathbf{src}(w), o = \mathbf{tgt}(w)$.
- (induction step) Let take any cell c of R. We call R' the subnet obtained by removing c from the set of cells of R.



  By induction, R' can be reduced to a routing area $\mathbf{R}'$. Then, depending on the nature of c:

  (co)weakening If c is a weakening or a coweakening, it is a routing area and can be composed with $\mathbf{R}'$, and reduced to a new routing area $\mathbf{R}$ according to Definition 3.2.5. Thus the whole net reduces to $\mathbf{R}$.

  (co)contraction If c is a contraction or a cocontraction, it is also a routing area and we juxtapose it to $\mathbf{R}'$. If we then perform thee traces operations to reconnect the ports of c one by one, we obtain a routing area which is a reduct of the original net R. What we need to check is that trace operations are legal, as they are only defined when the input and the output at which we perform the operation at are not connected - or, put differently, that there is no path between them - in the corresponding intermediate routing areas. The first operation is always legal, as it is

actually a composition of disjoint areas. For the remaining two, if a path existed between two ports in an intermediate routing area, this path already existed in R by Proposition 3.3.5. But this would result in a cycle in R, which is excluded by Proposition 3.3.3. Thus the trace operations are legal.

□

We can thus associate to a routing net the routing area that is its normal form. This area being described by a multirelation, we derive from our study of routing areas a semantics for routing nets.

3.3.3 The Routing Semantics

For a routing net S, we define the application $[\![.]\!] : S \mapsto R$ that maps S to the multirelation R describing its normal form. By unicity of normal forms, the application is invariant by reduction and is thus a semantics for routing nets, with the following properties:

Sound The multirelation only depends on the normal form. As the reduction is confluent, the normal form of a routing net is unique, and $[\![.]\!]$ is invariant by reduction.

Adequate Two routing nets with the same denotation have the same normal form, as a multirelation defines a routing area uniquely up to isomorphism.

Fully complete Any multirelation on finite sets is realised by the associated routing area.

Compositionnal We can compute the semantics of a net in a compositional way from the semantics of its smaller parts, and combine them through juxtaposition, trace and composition.

Proposition 3.3.6 formalizes the idea that the multirelation describing a routing area is determined by paths between free ports: the value on a pair (i, o) is the number of paths between them. This interpretation extends to routing nets: for a routing net R, its normal form have the same free ports as S. They can be seen as inputs and outputs, and $[\![R]\!]$ can be computed by counting the paths between the free ports directly in R:

Theorem 3.3.8. Path semantic
Let S be a routing net. Let $\mathcal{L}_i$ be the set of free ports of S which are the source of a wire, and $\mathcal{L}_o$ the ones that are the target. For $(i, o) \in \mathcal{L}_i \times \mathcal{L}_o$, we write

$N(i,o) = |\mathtt{Paths}_{i\to o}(S)|$ the number of paths between i and o. Then $[\![S]\!]$ is the multirelation between $\mathcal{L}_i$ and $\mathcal{L}_o$ given by

$$[\![S]\!](i,o) = N(i,o)$$

Proof. Theorem 3.3.8

- If S is a normal forms, then it is a routing area, and this was already noted for routing areas in Proposition 3.3.6.
- Otherwise, by path preservation (Proposition 3.3.5), $N(i,o)$ is invariant by reduction. We conclude by induction on the length of a reduction of S to its normal form.

□

Taking a step back, the routing semantics relies on the fundamental observation that specific sets of paths are preserved by reduction and totally determines the normal form of a net. The equivalence we set on simple nets, namely associativity and commutativity of (co)contraction, makes two paths from a free port to another one indistinguishable in the normal forms. This is why we can ignore the structure of these paths and simply count them.

3.4 Summary

This chapter introduced and studied routing areas, which are simple nets designed to implement communication primitives. Routing areas are inspired by communication areas introduced in [14]. We show that routing areas are *modular*, as they can be composed in different ways such that the result still reduce to a routing area. The chapter ends by showing that routing areas are actually the language of normal forms of more general nets, routing nets. Any correct net, composed of contraction, cocontraction, weakening and coweakening is in fact a routing area up to normalization. From this result, we derive a semantics for routing nets, the routing semantics. It associates to any routing net a multirelation describing it. This semantics can be defined solely in terms of paths: computing the denotation of a routing nets amount to count specific paths in this net.

3.5 Discussion

Communication areas vs routing areas For an integer n, the n-communication area is a simple net with $2n+2$ free ports, corresponding to inputs and outputs

grouped in pairs. The n-communication area enables a bidirectional and indiscriminate communication between the n processes which plugged in. For a given n, the associated communication area is unique, and allow any process to communicate with any other one. If we connect several process by a communication area, then each one of them is able to send messages to or receive messages from the others indiscriminately. On the other hand, there exists many routing areas that connect n processes: thanks to the parametrization by a multirelation, we can configure the read/write permissions on process by process basis. For example, we can require that sending messages is forbidden for all processes but one, or rather give the send/receive capabilities on a process-by-process basis. This gives the ability to statically encode various communication scheme of concurrents languages.

This capacity is put to use in Chapter 5, where we propose a translation of a concurrent calculus inside nets. We give an adequacy theorem that formalizes the fact that the translation of a term can not produce "more result" that the original term. If a term either reduces to 1 or 2, then adequacy guarantee that its translation will not reduces to (the translation of) 3 for example. Adequacy relies on the specialization of routing areas to match the calculus semantics: if we used communication areas instead, due to them being more permissive, the translation in nets would be able to compute more than the initial term.

Modularity An important feature of routing areas is their modularity. They can be assembled and connected, as the electric component of a circuit, in various ways such that the result is still a routing area, up to normalization. Modularity is illustrated by the operations defined on them, juxtaposition, trace and composition. This allows to build compositional translations without requiring the knowledge of a whole source program at once to perform the transformation. In other words, this gives routing areas the capability of supporting *separate compilation*.

Paths in routing nets The chapter ends by showing that routing areas are actually the language of normal forms of more general nets, routing nets. Any correct net, composed of contraction, cocontraction, weakening and coweakening is a routing area up to normalization. The proof of this fact exposes that routing nets are somehow the "free structure" generated by structural cells as constants and juxtaposition and pretrace as operations. This instills the idea that an arbitrary piece of proof composed only of structural rules is all about *wiring*. This is reflected by the fact that the routing semantics can be defined directly in terms of paths. Paths are known to be a central notion in nets - and λ-calculus

- both dynamically and semantically. They unifies Levy's optimal reduction, Lamping's graph-reduction algorithm and Girard's Geometry of Interaction [7]. While multirelations and routing areas are designed with practical goal in minds, we have seen paths naturally emerging as one tries to extend these ideas to routing nets.

We are now armed to encode various communication primitives inside proof nets. This is illustrated in Chapter 5 by the translation of the λ_{cES} calculus, described in Chapter 4.

Chapter 4

The concurrent λ-calculus with explicit substitutions $\lambda_{\mathbf{cES}}$

The λ-calculus is a versatile framework in the study and design of higher-order functional programming languages. One of the reasons of its widespread usage is the fact that it can easily be extended to model various computational side-effects. Another reason comes from its theoretical ground and the fine granularity it allows in the design of abstract machines to express various reduction strategies. These abstract-machines can then serve as foundation for the design of efficient interpreters and compilers.

A specially useful tool in the design of such abstract machines is the notion of explicit substitution, a refinement over β-reduction. The β-reduction of the λ-calculus is a meta-rule where substitution is defined inductively and performed all at once on the term. But its implementation is a whole different story: to avoid size explosion in presence of duplication, mechanisms such as sharing are usually deployed. Abstract machines implement various specific strategies that may either be representable in pure λ-calculus (call-by-value or call-by-name) or for which the syntax needs to be augmented with new objects (e.g. call-by-need or linear head reduction). The mismatch between β-reduction and actual implementations can make the proof of soundness for an evaluator or a compiler a highly nontrivial task. The heart of the theory of explicit substitutions, introduced in [1], is to give substitutions a first class status as objects of the syntax to better understand the dynamics and implementation of β-reduction. It consists in decomposing a substitution into explicit atomic steps. The main ingredient is to modify the β rule so that $(\lambda x.M)N$ reduces to $M[N/x]$, where $[N/x]$ is now part of the syntax. Additional reduction rules are then provided to propagate the substitution $[N/x]$ to atoms.

Studied for the last thirty years [1, 2, 3, 5, 4, 17, 32, 38, 50, 52], explicit substitution turns out to be a crucial device when transitioning from a formal higher-order calculus to a concrete implementation. It has been considered

in the context of sharing of mutual recursive definitions [50], higher-order unification [38], algebraic data-types [17], efficient abstract machines [2, 52], cost-model analysis [5], *etc.* The use of explicit substitutions however comes at a price [32]. Calculi with such a feature are sensitive to the definition of reduction rules. If one is too liberal in how substitutions can be composed then a strongly normalizing λ-term may diverge in a calculus with explicit substitutions [45]. If one is too restrictive, confluence on metaterms is lost [9]. The challenge is to carefully design the language to implement desirable features without losing fundamental properties. Several solutions have been proposed to fix these defects [4, 32] for explicit substitutions of term variables.

This chapter introduces an extension of explicit substitutions to a novel case: a lambda-calculus augmented with concurrency and references. Such a calculus forms a natural model for shared memory and message passing. We aim at proving that a translation of a shared memory model to a message passing one is sound.

Strong Normalization in a Concurrent Calculus with References A concurrent lambda-calculus with references – referred as λ_{C} below – has been introduced by Amadio in [6]. It is a call-by-value λ-calculus extended with:

- a notion of threads and an operator $\parallel$ for parallel composition of threads,
- two terms $\mathrm{set}(r, V)$ and $\mathrm{get}(r)$, to respectively assign a value to and read from a reference,
- special threads $r \Leftarrow V$, called stores, accounting for assignments.

When $\mathrm{set}(r, V)$ is reduced, it turns to the unit value $*$ and produces a store $r \Leftarrow V$ making the value available to all the other threads. A corresponding construct $\mathrm{get}(r)$ is reduced by choosing non deterministically a value among all the available stores. For example, assuming some support for basic arithmetic consider the program $(\lambda x.x + 1)\ \mathrm{get}(r) \parallel \mathrm{set}(r, 0) \parallel \mathrm{set}(r, 1)$. It consists of 3 threads: two concurrent assignments $\mathrm{set}(r, 0)$ and $\mathrm{set}(r, 1)$, and an application $(\lambda x.x + 1)\,\mathrm{get}(r)$. This programs admits two normal forms depending on which assignment "wins": the term $1 \parallel * \parallel * \parallel r \Leftarrow 0 \parallel r \Leftarrow 1$ and the term $2 \parallel * \parallel * \parallel r \Leftarrow 0 \parallel r \Leftarrow 1$. Despite the $\parallel$ operator being a static constructor, it can be embedded in abstractions and thus dynamically liberated or duplicated. For example, the term $(\lambda f.f * \parallel f *)$ act like a *fork* operation: if applied to M, it generates two copies of its argument in two parallel threads $M * \parallel M *$. Coupled with the Landin's fixpoint introduced below one can even write a fork

bomb, that is a non terminating term which spans an unbounded number of threads.

In this language, the stores are global and cumulative: their scope is the whole program, and each assignment adds a new binding that does not erase the previous one. Reading from a store is a non deterministic process that chooses a value among the available ones. References are able to handle an unlimited number of values and are understood as a typed abstraction of possibly several concrete memory cells. This feature allows λ_{C} to simulate various other calculi with references such as variants with dynamic references or communication [41].

While a simple type system for usual λ-calculus ensures termination, the situation is quite different in a language with higher-order references. The so called Landin's trick [37] allows to encode a fixpoint in the simply typed version of a calculus with references. The problem lies in the fact that one can store in a reference r values that can themselves read from the reference r, leading to a circularity. For example, the term $\mathrm{get}(r) * \parallel r \Leftarrow (\lambda x.\mathrm{get}(r) *)$ loops while involving only simple types Unit and Unit $\to$ Unit.

In order to address this issue, type and effects systems have been introduced to track the potential effects produced by a term during its evaluation. Together with stratification on references [8], one can recast termination in such an imperative context. Intuitively, stratification imposes an order between references: a reference can only store terms that access smaller ones, ruling out Landin's fixpoint. Formally, this allows to apply the usual reducibility argument to a calculus with references: stratification ensures that the inductive definition of reducibility sets on types with effects is well-founded.

While scheduling is explicitly handled through language constructs in [8], λ_{C}'s liberal reduction allows to chose a different thread to operate on at any time. This cause additional difficulty, as from a single thread's point of view, arbitrary new assignments may become available between two reduction steps. For λ_{C}, the proof of termination in [6] resorts to what amounts to infinite terms with the notion of saturated stores.

In this chapter, we introduce a concurrent λ-calculus with explicit substitution, based on the λ_{C} calculus. We provide a stratified type and effect system for this calculus, and show that typing ensures strong normalization.

Overview In Section 4.1, we introduce the concurrent λ-calculus with explicit substitutions λ_{cES}, with both substitutions for variable and references. The syntax of terms is spelled out in Section 4.1.1, while the reduction is given in Section 4.1.2. In Section 4.1.4, we define a partial preorder on terms, which will be used to compare the behaviors and the normalization of terms in the

following sections.

In Section 4.2, we introduce a *stratified type and effect system* for λ_{cES}, adapted from the one of λ_{C}. We show basic properties of the typed fragment, such as subject reduction and progress, in Section 4.2.2.

In Section 4.3, we prove an important result of this chapter, which is that well typed terms are strongly normalizing. To do so, we adapt a reducibility proof to our setting by incorporating an interactive properties. In Section 4.3.1, we lay out some necessary technical definitions, such as sets of strongly computable terms and our interactive condition of being a *well-behaved term*. We give a high level explanation of the proof in Section 4.3.2, and detail two representative cases. In Section 4.3.3, we give a more detailed proof of the strong normalization theorem and other intermediate results.

At last, Section 4.4 focuses on the relation between λ_{cES} and λ_{C}. While λ_{cES} should be thought of as a version of λ_{C} with explicit substitutions, the latter is not a proper sublanguage of the former. We can however define a translation, or rather an embedding, of λ_{C} in λ_{cES}. We prove a simulation for this embedding in Section 4.4.2. We explain why a simulation result in a non-deterministic setting is only half satisfying when one ought to relate the computational behaviors of two languages. In Section 4.4.3, we give an adequacy theorem, which is complementary to the simulation result. It states that the values than can be computed by a term of λ_{C} and the ones that can be computed by its translation in λ_{cES} are essentially the same.

4.1 A Concurrent λ-calculus with Explicit Substitutions

In standard presentations of the lambda-calculus and its extensions such as [6], substitutions are applied globally. This hides the implementation details of the procedure. Exposing such an implementation is one of the reasons for the introduction of explicit substitutions. In the literature, explicit substitutions have only been used for term variables and not for references.

In this section, we introduce the language λ_{cES}, a call-by-value, concurrent λ-calculus with explicit substitutions for both term variables and references.

4.1.1 Syntax

The language λ_{cES} has two kinds of variables: term variables (simply named *variables*) represented with $x, y, \ldots$, and *references*, represented with $r, r', \ldots$. Substitutions are represented by partial functions with finite support. Variable substitutions, denoted with Greek letters $\sigma, \tau, \ldots$, map variables to values.

Reference substitutions, denoted with calligraphic uppercase letters $\mathcal{V}, \mathcal{U}, \ldots$, map references to *finite multisets* of values. Multisets reflect the non-determinism, as multiple writes may have been performed on the same reference. They are represented with the symbol $\mathcal{E}$. The language consists of values, terms and sums of terms, representing non-determinism.

$$\begin{array}{llll} \text{-values} & V & ::= & x \mid * \mid \lambda x.M \\ \text{-terms} & M & ::= & V \mid M[\sigma] \mid (M\,M)[\mathcal{V}]_\lambda \mid \mathrm{get}(r) \mid M[\mathcal{V}]_\downarrow \mid M[\mathcal{V}]_\uparrow \mid M \parallel M \\ \text{-sums} & \mathbf{M} & ::= & \mathbf{0} \mid M \mid \mathbf{M} + \mathbf{M} \end{array}$$

The construct $M[\sigma]$ stands for the explicit substitutions of variables in M under the substitution σ. There are three constructs for explicit substitutions for references: $M[\mathcal{V}]_\downarrow$ and $M[\mathcal{V}]_\uparrow$ are respectively the downward and upward references substitutions, while $(M\,M)[\mathcal{V}]_\lambda$ is the λ-substitution. The reason for which the language needs three distinct notations is explained in the next section while presenting the reduction rules. Finally, the role of the sum-terms $\mathbf{M}$ is to capture and keep all non-deterministic behaviours.

Terms are considered modulo an equivalence relation presented in Table 4.1. The sum is idempotent, associative and commutative, while the parallel composition is associative and commutative.

Definition 4.1.1. Free variables
The set of free variables $\mathrm{FV}(M)$ of a term M is defined as follow:

- $\mathrm{FV}(x) = \{x\}$
- $\mathrm{FV}(*) = \emptyset$
- $\mathrm{FV}(\lambda x.M) = \mathrm{FV}(M) \setminus \{x\}$
- $\mathrm{FV}(M[\sigma]) = \mathrm{FV}(M) \setminus \mathrm{dom}(\sigma) \cup \mathrm{FV}(\sigma)$
- $\mathrm{FV}(M\ N[\mathcal{V}]_\lambda) = \mathrm{FV}(M) \cup \mathrm{FV}(N) \cup \mathrm{FV}(\mathcal{V})$
- $\mathrm{FV}(\mathrm{get}(r)) = \emptyset$
- $\mathrm{FV}(M[\mathcal{V}]_\downarrow) = \mathrm{FV}(M[\mathcal{V}]_\uparrow) = \mathrm{FV}(M) \cup \mathrm{FV}(\mathcal{V})$
- $\mathrm{FV}(M \parallel N) = \mathrm{FV}(M) \cup \mathrm{FV}(N)$
- $\mathrm{FV}(\sigma) = \bigcup_{x \in \mathrm{dom}(\sigma)} \mathrm{FV}(\sigma(x))$
- $\mathrm{FV}(\mathcal{V}) = \bigcup_{r \in \mathrm{dom}(\mathcal{V})} \bigcup_{V \in \mathcal{V}(r)} \mathrm{FV}(V)$

A *closed term* is a term M such that $\mathrm{FV}(M) = \emptyset$.

$$
\begin{array}{rcl}
M \parallel M' & = & M' \parallel M \\
(M \parallel M') \parallel M'' & = & M \parallel (M' \parallel M'') \\
\mathbf{M} + \mathbf{M}' & = & \mathbf{M}' + \mathbf{M} \\
(\mathbf{M} + \mathbf{M}') + \mathbf{M}'' & = & \mathbf{M} + (\mathbf{M}' + \mathbf{M}'') \\
\mathbf{M} + \mathbf{M} = \mathbf{M} + \mathbf{0} & = & \mathbf{M}
\end{array}
$$

Table 4.1: Structural Rules

$$
\begin{array}{rcl}
E & ::= & [.] \mid (E\ M)[\mathcal{V}]_\lambda \mid (M\ E)[\mathcal{V}]_\lambda \mid \\
 & & E[\mathcal{V}]_\downarrow \mid E[\mathcal{V}]_\uparrow \\
C & ::= & [.] \mid (C \parallel M) \mid (M \parallel C) \\
\mathbf{S} & ::= & [.] \mid \mathbf{S} + \mathbf{M} \mid \mathbf{M} + \mathbf{S}
\end{array}
$$

Table 4.2: Evaluation Contexts

Remark 4.1.2. The three constructs $[\mathcal{V}]_\downarrow$, $[\mathcal{V}]_\uparrow$ and $[\mathcal{V}]_\lambda$ encapsulate the assignments in terms and in stores presented in the introduction of Chapter 4: there is no need anymore for $\mathrm{set}(r, V)$ and $r \Leftarrow V$. See Section 4.6 for a discussion of this aspect.

Notation 4.1.3. Reference substitutions will be sometimes written with the notation $[r \Leftarrow V]$ to mean $[\mathcal{V}]$ with $\mathcal{V} : r \mapsto [V]$. Explicit variables substitutions are written $M[\{x_1 \mapsto V_1, \ldots, x_n \mapsto V_n\}]$. Finally, by abuse of notation we write $(M\ N)$ for $(M\ N)[\bot]$, where $\bot$ is the nowhere defined function.

4.1.2 Reduction

We adopt a weak call-by-value reduction where the reduction order of an application is not specified. It is weak in the sense that no reduction occurs under abstractions.

Although in a general setting non-determinism and call-by-value taken together may break confluence even when collecting all possible outcomes [12], this phenomenon does not happen here. Indeed, we cannot reduce under abstractions, and the only non-deterministic construct $\mathrm{get}(r)$ must be reduced before being duplicated, avoiding problematic interactions between β-reduction and non-deterministic choice.

The language λ_{cES} is equipped with the reduction defined in Table 4.3. The rules presented are closed under the structural rules of Table 4.1. We assume the usual conventions on alpha-equivalence of term, and as customary substitutions are considered modulo this alpha-equivalence. They make use of several notations that we lay out below. Rules devoted to dispatching substitutions are referred as *structural rules*. The *variable* (resp. *downward*, *upward*) *structural rules* consist in (`subst`) (resp. (`subst-r`), (`subst-r'`)) rules excluding ($\mathtt{subst}_{\mathtt{var}}$) (resp. ($\mathtt{subst\text{-}r}_{\mathtt{get}}$), ($\mathtt{subst\text{-}r}_\top$)). An in-depth discussion about these rules follows.

Notation 4.1.4. the contexts E, C and $\mathbf{S}$ are defined in Table 4.2. The context E stands for a usual call-by-value applicative context, C picks a thread, while

(a) β-reduction

(β_v)	$((\lambda x.M)\,V)[\mathcal{U}]_\lambda$	$\to$	$(M[\{x \mapsto V\}])[\mathcal{U}]_\downarrow$

If $M \to M'$ with rule (β_v), then $\mathbf{S}[C[E[M]]] \to \mathbf{S}[C[E[M']]]$

(b) Variable Substitutions

$(\mathtt{subst}_{\mathtt{var}})$	$x[\sigma]$	$\to$	$\sigma(x)$ if defined, or x otherwise
$(\mathtt{subst}_{\mathtt{unit}})$	$*[\sigma]$	$\to$	$*$
$(\mathtt{subst}_{\mathtt{app}})$	$(M\,N)[\mathcal{V}]_\lambda[\sigma]$	$\to$	$((M[\sigma])\,(N[\sigma]))[\mathcal{V}[\sigma]]_\lambda$
$(\mathtt{subst}_{\lambda})$	$(\lambda y.M)[\sigma]$	$\to$	$\lambda y.(M[\sigma])$
$(\mathtt{subst}_{\mathtt{get}})$	$\mathrm{get}(r)[\sigma]$	$\to$	$\mathrm{get}(r)$
$(\mathtt{subst}_{\parallel})$	$(M \parallel M')[\sigma]$	$\to$	$(M[\sigma]) \parallel (M'[\sigma])$
$(\mathtt{subst}_{\mathtt{subst\text{-}r}})$	$(M[\mathcal{V}]_\downarrow)[\sigma]$	$\to$	$M[\sigma][\mathcal{V}\{\sigma\}]_\downarrow$
$(\mathtt{subst}_{\mathtt{subst\text{-}r'}})$	$(M[\mathcal{V}]_\uparrow)[\sigma]$	$\to$	$M[\sigma][\mathcal{V}\{\sigma\}]_\uparrow$
$(\mathtt{subst}_{\mathtt{merge}})$	$M[\sigma][\tau]$	$\to$	$M[\sigma,\tau]$

Congruence case:
If $M \to M'$ by any of the previous rules, then $\mathbf{S}[C[E[M]] \to \mathbf{S}[C[E[M']]]$

(c) Downward Reference Substitutions

$(\mathtt{subst\text{-}r}_{\mathtt{val}})$	$V[\mathcal{V}]_\downarrow$	$\to$	V
$(\mathtt{subst\text{-}r}_{\parallel})$	$(M \parallel M')[\mathcal{V}]_\downarrow$	$\to$	$(M[\mathcal{V}]_\downarrow) \parallel (M'[\mathcal{V}]_\downarrow)$
$(\mathtt{subst\text{-}r}_{\mathtt{subst\text{-}r'}})$	$M[\mathcal{U}]_\uparrow[\mathcal{V}]_\downarrow$	$\to$	$M[\mathcal{V}]_\downarrow[\mathcal{U}]_\uparrow$
$(\mathtt{subst\text{-}r}_{\mathtt{merge}})$	$M[\mathcal{U}]_\downarrow[\mathcal{V}]_\downarrow$	$\to$	$M[\mathcal{U},\mathcal{V}]_\downarrow$
$(\mathtt{subst\text{-}r}_{\mathtt{app}})$	$(M\,N)[\mathcal{U}]_\lambda[\mathcal{V}]_\downarrow$	$\to$	$(M[\mathcal{V}]_\downarrow)\,(N[\mathcal{V}]_\downarrow)[\mathcal{U},\mathcal{V}]_\lambda$

Congruence cases:
If $M \to M'$ by any of the previous rules, then $\mathbf{S}[C[E[M]] \to \mathbf{S}[C[E[M']]]$

Finally:

$(\mathtt{subst\text{-}r}_{\mathtt{get}})$	$\mathbf{S}[C[E[\mathrm{get}(r)[\mathcal{V}]_\downarrow]]]$	$\to$	$\mathbf{S}[C[E[\mathrm{get}(r)]]] + \sum_{V \in \mathcal{V}(r)} C[E[V]]$

(d) Upward Reference Substitutions

$(\mathtt{subst\text{-}r'}_{\parallel})$	$(M[\mathcal{V}]_\uparrow) \parallel N$	$\to$	$(M \parallel (N[\mathcal{V}]_\downarrow))[\mathcal{V}]_\uparrow$
$(\mathtt{subst\text{-}r'}_{\mathtt{lapp}})$	$((M[\mathcal{V}]_\uparrow)\,N)[\mathcal{U}]_\lambda$	$\to$	$(M\,(N[\mathcal{V}]_\downarrow))[\mathcal{U},\mathcal{V}]_\lambda[\mathcal{V}]_\uparrow$
$(\mathtt{subst\text{-}r'}_{\mathtt{rapp}})$	$(M\,(N[\mathcal{V}]_\uparrow))[\mathcal{U}]_\lambda$	$\to$	$((M[\mathcal{V}]_\downarrow)\,N)[\mathcal{U},\mathcal{V}]_\lambda[\mathcal{V}]_\uparrow$

Congruence case:
If $M \to M'$ by any of the previous rules, then $\mathbf{S}[C[E[M]] \to \mathbf{S}[C[E[M']]]$

Finally:

$(\mathtt{subst\text{-}r'}_{\top})$	$\mathbf{S}[M[\mathcal{V}]_\uparrow]$	$\to$	$\mathbf{S}[M]$

Table 4.3: Reduction Rules. These are closed under the structural rules of Table 4.1

S picks a term in a non-deterministic sum. Note how E does not enforce any reduction order on an application.

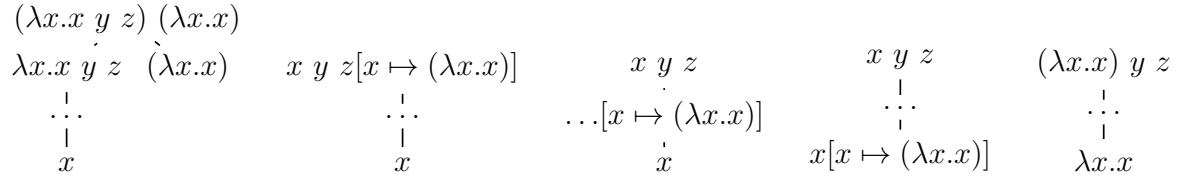
Notation 4.1.5. Given a variable substitution σ and a value V, we define the value $V\{\sigma\}$ as follows: $(\lambda x.M)\{\sigma\} = \lambda x.(M[\sigma])$, $(x)\{\sigma\} = \sigma(x)$ if σ is defined at x or $(x)\{\sigma\} = x$ if not, and $(*)\{\sigma\} = *$.

Notation 4.1.6. We use the notation $\mathcal{X} = \mathcal{V}, \mathcal{W}$ for the juxtaposition of references substitutions. It is defined by $\mathcal{X}(r) = \mathcal{V}(r) + \mathcal{W}(r)$ if both are defined and where $+$ is the union of multisets, $\mathcal{V}(r)$ if only $\mathcal{V}$ is defined, and $\mathcal{W}(r)$ if only $\mathcal{W}$ is defined. We use the same notation for the composition of variable substitutions, defined by $(\sigma, \tau)(x) = \sigma(x)\{\tau\}$ if both are defined, $\sigma(x)$ if only σ is defined $\tau(x)$ if only τ is defined. Finally, we define $\mathcal{V}\{\sigma\} : r \mapsto [V_i\{\sigma\} \mid V_i \in \mathcal{V}(r)]$.

We now give some explanations on the rules of Table 4.3.

(a) β-reduction If one forgets the λ-substitution explained below in (d), this set of rules encapsulates the call-by-value behavior of the language: only values can be substituted in the body of abstractions, and this happens within a thread in a call-by-value applicative context.

(b) Variable Substitutions A variable substitution can be seen as a message emitted by a β-redex and dispatched through the term seen as a tree. The substitution flows from the redex downward the term-tree until it reaches the occurrence of a variable. The occurrence is then replaced, or not, depending on the variable to be substituted. The rules in Table 4.3(a) are an operational formalization of this step-by-step procedure. Consider for example the reduction of the term $(\lambda x.x\ y\ z)\ (\lambda x.x)$. The redex triggers with (β_v) the substitution of all occurrences of x in what was the body of the lambda-abstraction. The substitution goes down the corresponding sub-term and performs the substitution when it reaches an occurrence of x.



Remark 4.1.7. On rules ($\texttt{subst}_{\texttt{subst-r}}$) and ($\texttt{subst}_{\texttt{subst-r'}}$). When composing or swapping substitutions, non-values may appear in unfortunate places: take for example $(*[\mathcal{V}]_\uparrow)[\sigma]$, its reduction should be $(*[\sigma])[\mathcal{V}']_\uparrow$ where $\mathcal{V}'(r) = [V[\sigma] \mid$

$V \in \mathcal{V}(r)]$ when $\mathcal{V}(r)$ is defined. But $V[\sigma]$ are not necessarily values and should then be able to be reduced inside substitutions. However, note that $V[\sigma]$ always reduces in one step to the value $V\{\sigma\}$. To avoid additional complexity, we perform this reduction at the same time, whence the use of $V\{\sigma\}$ instead of $V[\sigma]$ in the actual rules.

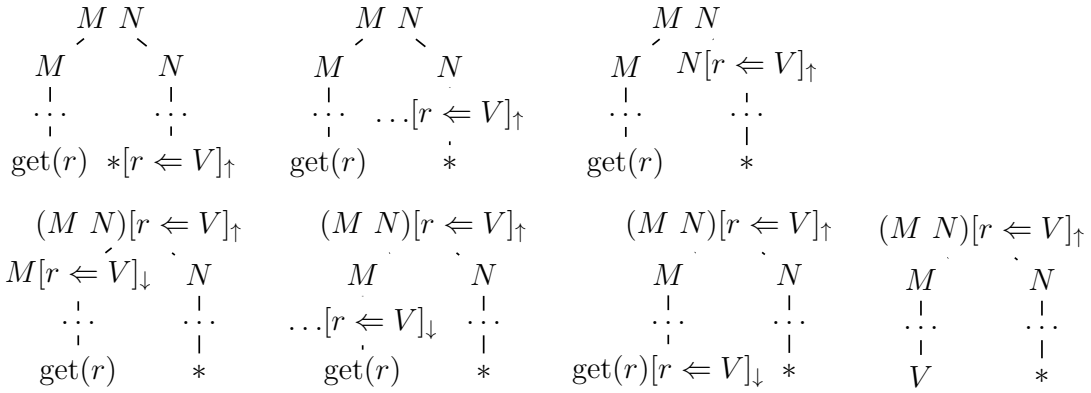
(c) Downward Reference Substitutions An assignment can occur anywhere within a term and it must be able to reach a read located in an arbitrary position. In a language such as [6], the solution is to keep all assignments in a global store. When a read gets evaluated, the value for the reference is taken from the store. This approach is very global in nature: the store is "visible" by every subterm.

The language λ_{cES} features a step-by-step decomposition of reference assignments akin to term variable substitutions: an assignment follows the branches of the term-tree, actively seeking a read. We therefore introduce two sorts of reference substitutions: one that goes downward (indicated by $\downarrow$), similar to variable substitutions, and one that goes upward (indicated by $\uparrow$). Starting from an assignment, the latter climbs up the tree up to the root. The rules in Table 4.3(c) describe the former while the rules in Table 4.3(d) describe the latter.

Remark 4.1.8. In Table 4.3, the rule ($\texttt{subst-r}_{\texttt{get}}$) is the central case of the reduction of reference substitutions. It says that whenever a downward substitution reaches a $\text{get}(r)$, then it generates a non deterministic sum of all the available values for the reference r (if $\mathcal{V}$ is undefined at r, then this sum is understood as a neutral element $\mathbf{0}$) plus a term where the substitution was discarded but the $\text{get}(r)$ is left unreduced. To see why this "remainder" is necessary, consider the term $\text{get}(r)[r \Leftarrow V_1]_{\downarrow}[r \Leftarrow V_2]_{\downarrow}$. If we omit the remainder, the term could reduce to $V_1[r \Leftarrow V_2]_{\downarrow}$ and finally to V_1. But another reduction is possible: one can first reduce the term to $\text{get}(r)[r \Leftarrow V_1, r \Leftarrow V_2]_{\downarrow}$ and then to $V_1 + V_2$. The $\text{get}(r)$ must not be greedy: when it meets a substitution, it has to consider the possibility that other substitutions will be available later. This aspect will be crucial when considering the proof of strong normalization of the language in Section 4.3.

(d) Upward Reference Substitutions Each time an upward reference substitution goes through a multi-ary constructor – as an application or a parallel composition – it propagates downward substitutions in all the children of the constructor except the one it comes from, while continuing its ascension. Eventually, all

the leafs are reached by a corresponding downward substitution. To illustrate the idea, consider a term $M\ N$ where M contains a $\mathrm{get}(r)$ somewhere and N an assignment $*[r \Leftarrow V]_\uparrow$. The reduction of explicit substitutions would go as follows.



One last subtlety in the movement of reference substitutions concerns λ-abstractions. As made explicit in Table 4.3(a), the language is call-by-value: reduction does not happen under λ-abstractions. In particular, a read within the body of a λ-abstraction should only be accessible by an assignment when the λ-abstraction is opened: we have a natural notion of pure and impure terms. Pure terms are terms that will not produce any effect when reduced, and in particular, all values are expected to be pure terms since they cannot reduce further. This is highlighted by rule ($\texttt{subst-r}_{\texttt{val}}$): when encountering a pure term, a reference substitution vanishes. But the case of abstraction is more subtle: computational effects frozen in its body are freed when the abstraction is applied. If one implements naively the reduction rules of reference substitutions, then the following example does not behave as expected: $((\lambda x.\mathrm{get}(r))\ *)[\mathcal{V}]_\downarrow \to ((\lambda x.\mathrm{get}(r))[\mathcal{V}]_\downarrow)\ (*[\mathcal{V}]_\downarrow) \to (\lambda x.\mathrm{get}(r))\ * \to \mathrm{get}(r)$. We end up with an orphan $\mathrm{get}(r)$ despite the fact that a substitution was available at the beginning. The problem is that the substitution diffuses through the application, then encounters two pure terms and vanishes.

In an application, the left term eventually exposes the body of an abstraction, and this body should be able to use any substitution that was in its scope. The λ-substitution $[-]_\lambda$ is a special stationary reference substitution attached to an application. Its goal is precisely to record all the substitutions that went down through it with Rules ($\texttt{subst-r'}_{\texttt{lapp}}$) and ($\texttt{subst-r'}_{\texttt{rapp}}$). When the application is finally reduced with a β_v-rule, this substitution will turn to a downward one and feed the $\mathrm{get}(r)$'s that were hidden in the abstraction's body.

Non-deterministic reduction In the following, we will sometimes not want to deal with the clumsiness of handling sums of terms everywhere. We thus define

an alternative non-deterministic reduction, denoted by $\rightarrow_{\mathtt{nd}}$. If a reduction sequence is seen as a tree, where branching points correspond to ($\mathtt{subst\text{-}r}_{\mathtt{get}}$) and the children to all the summands produced by this rule, then a sequence of reductions $\rightarrow_{\mathtt{nd}}$ corresponds to a branch in this tree.

Definition 4.1.9. Non-deterministic reduction of λ_{cES}
We define $\rightarrow_{\mathtt{nd}}$ by replacing the ($\mathtt{subst\text{-}r}_{\mathtt{get}}$) reduction by the following two rules:

$$\begin{array}{lll} \text{get}(r)[\mathcal{V}]_{\downarrow} & \rightarrow_{\mathtt{nd}} & V \text{ if } V \in \mathcal{V}(r) \\ \text{get}(r)[\mathcal{V}]_{\downarrow} & \rightarrow_{\mathtt{nd}} & \text{get}(r) \end{array}$$

Remark 4.1.10. An alternative approach to λ-substitution would be to make downward substitutions not vanish (i.e. getting rid of Rule ($\mathtt{subst\text{-}r}_{\mathtt{val}}$)). In this situation, values would be handled with their whole context of references assignment. Apart from the heavy syntactical cost of carrying around a lot of similar and possibly useless substitutions, the idea that hidden effects are released at application appears more natural regarding type and effect systems, as the one we introduce in Section 4.2.

Remark 4.1.11. Rule ($\mathtt{subst\text{-}r'}_{\top}$) acts as a garbage collection to eliminate top-level upward substitutions. While not necessary, this will greatly ease the statement and proof of lemmas and theorems (such as Lemma 4.2.7).

4.1.3 Weak confluence

As opposed to calculus with implicit substitution, such as the λ-calculus, λ_{cES} may have numerous overlapping redexes, that are often called critical pairs in the literature. Confluence then becomes painful to establish. We prove a weaker property, weak confluence: an ARS $(A, \rightarrow)$ is weakly confluent if, for any term $t \in A$ such that $u \leftarrow t \rightarrow u'$, then there exists v such that $u \rightarrow^* v \ {}^*\!\leftarrow u'$. The difference with confluence is that in the hypothesis, t is assumed to reduce to u and u' in only one step, instead of an arbitrary number of steps. Thanks to the termination theorem proved in Section 4.3, the Newman's lemma will allow us to deduce that the typed fragment is confluent.

Proposition 4.1.12. Weak confluence
Let $\mathbf{M}$ be a term such that $\mathbf{M} \rightarrow \mathbf{M_1}$ and $\mathbf{M} \rightarrow \mathbf{M_2}$. Then

$$\exists \mathbf{M'}, \mathbf{M_1} \rightarrow^* \mathbf{M'} \text{ and } \mathbf{M_2} \rightarrow^* \mathbf{M'}$$

To show weak confluence, we just examine all the possible critical pairs of the language. The following lemma classifies the different kind of possible critical pairs:

Lemma 4.1.13. Critical pairs
We write $E_1 \# E_2$ for two contexts E_1, E_2 if $E_1 \neq E_2$, and each one is not a prefix of the other, i.e. $\forall E, E_1 \neq E_2[E]$ and $E_2 \neq E_1[E]$. In the following, we write the reduction rules as $S[N] \to S[N'] + \mathbf{M'}$, where $\mathbf{M'}$ is equals to $\mathbf{0}$ unless when ($\texttt{subst-r}_{\texttt{get}}$) occurs, where it may correspond to additional terms.

If $\mathbf{M} \to \mathbf{M_1}$ and $\mathbf{M} \to \mathbf{M_2}$ with $\mathbf{M_1} \neq \mathbf{M_2}$, then one of the assertion holds :

1. The two rules are of the form $S_i[N_i] \to S_i[N_i'] + \mathbf{M}_i'$ with $S_1 \# S_2, \mathbf{M} = S_i[N_i] = \sum_i N_i$
2. The two rules are of the form $S[C_i[N_i]] \to S_i[C_i[N_i']] + \mathbf{M}_i'$ with $C_1 \# C_2, C_i[N_i] = \|_i\ N_i$
3. The two rules are of the form $S[C[N_1]] \to S[C[N_1']]$ and $S[C[C_2[N_2]]] \to S_2[C[C_2[N_2']]] + \mathbf{M}_2'$, the first rule being ($\texttt{subst-r'}_{\|}$).
4. The two rules are of the form $S[C[E[E_i[N_i]]]] \to S_i[C[E[E_i[N_i']]]] + \mathbf{M}_i'$ with $E_1 = E_1'[.]\ E_2'[N_2][\mathcal{V}]_\lambda$ and $E_2 = E_1'[N_1]\ E_2'[.][\mathcal{V}]_\lambda$, $N_i \to N_i'$
5. The two rules are of the form $S[C[E[N_1]]]] \to S[C[E[N_1']]$ and $S[C[E[E_2[N_2]]]] \to S_2[C[E[E_2[N_2']]]] + \mathbf{M}_2'$, with one of the following :
   (a) $N_1 = M'[\mathcal{V}]_\downarrow$ and the applied rule is ($\texttt{subst-r}_{app}$), ($\texttt{subst-r}_{subst-r'}$) or ($\texttt{subst-r}_{merge}$)
   (b) $N_1 = (P[\mathcal{U}]_\uparrow)\ E_2'[N_2][\mathcal{V}]_\lambda$ or $N_1 = E_2'[N_2]\ (P[\mathcal{U}]_\uparrow)[\mathcal{V}]_\lambda$
   (c) $N_1 = (E_2'[N_2][\mathcal{U}]_\uparrow)\ P[\mathcal{V}]_\lambda$ or $N_1 = P\ (E_2'[N_2][\mathcal{U}]_\uparrow)[\mathcal{V}]_\lambda$
   (d) $C = E = [.]$, $N_1 = E_2[N_2][\mathcal{V}]_\uparrow$ and the first rule used is ($\texttt{subst-r'}_\top$)

Proof. Lemma 4.1.13
We can write $\mathbf{M}$ in a unique way (modulo structual rules) as a sum of parallel of simple terms :

$$\mathbf{M} = \sum_k M_k,\ M_k = \ \|_i\ M_k^i$$

- The two reductions rules have a premise of the form $S[N_i]$. Identifying the terms of both sums, we can write $S_1 = [.] + \sum_{k \neq k_1} M_k$ and $S_2 = [.] + \sum_{k \neq k_2} M_k$. If $k_1 \neq k_2$ we are in the case (1), or $S_1 = S_2$.
- If one of the rule used (let say the first one) is ($\texttt{subst-r'}_\top$), then $M_{k_1} = M_{k_1}'[\mathcal{V}]_\uparrow$. Since $\mathbf{M_1} \neq \mathbf{M_2}$, the second rule can't be the same and is of the form $E[T] \to E[T']$. This is the case (5d) of the lemma.

- Otherwise, the premises of the two rules have the form $S[C_1[E_1[P_1]]] \to S[C_1[E_1[P_1']]] + \mathbf{M'}_1$ and $S[C_2[E_2[P_2]]] \to S[C_2[E_2[P_2']]] + \mathbf{M'}_2$. If $C_1 \# C_2$, we are in case (2). If not, then either $C_1 = C_2$ or $C_2 = C_1[C_2'], C_2' \neq [.]$ but the only rule that matches a parallel is ($\texttt{subst-r'}_{\parallel}$), and this is case (3). We assume from now on that $C_1 = C_2$. We decompose E_1 and E_2 by their greatest common prefix, such that $E_1 = E[E_1']$ and $E_2 = E[E_2']$ with either $E_1' = E_2' = [.]$, or $E_1' = [.],\ E_2' \neq [.]$, or $E_1 \# E_2$. The former is excluded since the reducts $\mathbf{N_1}$ and $\mathbf{N_2}$ are assumed differents, and no rule have overlapping redex on base cases (when C and E are empty).

- If $E_1' \# E_2'$, since $E_1'[P_1] = E_2'[P_2]$, then E_1 must be of the form $E_1''\ R[\mathcal{V}]_\lambda$, $E_2 = L\ E_2''[\mathcal{V}]_\lambda$ with $L = E_1''[P_1]$ and $R = E_2''[P_2]$. This is case (4).

- Assume now that one of the two (let say E_1') is $[.]$. Then P_1 and E_2' have a common prefix. If P_1 is an application, it can't be the premise of the (β_v) rule with $P_1 = (\lambda x.M)\ V[\mathcal{U}]_\lambda$, because then $E_2' = E_2''\ V[\mathcal{U}]_\lambda$ or $E_2' = (\lambda x.M)\ E_2''[\mathcal{U}]_\lambda$ but no non-empty context verifies $E_2''[P_2] = V$ for a value V. Thus it must be the premise of ($\texttt{subst-r'}_{\texttt{app}}$), and this corresponds to cases (5b) and (5c).

- If P_1 is not an application, since it must be both the premise of a rule and prefix of the context E_2', the only remaining possibility is $P_1 = \widetilde{P}_1[\mathcal{V}]_\downarrow$. Then $\widetilde{P}_1$ can't be a value $y, *, \lambda y._-$ or $\mathrm{get}(r), \_ \parallel \_, \_[\mathcal{U}]_\uparrow$, because these constructors can't be in E_2' : we are in case (5a).

□

We can then proceed by case analysis to show weak confluence.

Proof. Proposition 4.1.12
We can write $\mathbf{M}, \mathbf{M_1}$ and $\mathbf{M_2}$ in a unique way (modulo structual rules) as a sum of parallel of simple terms :

$$\mathbf{M} = \sum_k M_k,\ \mathbf{M_1} = \sum_k M_k^1,\ \mathbf{M_2} = \sum_k M_k^2$$

Let us examine all the possible cases of Lemma 4.1.13, assuming that $\mathbf{N_1} \neq \mathbf{N_2}$:

1. We have $S_1 \# S_2$, by identifying each terms, $\exists k_1 \neq k_2, S_i = [.] + \sum_{k \neq k_i}$, and we have

$$\begin{array}{rcl} \mathbf{M}_1 & = & \sum_{k \neq k_1} M_k + N_1' + \mathbf{M}_1' \\ & \to & \sum_{k \neq k_1, k_2} M_k + N_1' + N_2' + \mathbf{M}_1' + \mathbf{M}_2' \end{array}$$

as well as $\mathbf{M_2}$.

2. Let write $C_i[N_i] = \|_{l\in\mathcal{L}}\ P_l$. Then there exists $\mathcal{L}_1, \mathcal{L}_2 \subseteq \mathcal{L}$. Since $C_1 \# C_2$, we have $\mathcal{L}_1 \neq \mathcal{L}_2$ and $\mathcal{L}_1 \not\subseteq \mathcal{L}_2$ and $\mathcal{L}_2 \not\subseteq \mathcal{L}_1$, such that $C_i = [.] \parallel (\|_{l\in\mathcal{L}_i}\ P_l)$ and $N_i = \|_{l\notin\mathcal{L}_i}\ P_l$. The only rule that has a parallel of terms as premise is ($\texttt{subst-r'}_{\parallel}$). Thus $\mathcal{L}_i$ are either singletons (if the corresponding rule is not ($\texttt{subst-r'}_{\parallel}$)) or have size two. If they are disjoint, then $\mathcal{L}_2 \subseteq \mathcal{L} \setminus \mathcal{L}_1$ and :

$$\begin{array}{rcl}
\mathbf{M}_1 & = & S[N'_1 \parallel (\|_{l\notin\mathcal{L}_1}\ P_l)] + \mathbf{M}'_1 \\
& = & S[N'_1 \parallel N_2 \parallel (\|_{l\notin\mathcal{L}_1\cup\mathcal{L}_2}\ P_l)] + \mathbf{M}'_1 \\
& \to & S[N'_1 \parallel N'_2 \parallel (\|_{l\notin\mathcal{L}_1\cup\mathcal{L}_2}\ P_l)] + \mathbf{M}'_1 + \mathbf{M}'_2
\end{array}$$

as do $\mathbf{M_2}$.
The only remaining case is if both rules are ($\texttt{subst-r'}_{\parallel}$) and $\mathcal{L}_1 \cap \mathcal{L}_2 = \{l_0\}$. We write $\mathcal{L}_1 = \{l_1, l_0\}, \mathcal{L}_2 = \{l_2, l_0\}$ and $\mathcal{L}_\ni = \mathcal{L} \setminus \{l_0, l_1, l_2\}$. If $P_{l_0} = P'[\mathcal{V}]_\uparrow$ is the "active" upward substitution in both reduction, we have

$$\begin{array}{rcl}
C_1[N'_1] & = & (P_{l_1}[\mathcal{V}]_\downarrow \parallel P')[\mathcal{V}]_\uparrow \parallel P_{l_2} \parallel (\|_{l\neq\mathcal{L}_3}\ P_l) \\
& \to & (P_{l_1}[\mathcal{V}]_\downarrow \parallel P' \parallel P_{l_2}[\mathcal{V}]_\downarrow)[\mathcal{V}]_\uparrow \parallel (\|_{l\neq\mathcal{L}_3}\ P_l)
\end{array}$$

If P_{l_0} is the "passive" term in both reductions, with $P_{l_1} = P'_{l_1}[\mathcal{V}]_\uparrow$ and $P_{l_2} = P'_{l_2}[\mathcal{U}]_\uparrow$, then

$$\begin{array}{rcl}
C_1[N'_1] & = & (P'_{l_1} \parallel P_{l_0}[\mathcal{V}]_\downarrow)[\mathcal{V}]_\uparrow \parallel P'_{l_2}[\mathcal{U}]_\uparrow \parallel (\|_{l\neq\mathcal{L}_3}\ P_l) \\
& \to^* & P_{l_1} \parallel (P_{l_0}[\mathcal{V}]_\downarrow) \parallel (P'_{l_2}[\mathcal{U}]_\uparrow[\mathcal{V}]_\downarrow) \\
& & \parallel (\|_{l\neq\mathcal{L}_3}\ P_l[\mathcal{V}]_\downarrow) \\
& \to & P_{l_1} \parallel (P_{l_0}[\mathcal{V}]_\downarrow) \parallel (P'_{l_2}[\mathcal{V}]_\downarrow[\mathcal{U}]_\uparrow) \\
& & \parallel (\|_{l\neq\mathcal{L}_3}\ P_l[\mathcal{V}]_\downarrow) \\
& \to^* & (P_{l_1}[\mathcal{U}]_\downarrow) \parallel (P_{l_0}[\mathcal{W}]_\downarrow) \parallel (P'_{l_2}[\mathcal{V}]_\downarrow) \\
& & \parallel (\|_{l\neq\mathcal{L}_3}\ P_l[\mathcal{W}]_\downarrow)
\end{array}$$

using repeated ($\texttt{subst-r'}_{\parallel}$), ($\texttt{subst-r'}_{\top}$), ($\texttt{subst-r}_{\texttt{subst-r'}}$) and ($\texttt{subst-r}_{merge}$).
Finally, if P_{l_0} is active in of the two (let say the first) and passive in the other, meaning that $P_{l_0} = P'_{l_0}[\mathcal{V}]_\uparrow, P_{l_2} = P'_{l_2}[\mathcal{U}]_\uparrow$, then

$$\begin{array}{rcl}
C_1[N'_1] & = & (P_{l_1}[\mathcal{V}]_\downarrow \parallel P'_{l_0})[\mathcal{V}]_\uparrow \parallel (P_{l_2}[\mathcal{U}]_\uparrow) \parallel (\|_{l\neq\mathcal{L}_3}\ P_l) \\
& \to & (P_{l_1}[\mathcal{V}]_\downarrow \parallel P' \parallel P_{l_2}[\mathcal{V}]_\downarrow)[\mathcal{V}]_\uparrow \parallel (\|_{l\neq\mathcal{L}_3}\ P_l)
\end{array}$$

$$\begin{array}{rcl} C_1[N_1'] & = & (P_{l_1}[\mathcal{V}]_\downarrow \parallel P'_{l_0})[\mathcal{V}]_\uparrow \parallel P'_{l_2}[\mathcal{U}]_\uparrow \parallel (\|_{l\neq\mathcal{L}_3}\ P_l) \\ & \to^* & (P_{l_1}[\mathcal{V}]_\downarrow) \parallel P'_{l_0} \parallel (P'_{l_2}[\mathcal{U}]_\uparrow[\mathcal{V}]_\downarrow) \\ & & \parallel (\|_{l\neq\mathcal{L}_3}\ P_l[\mathcal{V}]_\downarrow) \\ & \to & (P_{l_1}[\mathcal{V}]_\downarrow) \parallel P'_{l_0} \parallel (P_{l_2}[\mathcal{V}]_\downarrow[\mathcal{U}]_\uparrow) \\ & & \parallel (\|_{l\neq\mathcal{L}_3}\ P_l[\mathcal{V}]_\downarrow) \\ & \to^* & (P_{l_1}[\mathcal{W}]_\downarrow) \parallel (P_{l_0}[\mathcal{U}]_\downarrow) \parallel (P_{l_2}[\mathcal{V}]_\downarrow) \\ & & \parallel (\|_{l\neq\mathcal{L}_3}\ P_l[\mathcal{W}]_\downarrow) \end{array}$$

On the other side,

$$\begin{array}{rcl} C_2[N_2'] & = & P_{l_1} \parallel ((P'_{l_0}[\mathcal{V}]_\uparrow[\mathcal{U}]_\downarrow) \parallel P'_{l_2})[\mathcal{U}]_\uparrow \parallel (\|_{l\neq\mathcal{L}_3}\ P_l) \\ & \to^* & (P_{l_1}[\mathcal{U}]_\downarrow) \parallel (P'_{l_0}[\mathcal{V}]_\uparrow[\mathcal{U}]_\downarrow) \parallel P'_{l_2} \\ & & \parallel (\|_{l\neq\mathcal{L}_3}\ P_l[\mathcal{U}]_\downarrow) \\ & \to & (P_{l_1}[\mathcal{U}]_\downarrow) \parallel (P'_{l_0}[\mathcal{U}]_\downarrow[\mathcal{V}]_\uparrow) \parallel P'_{l_2} \\ & & \parallel (\|_{l\neq\mathcal{L}_3}\ P_l[\mathcal{U}]_\downarrow) \\ & \to^* & (P_{l_1}[\mathcal{W}]_\downarrow) \parallel (P'_{l_0}[\mathcal{U}]_\downarrow) \parallel (P'_{l_2}[\mathcal{V}]_\downarrow) \\ & & \parallel (\|_{l\neq\mathcal{L}_3}\ P_l[\mathcal{W}]_\downarrow) \end{array}$$

3. $C_2 = Q[\mathcal{V}]_\uparrow \parallel C'_2$. Let write $C = \ \|_l\ P_l$ and $C_2 = [.] \parallel P'$. then $C[N_1'] = (Q \parallel (N_2 \parallel P')[\mathcal{V}]_\downarrow)[\mathcal{V}]_\uparrow$.

   - Either $N_2 = N_2'[\mathcal{U}]_\uparrow \parallel Q'$ and

$$\begin{array}{rcl} C[N_1'] & \to^* & Q \parallel ((N_2'[\mathcal{U}]_\uparrow) \parallel Q')[\mathcal{V}]_\downarrow \parallel (\|_l\ P_l[\mathcal{V}]_\downarrow) \\ & \to^* & Q \parallel (N_2'[\mathcal{V}]_\downarrow[\mathcal{U}]_\uparrow) \parallel (Q'[\mathcal{V}]_\downarrow) \parallel (\|_l\ P_l[\mathcal{V}]_\downarrow) \\ & \to^* & (Q[\mathcal{U}]_\downarrow) \parallel (N_2'[\mathcal{V}]_\downarrow) \parallel (Q'[\mathcal{W}]_\downarrow) \parallel (\|_l\ P_l[\mathcal{W}]_\downarrow) \end{array}$$

     and

$$\begin{array}{rcl} C[C_2[N_2']] & = & (Q[\mathcal{V}]_\uparrow) \parallel (N_2' \parallel Q'[\mathcal{U}]_\downarrow)[\mathcal{U}]_\uparrow \parallel (\|_l\ P_l) \\ & \to^* & (Q[\mathcal{V}]_\uparrow[\mathcal{U}]_\downarrow) \parallel N_2' \parallel (Q'[\mathcal{U}]_\downarrow) \parallel (\|_l\ P_l[\mathcal{U}]_\downarrow) \\ & \to^* & (Q[\mathcal{U}]_\downarrow) \parallel (N_2'[\mathcal{V}]_\downarrow) \\ & & \parallel (Q'[\mathcal{W}]_\downarrow) \parallel (\|_l\ P_l[\mathcal{W}]_\downarrow) \end{array}$$

   - Otherwise, N_2 is a premise of the form $E[Q']$ and $S[C[C_2[E[Q']]] \to S[C[C_2[E[Q'']]]] + \mathbf{M}_2'$. Then

$$\begin{array}{rcl} \mathbf{M_1} & \to^* & S[Q \parallel E[Q'][\mathcal{V}]_\downarrow \parallel (\|_l\ P_l[\mathcal{V}]_\downarrow)] \\ & \to & S[Q \parallel E[Q'][\mathcal{V}]_\downarrow \parallel (\|_l\ P_l[\mathcal{V}]_\downarrow)] + \mathbf{M_2}' \end{array}$$

     On the other side,

$$\begin{array}{rcl} \mathbf{M_2} & = & S[(Q[\mathcal{V}]_\uparrow) \parallel E[Q''] \parallel (\|_l\ P_l)] + \mathbf{M_2}' \\ & \to^* & S[Q \parallel (E[Q''][\mathcal{V}]_\downarrow) \parallel (\|_l\ P_l[\mathcal{V}]_\downarrow)] + \mathbf{M_2}' \end{array}$$

4. $$\begin{array}{rcl} S[C[E[E_1[N_1']]]] + \mathbf{M}_1' & = & S[C[E[E_1'[N_1'] \ E_2'[N_2][\mathcal{V}]_\lambda]]] + \mathbf{M}_1' \\ & \to & S[C[E[E_1'[N_1'] \ E_2'[N_2'][\mathcal{V}]_\lambda]]] + \mathbf{M}_1' + \mathbf{M}_2' \end{array}$$

5. (a) The applied rule is either :
   - ($\texttt{subst-r}_{\texttt{app}}$) and $E_2 = E_2' \ Q[\mathcal{U}]_\lambda$ or $E_2 = Q \ E_2'[\mathcal{U}]_\lambda$
   - ($\texttt{subst-r}_{\texttt{subst-r'}}$) and $E_2 = E_2'[\mathcal{U}]_\uparrow$
   - ($\texttt{subst-r}_{\texttt{merge}}$) and $E_2 = E_2'[\mathcal{V}]_\downarrow$

   In the three cases, it is clear that the reductions are independant : the first one can be performed in $\mathbf{M_2}$ and vice-versa to get a common reduct.

The same argument applies to the other four cases. □

4.1.4 Preorder on terms

In the coming Section 4.4 and Section 4.3, we will need to compare the possible behaviors of similar terms. For example, take a term $M[\mathcal{U}, \mathcal{V}]_\downarrow$ that is strongly normalizing. Intuitively, if we remove available reference bindings to form $M[\mathcal{U}]_\downarrow$ or $M[\mathcal{V}]_\downarrow$, the result should also be strongly normalizing: removing available substitutions may only restrict the possible behaviors. The aim of this subsection is formalize this relation between terms of "being similar but able to do more", and to prove the kind of statement about termination we just made above.

To do so, we introduce the $\sqsubseteq$ preorder (and an indexed family $\sqsubseteq_\mathcal{V}$ of preorders), where $M \sqsubseteq N$ means that the two terms are essentially the same, but N may have more available reference substitutions and possibly in different positions. For example, this is typically the case if N is a reduct of $M[\mathcal{V}]_\downarrow$ using only downward structural rules. This preorder encompasses the notion of possible behaviors: $M \sqsubseteq N$ means that everything that can be done by M can be done by N. This is the substance of Proposition 4.1.18.

Definition 4.1.14. Reachability and Associated Preorder
Let M be a term, and N an occurrence of a subterm in M that is not under an abstraction. We define $\mathrm{Reach}(N, M)$, a reference substitution, as the merge of all substitutions that are in scope of this subterm in M, as follows. Recall Notation 4.1.5.

- If $M = N$ then $\mathrm{Reach}(N, M)$ is nowhere defined.
- If $M = M'[\mathcal{U}]_\downarrow$ then $\mathrm{Reach}(N, M) = \mathcal{U}, \mathrm{Reach}(N, M')$, the juxtaposition of $\mathcal{U}$ and $\mathrm{Reach}(N, M')$.
- If $M = M'[\sigma]$ then $\mathrm{Reach}(N, M) = \mathrm{Reach}(N, M')\{\sigma\}$.

- If $M = M'[\mathcal{U}]_\uparrow$ then $\text{Reach}(N, M) = \text{Reach}(N, M')$.
- If $M = M_1\ M_2[\mathcal{U}]_\lambda$ or $M = M_1 \parallel M_2$, let i be the index such that N occurs in M_i, then $\text{Reach}(N, M) = \text{Reach}(N, M_i)$.

We define the skeleton of a term $\text{Sk}(M)$ by removing all downward reference substitutions that are not under an abstraction.

Definition 4.1.15. Skeleton
Let M be a term of λ_{cES}. We define the term $\text{Sk}(M)$ as:

$$\begin{array}{lcl} \text{Sk}(V) & = & V \\ \text{Sk}(M[\mathcal{V}]_\downarrow) & = & \text{Sk}(M) \\ \text{Sk}(M[\mathcal{V}]_\uparrow) & = & \text{Sk}(M)[\mathcal{V}]_\uparrow \\ \text{Sk}(M \parallel N) & = & \text{Sk}(M) \parallel \text{Sk}(N) \\ \text{Sk}(M\ N[\mathcal{V}]_\lambda) & = & \text{Sk}(M)\ \text{Sk}(N) \\ \text{Sk}(M[\sigma]) & = & \text{Sk}(M)[\sigma] \end{array}$$

Definition 4.1.16. Preorder
We say that $M \sqsubseteq N$ if:

- $\text{Sk}(M) = \text{Sk}(N)$, and thus we can put in a one-to-one correspondence the occurrences of $\text{get}(r)$ and $(M_1\ M_2)[\mathcal{V}]_\lambda$ subterms of M and N
- For all such $\text{get}(r)$ occurrences, $\text{Reach}(\text{get}(r), M) \subseteq \text{Reach}(\text{get}(r), N)$
- For all such $M' = (M_1\ M_2)[\mathcal{V}]_\lambda$ corresponding to $N' = (N_1\ N_2)[\mathcal{U}]_\lambda$, then $\text{Reach}(M', M), \mathcal{V} \subseteq \text{Reach}(N', N), \mathcal{U}$

Similarly, we say that $M \sqsubseteq_\mathcal{V} N$ if the difference between reachability sets involved in the definition is somehow "bounded" by $\mathcal{V}$:

- $\text{Sk}(M) = \text{Sk}(N)$
- For all such occurrences of $\text{get}(r)$, $\text{Reach}(\text{get}(r), M) \subseteq \text{Reach}(\text{get}(r), N) \subseteq \text{Reach}(\text{get}(r), M), \mathcal{V}$
- For all such $M' = M_1\ M_2[\mathcal{U}]_\lambda$ corresponding to $N' = N_1\ N_2[\mathcal{W}]_\lambda$, then $\text{Reach}(M', M), \mathcal{W} \subseteq \text{Reach}(N', N), \mathcal{U} \subseteq \text{Reach}(M', M), \mathcal{W}, \mathcal{V}$

The relations $\sqsubseteq$ and $\sqsubseteq_\mathcal{V}$ are partial preorders on terms. We write $M \backsimeq N$ when $M \sqsubseteq N$ and $N \sqsubseteq M$.

$M \sqsubseteq M'$ if M and M' have the same structure but the available substitutions in scope of each $\text{get}(r)$ in M are contained in M' ones. Thus, M' can do at least everything M can do. The second preorder $\sqsubseteq_\mathcal{V}$ controls precisely what the difference between reachability sets can be. The following properties make these explanations formal:

Lemma 4.1.17. Invariance by (`subst-r`) reductions
Let $M \sqsubseteq N$ (resp. $M \sqsubseteq_{\mathcal{V}} N$).

- If $M \to M'$ by a (`subst-r`) rule except (`subst-r`$_{\texttt{get}}$) then $M' \sqsubseteq N$ (resp. $M' \sqsubseteq_{\mathcal{V}} N$)
- If $N \to N'$ by a (`subst-r`) by a (`subst-r`) rule except (`subst-r`$_{\texttt{get}}$) then $M \sqsubseteq N'$ (resp. $M \sqsubseteq_{\mathcal{V}} N'$).

Proof. Lemma 4.1.17
Clearly, a (`subst-r`) rule does not modify the skeleton, so $\mathrm{Sk}(M) = \mathrm{Sk}(M') = \mathrm{Sk}(N)$. It is also almost immediate to see that the rules that propagate reference substitutions, or the (`subst-r`$_{\texttt{val}}$) rule that erases a substitution that is only in the scope of a value, do not modify the reachability sets of an occurrence of $\mathrm{get}(r)$ or $M' = M_1\ M_2[\mathcal{V}]_\lambda$ in M. □

This is the main technical result on the preorders:

Proposition 4.1.18. Simulation preorder
Let $M \sqsubseteq N$ (resp. $M \sqsubseteq_{\mathcal{V}} N$). If $M \to M'$ then $\exists n \geq 0,\ N \to^n N'$ with $M' \sqsubseteq N'$ (resp. $M' \sqsubseteq_{\mathcal{V}} N'$). If the applied rule is not a (`subst-r`) or is (`subst-r`$_{\texttt{get}}$), then $n > 0$.

From this, we deduce the important properties of the preorders:

Corollary 4.1.19. Let $M \sqsubseteq N$ or $M \sqsubseteq_{\mathcal{V}} N$.

1. If $M \sqsubseteq N$, then if N is strongly normalizing, so is M.
2. If $M \backsimeq N$, then M is strongly normalizing if and only if N is.

Proof. Corollary 4.1.19 Let $M \sqsubseteq N$, assume that M has an infinite reduction sequence. We can map this sequence to (finite or infinite) sequence $N \to^* N_1 \to^* N_2 \ldots$ by 4.1.18. As (`subst-r`) rules alone without (`subst-r`$_{\texttt{get}}$) are strongly normalizing, this means that the infinite reduction sequence of M must contain an infinite number of steps that are not (`subst-r`) rules or are a (`subst-r`$_{\texttt{get}}$). But then, as such steps are simulated by at least one step in the sequence $N \to^* N_1 \to^* \ldots$, the latter must be infinite. This contradicts the fact that N is strongly normalizing. Thus M has no infinite reduction sequence. □

Proof. Proposition 4.1.18
Before simulating the reduction of M in N, we may have to carry around reference substitutions that are at a different level in the two terms. Let us first prove that if $M = C[E[P]]$, then $N \to^* C[E'[P']]$, that is we can reduce M to a term with the same prefix $C[.]$ by pushing down pending reference substitutions.

To do so, we just apply the rule ($\texttt{subst-r}_{\|}$) until it is not possible anymore to get $N \to^* C[N']$. Then, $E[P]$ and N' having the same skeleton, N' can be written as $E'[P']$ where E' is E with additional references substitutions, and P and P' have the same skeleton and the same head constructor. Indeed, if P' would have additional substitutions in head position we could always include them in E': in fact we take the maximal E' that satisfies the previous decomposition.

Note that the reachability sets of subterms in C (respectively E,E') only depends on C (resp. C, E and C, E'). On the other side, the reachability sets of subterms in P (resp. P') are unions of substitutions occurring in P, E (resp. P', E') and C.

- If the reduction rule is one of the ($\texttt{subst-r}$) except ($\texttt{subst-r}_{\texttt{get}}$), by Lemma 4.1.17, $n = 0$ works.

- ($\texttt{subst-r}_{\texttt{get}}$) : $P = \text{get}(r)[\mathcal{V}]_{\downarrow} \to V \in \mathcal{V}(r)$ and $P' = \text{get}(r)[\mathcal{V}']_{\downarrow}$. $\mathcal{V}(r)$ is in $\text{Reach}(\text{get}(r), N)$, so by iterated application of ($\texttt{subst-r}$) rules except ($\texttt{subst-r}_{\texttt{get}}$) and ($\texttt{subst-r}_{\texttt{val}}$), we can push (without modifying the skeleton nor the reachability sets) the corresponding substitutions down until it reaches $\text{get}(r)$ in P'. Then, we can perform a ($\texttt{subst-r}_{\texttt{get}}$) reduction. All the other reachability sets of gets or application are left unmodified.

- (β_v) : $P = (\lambda x.Q)\ V[\mathcal{V}]_{\lambda} \to T = Q[x/V][\mathcal{V}]_{\downarrow}$. Up to ($\texttt{subst-r}_{\texttt{val}}$) reductions, $P' = (\lambda x.Q)\ V[\mathcal{V}']_{\lambda} \to T' = Q[x/V][\mathcal{V}']_{\downarrow}$. The condition on reachabiliy sets for application in the defintion of $\sqsubseteq$ precisely ensures that all the gets and applications in Q have the same reachability in $C[E[T]]$ and in $C[E'[T']]$.

($\texttt{subst}$) rules

- ($\texttt{subst}_{\texttt{var}}$) : $P = x[\sigma] \to x$ or $\sigma(x)$, and $P' = x[\mathcal{V}]_{\downarrow}[\sigma] \to x[\sigma] \to x$ or $\sigma(x)$.

- ($\texttt{subst}_{\texttt{app}}$) : $P = Q_1\ Q_2[\mathcal{V}]_{\lambda}[\sigma] \to T = (Q_1[\sigma])\ (Q_2[\sigma])[(V[\sigma])]_{\lambda}$. We have $P' = (Q_1'[\mathcal{U}]_{\lambda})\ (Q_2'[\mathcal{W}]_{\lambda})[\mathcal{V}']_{\lambda}[\sigma] \to^* (Q_1[\sigma][U[\sigma]]_{\downarrow})\ (Q_2[\sigma][V[t]W]_{\downarrow})[V'[\sigma]]_{\lambda}$. By definition of reachability sets, they are invariant by all the rule applied.

- We proceed the same way for other cases : the var substitution just go through the additionnal references substitutions, and by design, reachability sets are not modified.

($\texttt{subst-r'}$) rules

- Upward substitutions commute with downward ones without interacting. On the other hand, they can span new downward substitutions but in this case, they do it in the same way for both P and P' and thus do not modify the inclusion relation on reachability sets.

□

4.2 Stratification and Type System

We present in this section a stratified type and effect system for λ_{cES} inspired from [6, 41]. A type and effect system aims at statically track the potential effects that a term can produce when reduced. Here, the considered effects are read from or write to references.

4.2.1 The Type System of λ_{cES}

Formally, the type and effect system is defined as follows.

$$\begin{array}{lrcl} \text{-effects} & e, e' & \subset & \{r_1, r_2, \ldots\} \\ \text{-types} & \alpha & ::= & \mathbf{B} \mid A \\ \text{-value types} & A & ::= & \texttt{Unit} \mid A \xrightarrow{e} \alpha \mid \text{Ref}_r A \end{array}$$

The type `Unit` is the type of $*$. The function type $A \xrightarrow{e} \alpha$ is annotated with an effect e: the set of references the function is allowed to use. Finally, the type $\text{Ref}_r A$ states that the reference r can only be substituted with values of type A. Since thread cannot be fed as an argument to a function, the type of the parallel components of a program is irrelevant. They are given the opaque behavior type $\mathbf{B}$. We separate α-types and A-types to ensure that $\mathbf{B}$ cannot be in the domain of a function.

In the typing rules we use two distinct contexts: variable contexts Γ of the form $x_1 : A_1, \ldots, x_n : A_n$ and reference contexts R of the form $r_1 : A_1, \ldots, r_n : A_n$. The latter indicates the type of the values that a reference r appearing in M can hold. If the order of variables in Γ is irrelevant, the order of references in R is important.

In order to ensure termination, the type and effect system is stratified: this stratification induces an order forbidding circularity in reference assignments. It is presented as a set of rules to build the reference context and can be found in Figure 4.1. It states that when a new reference is added to the context, all references appearing in its type must already be in R. In Figure 4.1 the entailment symbol ($\vdash$) is overloaded with several meanings:

$$\frac{}{\emptyset \vdash} \qquad \frac{R \vdash A \qquad r \notin \mathrm{dom}(R)}{R, r : A \vdash} \qquad \frac{R \vdash}{R \vdash \mathtt{Unit}} \qquad \frac{R \vdash}{R \vdash \mathbf{B}}$$

$$\frac{R \vdash A \qquad R \vdash \alpha \qquad e \subseteq \mathrm{dom}(R)}{R \vdash A \xrightarrow{e} \alpha} \qquad \frac{R \vdash \qquad r : A \in R}{R \vdash \mathrm{Ref}_r A}$$

Figure 4.1: Stratification of the type system

- R is well formed, written $R \vdash$, means that the references appearing in R are stratified.
- A type α is well formed under R, written $R \vdash (\alpha, e)$, means that all references appearing in e and α are in R.
- A variable context Γ is well formed under R, written $R \vdash \Gamma$, means that all the types appearing in Γ are well formed under R.

The type and effect system features a subtyping relation whose definition rules are presented in Figure 4.2. It formalizes the idea that a function of type $A \xrightarrow{\{r\}} \alpha$ is not obliged to use the reference r.

Typing judgments overload once more the symbol ($\vdash$) and take the form $R; \Gamma \vdash M : (\alpha, e)$ where R is the reference context, Γ the variable context, α the type of M and e the references that M may affect. Using the stratification and the subtyping relation, the typing rules for the language λ_{cES} are presented in Figure 4.3. For succinctness, the application rule has been factorized into two rules, (APP) and (SUBST) for $\xi = \lambda$. Thus (APP) is not a legitimate rule but an abuse of notation, and must be followed by an appropriate (SUBST) in any type derivation.

Remark 4.2.1. In Rule (LAM), when abstracting over a variable in a term $R; \Gamma, x : A \vdash M : (\alpha, e)$, the resulting value $\lambda x.M$ is pure and hence its effects should be the empty set. However one must remember that the body of this abstraction is potentially effectful: this is denoted by annotating the functional arrow "$\rightarrow$" with a superscript indicating these effects. Also note that in general, the order of references in R is capital: it is the order induced by stratification.

Let (α, e) be a type and effect well-formed under a reference context R, i.e. $R \vdash (\alpha, e)$. We give two definitions that allow to refer to the effects contained implicitly or explicitly in (α, e). $\mathtt{Eff}_R(\alpha, e)$ represents the effects of e together with the effects placed on the arrows of α. $\mathtt{Reg}_R(\alpha, e)$, includes not only $\mathtt{Eff}_R(\alpha, e)$ but also all the effects involved in the judgement $R \vdash (\alpha, e)$.

$$\frac{}{R \vdash \alpha \leq \alpha}\ (ref) \qquad \frac{e \subset e' \subset \mathrm{dom}(R) \qquad R \vdash \alpha \leq \alpha'}{R \vdash (\alpha, e) \leq (\alpha', e')}\ (cont)$$

$$\frac{R \vdash A' \leq A \qquad R \vdash (\alpha, e) \leq (\alpha', e')}{R \vdash A \xrightarrow{e} \alpha \leq A' \xrightarrow{e'} \alpha'}\ (arrow)$$

Figure 4.2: Subtyping relation

$$\frac{R \vdash \Gamma, x : A}{R; \Gamma, x : A \vdash x : (A, \emptyset)}\ (var) \qquad \frac{R \vdash \Gamma}{R; \Gamma \vdash * : (\mathtt{Unit}, \emptyset)}\ (unit) \qquad \frac{R \vdash \Gamma \qquad r : A \in R}{R; \Gamma \vdash r : Ref_r A}\ (reg)$$

$$\frac{R; \Gamma, x : A \vdash M : (\alpha, e)}{R; \Gamma \vdash \lambda x.M : (A \xrightarrow{e} \alpha, \emptyset)}\ (lam) \qquad \frac{R; \Gamma \vdash M : (A \xrightarrow{e_1} \alpha, e_2) \qquad R; \Gamma \vdash N : (A, e_3)}{R; \Gamma \vdash M\ N : (\alpha, e_1 \cup e_2 \cup e_3)}\ (app)$$

$$\frac{R; \Gamma \vdash r : \mathrm{Ref}_r A}{R; \Gamma \vdash \mathrm{get}(r) : (A, \{r\})}\ (get) \qquad \frac{R; \Gamma \vdash M : (\alpha, e) \qquad R \vdash (\alpha, e) \leq (\alpha', e')}{R; \Gamma \vdash M : (\alpha', e')}\ (sub)$$

$$\frac{R; \Gamma, x_1 : A_1, \ldots, x_n : A_n \vdash M : (\alpha, e) \qquad \forall i : \quad R; \Gamma \vdash V_i : (A_i, \emptyset)}{R; \Gamma \vdash M[\forall i : \ x_i \mapsto V_i] : (\alpha, e)}\ (subst)$$

$$\frac{\forall i : \ R; \Gamma \vdash r_i : \mathrm{Ref}_{r_i} A_i \quad R; \Gamma \vdash M : (\alpha, e) \quad \forall i : \ r_i \in e \quad \forall i : \ V \in \mathscr{E}_i \implies R; \Gamma \vdash V : (A_i, \emptyset)}{R; \Gamma \vdash M[\forall i : \ r_i \mapsto \mathscr{E}_i]_\xi : (\alpha, e) \qquad \text{for } \xi \in \{\uparrow, \downarrow, \lambda\}}\ (subst\text{-}r)$$

$$\frac{i = 1, 2 \qquad R; \Gamma \vdash M_i : (\alpha_i, e_i)}{R; \Gamma \vdash M_1 \parallel M_2 : (\mathbf{B}, e_1 \cup e_2)}\ (par) \qquad \frac{i = 1, 2 \qquad R; \Gamma \vdash \mathbf{M_i} : (\alpha, e)}{R; \Gamma \vdash \mathbf{M}_1 + \mathbf{M}_2 : (\alpha, e)}\ (sum)$$

Figure 4.3: Typing rules for λ_{cES}

For example, take $R = r : Ref_r\ \mathtt{Unit}, s : Ref_s\ (\mathtt{Unit} \xrightarrow{\{r\}} \mathtt{Unit})$ and $\alpha = \mathtt{Unit} \xrightarrow{\{s\}} \mathtt{Unit}$. Then $\mathtt{Eff}_R(\alpha, \emptyset) = \{s\}$ while $\mathtt{Reg}_R(\alpha, \emptyset) = \{r, s\}$.

Definition 4.2.2. Effect of a type and effect
Let $R \vdash (\alpha, e)$. We define the effect $\mathtt{Eff}_R(\alpha)$ as:

$$\begin{array}{lcl} \mathtt{Eff}_R(\mathtt{Unit}) & = & \emptyset \\ \mathtt{Eff}_R(\mathbf{B}) & = & \emptyset \\ \mathtt{Eff}_R(A \xrightarrow{e} \alpha) & = & \mathtt{Eff}_R(A) \cup \mathtt{Eff}_R(\alpha) \cup e \end{array}$$

We define $\mathtt{Eff}_R(\alpha, e) = \mathtt{Eff}_R(\alpha) \cup e$.

Definition 4.2.3. Region of a type and effect
Let $R \vdash (\alpha, e)$. We define the set $\mathtt{Reg}_R(\alpha)$ as:

$$
\begin{array}{lcl}
\mathtt{Reg}_R(\mathtt{Unit}) & = & \emptyset \\
\mathtt{Reg}_R(\mathbf{B}) & = & \emptyset \\
\mathtt{Reg}_R(A \xrightarrow{e} \alpha) & = & \mathtt{Reg}_R(A) \cup \mathtt{Reg}_R(e) \cup \mathtt{Reg}_R(\alpha)
\end{array}
$$

Where $\mathtt{Reg}_R(r)$ for a reference r is defined by $\mathtt{Reg}_R(r) = \{r\} \cup \mathtt{Reg}_R(R(r), \emptyset)$, and is naturally extended to an effect e by $\mathtt{Reg}_R(e) = \bigcup_{r \in e} \mathtt{Reg}_R(r)$. Finally, the region of a type and effect (α, e) is defined as $\mathtt{Reg}_R(\alpha, e) = \mathtt{Reg}_R(\alpha) \cup \mathtt{Reg}_R(e)$.

Remark 4.2.4. Note that we always have $\mathtt{Eff}_R(\alpha, e) \subseteq \mathtt{Reg}_R(\alpha, e) \subseteq \mathrm{dom}(R)$.

4.2.2 Subject reduction

The language λ_{cES} satisfies the usual safety properties of a typed calculus: subjet reduction and progress.

Lemma 4.2.5. Subject reduction
Let $R; \Gamma \vdash M : (\alpha, e)$ be a typing judgment, and assume that $M \to M'$. Then $R; \Gamma \vdash M' : (\alpha, e)$. □

Remark 4.2.6. The fact that an effectful term may become pure after reduction is reflected by the subtyping relation. For example, consider $P = \mathrm{get}(r)[\mathcal{V}]_\downarrow$ where $R \vdash P : (A, \{r\})$ and $P \to (V + \ldots)$. Since V is a value it can only be given the type $R \vdash V : (A, \emptyset)$. Subject reduction would however require that V has the same type $(A, \{r\})$ as P. The subtyping relation corresponds to effect containment, meaning that the effects appearing in types are an upper bound of the actual effects produced by a term, so that $(A, \emptyset)$ is a subtype of $(A, \{r\})$.

Proof. Lemma 4.2.5
Case analysis on the reduction rule used :

Main reduction rules

- (β_v) Then the typing derivation of M is of the form :

$$
\dfrac{R; \Gamma \vdash \mathrm{Reg}_r B \qquad \forall U \in \mathcal{U} :\ R; \Gamma \vdash U : (B, \emptyset) \qquad \dfrac{\dfrac{R; \Gamma, x : A \vdash M : (\alpha, e)}{R; \Gamma \vdash (\lambda x.M) : (A \xrightarrow{e}, \emptyset)}\ (\textsc{lam}) \qquad R; \Gamma \vdash V : (A, \emptyset)}{R; \Gamma \vdash (\lambda x.M)\ V : (\alpha, e)}\ (\textsc{app})}{R; \Gamma \vdash (\lambda x.M)\ V[\mathcal{U}]_\lambda : (\alpha, e)}\ (\textsc{subst-r})
$$

  Then one can form the derivation

$$
\dfrac{R; \Gamma \vdash \mathrm{Reg}_r B \qquad \forall U \in \mathcal{U} :\ R; \Gamma \vdash U : (B, \emptyset) \qquad \dfrac{R; \Gamma \vdash V : (A, \emptyset) \qquad R; \Gamma, x : A \vdash M : (\alpha, e)}{R; \Gamma \vdash M[x/V] : (\alpha, e)}\ (\textsc{subst})}{R; \Gamma \vdash M[x/V][\mathcal{U}]_\downarrow : (\alpha, e)}\ (\textsc{subst-r})
$$

  using the admissibility of generalized weakening.

Variables substitutions

- ($\texttt{subst}_{\texttt{var}}$) We have

$$\dfrac{R;\Gamma \vdash \sigma(x_i) : (A_i, \emptyset) \qquad \dfrac{}{R;\Gamma' \vdash y : (B, \emptyset)}\ (\text{VAR})}{R;\Gamma \vdash y[\sigma] : (B, \emptyset)}\ (\text{SUBST})$$

  with $\Gamma' = \Gamma, x_1 : A_1, \ldots, x_n : A_n$ if i ranges over $1 \ldots n$. We have

$$\dfrac{}{R;\Gamma \vdash y : (B, \emptyset)}\ (\text{VAR})$$

  if σ is undefined at y or

$$R;\Gamma \vdash \sigma(y) : (A_{i_0}, \emptyset)$$

  if $\exists i_0, x_{i_0} = y$ and thus $A_{i_0} = B$.

- ($\texttt{subst}_{\texttt{unit}}$) : just apply the ($\texttt{unit}$) typing rule.

- ($\texttt{subst}_{\texttt{app}}$)

$$\dfrac{R;\Gamma' \vdash r_i : \mathrm{Reg}_{r_i} A_i \qquad \forall V \in \mathcal{V}(r_i) :\ R;\Gamma' \vdash V : (A_i, \emptyset) \qquad \dfrac{R;\Gamma' \vdash M : (C \xrightarrow{e_1} \alpha, e_2) \qquad R;\Gamma' \vdash N : (C, e_3)}{R;\Gamma' \vdash M\ N : (\alpha, e)}\ (\text{APP})}{R;\Gamma' \vdash M\ N[\mathcal{V}]_\lambda}\ (\text{SUBST-R})$$

$$(\text{SUBST})\ \dfrac{R;\Gamma' \vdash \sigma(x_j) : (B_j, \emptyset) \qquad R;\Gamma' \vdash M\ N[\mathcal{V}]_\lambda}{R;\Gamma \vdash M\ N[\mathcal{V}]_\lambda[\sigma] : (\alpha, e)}$$

  With $\Gamma' = \Gamma, x_1 : A_1, \ldots, x_n : A_n$ and $e_1 \cup e_2 \cup e_3 = e$. We can derive

$$(\text{SUBST})\ \dfrac{\dfrac{R;\Gamma \vdash \sigma(x_j) : (B_j, \emptyset) \qquad R;\Gamma' \vdash M : (C \xrightarrow{e_1} \alpha, e_2)}{R;\Gamma \vdash M[\sigma] : (C \xrightarrow{e_1} \alpha, e_2)} \qquad \dfrac{R;\Gamma \vdash \sigma(x_j) : (B_j, \emptyset) \qquad R;\Gamma' \vdash N : (C, e_3)}{R;\Gamma \vdash N[\sigma] : (C, e_3)}\ (\text{SUBST})}{R;\Gamma \vdash (M[\sigma])\ (N[\sigma]) : (\alpha, e)}\ (\text{APP})$$

  For any $V \in \mathcal{V}(r_i)$, one has

$$\dfrac{R;\Gamma \vdash \sigma(x_j) : (B_j, \emptyset) \qquad R;\Gamma' \vdash V : (A_i, \emptyset)}{R;\Gamma \vdash V[\sigma] : (A_i, \emptyset)}\ (\text{SUBST})$$

  And finally

$$\dfrac{R;\Gamma \vdash P : (\alpha, e) \qquad R;\Gamma \vdash r_i : \mathrm{Reg}_{r_i} A_i \qquad \forall V \in \mathcal{V}(r_i) :\ R;\Gamma \vdash V[\sigma] : (A_i, \emptyset)}{R;\Gamma \vdash (M[\sigma])\ (N[\sigma])[\mathcal{V}[\sigma]]_\lambda}\ (\text{SUBST-R})$$

- ($\texttt{subst}_{\lambda}$) :

$$\dfrac{R;\Gamma \vdash \sigma(x_i) : (A_i, \emptyset) \qquad \dfrac{R;\Gamma', y : A \vdash M : (\alpha, e)}{R;\Gamma' \vdash \lambda y.M : (A \xrightarrow{e} \alpha, \emptyset)}\ (\text{LAM})}{R;\Gamma \vdash \lambda y.M[\sigma] : (A \xrightarrow{e} \alpha, \emptyset)}\ (\text{SUBST})$$

  then we just swap the two rules :

$$\dfrac{\dfrac{\dfrac{R;\Gamma \vdash \sigma(x_i) : (A_i, \emptyset)}{R;\Gamma, y : A \vdash \sigma(x_i) : (A_i, \emptyset)}\,(\textsc{weak}) \qquad R;\Gamma', y : A \vdash M : (\alpha, e)}{R;\Gamma, y : A \vdash M[\sigma] : (\alpha, \emptyset)}\,(\textsc{subst})}{R;\Gamma \vdash \lambda y.(M[\sigma]) : (A \xrightarrow{e} \alpha, \emptyset)}\,(\textsc{lam})$$

- $(\texttt{subst}_{\texttt{get}})$:

$$\dfrac{R;\Gamma \vdash \sigma(x_i) : (A_i, \emptyset) \qquad \dfrac{R;\Gamma' \vdash r : \mathrm{Reg}_r B}{R;\Gamma' \vdash \mathrm{get}(r) : (B, \{r\})}\,(\textsc{get})}{R;\Gamma \vdash \mathrm{get}(r)[\sigma] : (B, \{r\})}\,(\textsc{subst})$$

We can juste recast the get rule without the weakened x_is :

$$\dfrac{R;\Gamma \vdash r : \mathrm{Reg}_r B}{R;\Gamma \vdash \mathrm{get}(r) : (B, \{r\})}\,(\textsc{get})$$

- $(\texttt{subst}_{\|})$:

$$\dfrac{R;\Gamma \vdash \sigma(x_j) : (A_j, \emptyset) \qquad \dfrac{i = 1, 2 \qquad R;\Gamma' \vdash M_i : (\alpha_i, e_i)}{R;\Gamma' \vdash M_1 \parallel M_2 : (\mathbf{B}, e)}\,(\textsc{par})}{R;\Gamma \vdash M_1 \parallel M_2[\sigma] : (\mathbf{B}, e)}\,(\textsc{subst})$$

Then

$$\dfrac{i = 1, 2 \qquad \dfrac{R;\Gamma \vdash \sigma(x_j) : (A_j, \emptyset) \qquad R;\Gamma' \vdash M_i : (\alpha_i, e_i)}{R;\Gamma \vdash M_i[\sigma] : (\alpha_i, e_i)}\,(\textsc{subst})}{R;\Gamma \vdash (M_1[\sigma]) \parallel (M_2[\sigma]) : (\mathbf{B}, e)}\,(\textsc{par})$$

- $(\texttt{subst}_{subst-r})$:

$$\dfrac{R;\Gamma \vdash \sigma(x_j) : (A_j, \emptyset) \qquad \dfrac{R;\Gamma' \vdash r_i : \mathrm{Reg}_{r_i} A_i \qquad \forall U \in \mathcal{U}(r_i) :\ R;\Gamma' \vdash U : (A_i, \emptyset) \qquad R;\Gamma' \vdash M : (\alpha, e)}{R;\Gamma' \vdash M[\mathcal{U}]_{\downarrow} : (\alpha, e)}\,(\textsc{subst-r})}{R;\Gamma \vdash M[\mathcal{U}]_{\downarrow}[\sigma] : (\alpha, e)}\,(\textsc{subst})$$

Then for any $U \in \mathcal{U}(r_i)$

$$\dfrac{R;\Gamma' \vdash U : (A_i, \emptyset)}{R;\Gamma \vdash U[\sigma] : (A_i, \emptyset)}\,(\textsc{subst})$$

It follows

$$\dfrac{R;\Gamma \vdash r_i : \mathrm{Reg}_{r_i} A_i \qquad \forall U \in \mathcal{U}(r_i) :\ R;\Gamma \vdash U : (A_i, \emptyset) \qquad \dfrac{R;\Gamma \vdash \sigma(x_j) : (A_j, \emptyset) \qquad R;\Gamma' \vdash M : (\alpha, e)}{R;\Gamma \vdash :[\sigma/M](\alpha, e)}\,(\textsc{subst})}{R;\Gamma \vdash M[\sigma][\mathcal{U}[\sigma]]_{\downarrow} : (\alpha, e)}\,(\textsc{subst-r})$$

- ($\texttt{subst}_{subst-r'}$) : exactly as the previous case (the typing rules and reduction rules being similar)

- ($\texttt{subst}_{merge}$) We need a simple substitution lemma to handle this case of the meta substitution. Let $\Gamma_{\mathcal{I}} = \bullet_{i\in\mathcal{I}} x_i : A_i$ and $\Gamma_{\mathcal{J}} = \bullet_{j\in\mathcal{J}} x_j : A_j$ where $\bullet$ is the list concatenation.

$$\dfrac{\forall U \in \mathcal{U}(r_j) :\ R;\Gamma \vdash U : (A_j, \emptyset) \qquad \dfrac{\forall V \in \mathcal{V}(r_i) :\ R;\Gamma,\Gamma_{\mathcal{J}} \vdash V : (A_i,\emptyset) \qquad R;\Gamma,\Gamma_{\mathcal{J}},\Gamma_{\mathcal{I}} \vdash M : (\alpha, e)}{R;\Gamma_{\mathcal{J}} \vdash M[(x_i)/(V_i)] : (\alpha, e)}\ (\textsc{subst})}{R;\Gamma \vdash M[(x_i)/(U_i)][(y_j)/(U_j)] : (\alpha, e)}\ (\textsc{subst})$$

  Then we have

$$\dfrac{\forall V \in \mathcal{V}_i :\ R;\Gamma \vdash V\{(U_j)/(y_j)\} : (A_i,\emptyset) \qquad \forall U \in \mathcal{U}_j :\ R;\Gamma \vdash U : (A_j,\emptyset) \qquad R;\Gamma,\Gamma_{\mathcal{J}},\Gamma_{\mathcal{I}} \vdash M : (\alpha, e)}{R;\Gamma \vdash M[(z_i)/(W_i)] : (\alpha, e)}\ (\textsc{subst})$$

Downward references substitutions

- ($\texttt{subst-r}_{\texttt{val}}$)

$$\dfrac{R;\Gamma \vdash r_i : \mathrm{Reg}_{r_i} A_i \qquad \forall U \in \mathcal{U}(r_i) :\ R;\Gamma \vdash U : (A_i,\emptyset) \qquad \dfrac{R;\Gamma \vdash V : (A,\emptyset)}{R;\Gamma \vdash V : (A,\{r\})}\ (\textsc{sub})}{R;\Gamma \vdash V[\mathcal{U}]_{\downarrow} : (A, e)}\ (\textsc{subst-r})$$

  Then we just reuse the typing derivation of

$$\dfrac{R;\Gamma \vdash V : (A,\emptyset)}{R;\Gamma \vdash V : (A,\{r\})}\ (\textsc{sub})$$

- ($\texttt{subst-r}_{\parallel}$)

$$\dfrac{R;\Gamma \vdash r_i : \mathrm{Reg}_{r_i} A_i \qquad \forall U \in \mathcal{U}(r_i) :\ R;\Gamma \vdash U : (A_i,\emptyset) \qquad \dfrac{R;\Gamma \vdash M_i : (\alpha_i, e_i)}{R;\Gamma \vdash M_1 \parallel M_2 : (\mathbf{B}, e)}\ (\textsc{par})}{R;\Gamma \vdash M_1 \parallel M_2[\mathcal{U}]_{\downarrow} : (\mathbf{B}, e)}\ (\textsc{subst-r})$$

  Then we commute the two rules

$$\dfrac{\dfrac{R;\Gamma \vdash r_i : \mathrm{Reg}_{r_i} A_i \qquad \forall U \in \mathcal{U}(r_i) :\ R;\Gamma \vdash U : (A_i,\emptyset) \qquad R;\Gamma \vdash M_i : (\alpha_i, e)}{R;\Gamma \vdash M_i[\mathcal{U}]_{\downarrow} : (\alpha_i, e)}\ (\textsc{subst-r})}{R;\Gamma \vdash (M_1[\mathcal{U}]_{\downarrow}) \parallel (M_2[\mathcal{U}]_{\downarrow}) : (\mathbf{B}, e)}\ (\textsc{par})$$

- ($\texttt{subst-r}_{\texttt{subst-r'}}$)

$$\frac{R;\Gamma \vdash r_i : \mathrm{Reg}_{r_i} A_i \qquad \forall V \in \mathcal{V}(r_i) :\ R;\Gamma \vdash V : (A_i, \emptyset) \qquad R;\Gamma \vdash M : (\alpha, e)}{R;\Gamma \vdash M[\mathcal{V}]_\uparrow : (\alpha, e)}\ (\textsc{subst-r})$$

$$\frac{R;\Gamma \vdash r_j : \mathrm{Reg}_{r_j} B_j \qquad \forall U \in \mathcal{U}(r_j) :\ R;\Gamma \vdash U : (B_j, \emptyset) \qquad R;\Gamma \vdash M[\mathcal{V}]_\downarrow : (\alpha, e)}{R;\Gamma \vdash M[\mathcal{V}]_\uparrow[\mathcal{U}]_\downarrow : (\alpha, e)}\ (\textsc{subst-r})$$

We just swap the two rules

$$\frac{R;\Gamma \vdash r_j : \mathrm{Reg}_{r_j} B_j \qquad \forall U \in \mathcal{U}(r_j) :\ R;\Gamma \vdash U : (B_j, \emptyset) \qquad R;\Gamma \vdash M : (\alpha, e)}{R;\Gamma \vdash M[\mathcal{U}]_\downarrow : (\alpha, e)}\ (\textsc{subst-r})$$

$$\frac{R;\Gamma \vdash r_i : \mathrm{Reg}_{r_i} A_i \qquad \forall V \in \mathcal{V}(r_i) :\ R;\Gamma \vdash V : (A_i, \emptyset) \qquad R;\Gamma \vdash M[\mathcal{U}]_\downarrow : (\alpha, e)}{R;\Gamma \vdash M[\mathcal{U}]_\downarrow[\mathcal{V}]_\uparrow : (\alpha, e)}\ (\textsc{subst-r})$$

- (`subst-r-merge`)

$$\frac{R;\Gamma \vdash r_i : \mathrm{Reg}_{r_i} A_i \qquad \forall V \in \mathcal{V}(r_i) :\ R;\Gamma \vdash V : (A_i, \emptyset) \qquad R;\Gamma \vdash M : (\alpha, e)}{R;\Gamma \vdash M[\mathcal{V}]_\downarrow : (\alpha, e)}\ (\textsc{subst-r})$$

$$\frac{R;\Gamma \vdash r_j : \mathrm{Reg}_{r_j} B_j \qquad \forall U \in \mathcal{U}(r_j) :\ R;\Gamma \vdash U : (B_j, \emptyset) \qquad R;\Gamma \vdash M[\mathcal{V}]_\downarrow : (\alpha, e)}{R;\Gamma \vdash M[\mathcal{V}]_\downarrow[\mathcal{U}]_\downarrow : (\alpha, e)}\ (\textsc{subst-r})$$

Then we have

$$\frac{R;\Gamma \vdash r_k : \mathrm{Reg}_{r_k} A_k \qquad \forall W \in (\mathcal{U} + \mathcal{V})(x_k) :\ R;\Gamma \vdash W : (A_k, \emptyset) \qquad R;\Gamma \vdash M : (\alpha, e)}{R;\Gamma \vdash M[\mathcal{V}, \mathcal{U}]_\downarrow : (\alpha, e)}\ (\textsc{subst-r})$$

- (`subst-r-app`)

$$\frac{R;\Gamma \vdash r_i : \mathrm{Reg}_{r_i} A_i \qquad \forall V \in \mathcal{V}(r_i) :\ R;\Gamma \vdash V : (A_i, \emptyset) \qquad \dfrac{R;\Gamma \vdash M : (A \xrightarrow{e_1} \alpha, e_2) \qquad R;\Gamma \vdash N : (A, e_3)}{R;\Gamma \vdash M\ N : (\alpha, e)}\ (\textsc{app})}{R;\Gamma \vdash M\ N[\mathcal{V}]_\lambda : (\alpha, e)}\ (\textsc{subst-r})$$

$$\frac{R;\Gamma \vdash r_j : \mathrm{Reg}_{r_j} B_j \qquad \forall U \in \mathcal{U}(r_j) :\ R;\Gamma \vdash U : (B_j, \emptyset) \qquad R;\Gamma \vdash M\ N[\mathcal{V}]_\lambda : (\alpha, e)}{R;\Gamma \vdash M\ N[\mathcal{V}]_\lambda[\mathcal{U}]_\downarrow : (\alpha, e)}\ (\textsc{subst-r})$$

That we can turn into

$$\frac{R;\Gamma \vdash r_j : \mathrm{Reg}_{r_j} B_j \qquad \forall U \in \mathcal{U}(r_j) :\ R;\Gamma \vdash U : (B_j, \emptyset) \qquad R;\Gamma \vdash M : (A \xrightarrow{e_1} \alpha, e_2)}{R;\Gamma \vdash M[\mathcal{U}]_\downarrow : (A \xrightarrow{e_1} \alpha, e_2)}\ (\textsc{subst-r})$$

$$\frac{R;\Gamma \vdash r_j : \mathrm{Reg}_{r_j} B_j \qquad \forall U \in \mathcal{U}(r_j) :\ R;\Gamma \vdash U : (B_j, \emptyset) \qquad R;\Gamma \vdash N : (A, e_3)}{R;\Gamma \vdash N[\mathcal{U}]_\downarrow : (A, e_3)}\ (\textsc{subst-r})$$

$$\frac{R;\Gamma \vdash M[\mathcal{U}]_\downarrow : (A \xrightarrow{e_1} \alpha, e_2) \qquad R;\Gamma \vdash N[\mathcal{U}]_\downarrow : (A, e_3)}{R;\Gamma \vdash (M[\mathcal{U}]_\downarrow)\ (N[\mathcal{U}]_\downarrow) : (\alpha, e)}\ (\textsc{app})$$

$$\frac{R;\Gamma \vdash r_k : \mathrm{Reg}_{r_k} A_k \qquad \forall W \in (\mathcal{V} + \mathcal{U})(r_i) :\ R;\Gamma \vdash W : (C_i, \emptyset) \qquad R;\Gamma \vdash (M[\mathcal{U}]_\downarrow)\ (N[\mathcal{U}]_\downarrow) : (\alpha, e)}{R;\Gamma \vdash (M[\mathcal{U}]_\downarrow)\ (N[\mathcal{U}]_\downarrow)[\mathcal{W}]_\lambda}\ (\textsc{subst-r})$$

- ($\texttt{subst-r}_{\texttt{get}}$) Write $\mathbf{S} = M_1 + \ldots + M_n + []$, with $n \geq 0$. The derivation tree of $P = C[E[\mathrm{get}(r)[\mathcal{V}]_\downarrow]]$ is of the form

$$\cfrac{\cfrac{\cfrac{R;\Gamma \vdash r_i : \mathrm{Reg}_{r_i} A_i \qquad \forall V \in \mathcal{V}(r_i) :\ R;\Gamma \vdash V : (A_i, \emptyset) \qquad R;\Gamma \vdash \mathrm{get}(r) : (A, e)}{R;\Gamma \vdash \mathrm{get}(r)[\mathcal{V}]_\downarrow : (A, e)}\ (\textsc{subst})}{[\ldots]}}{R;\Gamma' \vdash P : (\alpha, e')}$$

  And of the full term :

$$\frac{(\forall i\ : 1 \leq i \leq n)\ R;\Gamma \vdash M_i : (\alpha, e') \qquad R;\Gamma \vdash P : (\alpha, e')}{R;\Gamma \vdash S[P] : (\alpha, e')}\ (\textsc{sum})$$

  To get the first term of the reduct, we just remove the subst rule :

$$\frac{(\forall i\ : 1 \leq i \leq n)\ R;\Gamma \vdash M_i : (\alpha, e') \qquad \cfrac{\cfrac{R;\Gamma \vdash \mathrm{get}(r) : (A, e)}{[\ldots]}}{R;\Gamma' \vdash C[E[\mathrm{get}(r)]] : (\alpha, e')}}{R;\Gamma' \vdash S[C[E[\mathrm{get}(r)]]] : (\alpha, e')}\ (\textsc{sum})$$

  Now if $\mathcal{V}$ is undefined at r we are done. Otherwise, we substitute $\mathrm{get}(r)$ by a value $V \in \mathcal{V}(r)$:

$$\cfrac{\cfrac{\cfrac{R;\Gamma \vdash V : (A, \emptyset)}{R;\Gamma \vdash V : (A, e)}\ (\textsc{sub})}{[\ldots]}}{R;\Gamma' \vdash C[E[V]] : (\alpha, e')}$$

  And we can finally derive

$$\frac{\forall V \in \mathcal{V}_{i_0} :\ R;\Gamma' \vdash C[E[V]] : (\alpha, e') \qquad R;\Gamma' \vdash S[C[E[\mathrm{get}(r)]]] : (\alpha, e')}{R;\Gamma' \vdash S[C[E[\mathrm{get}(r)]]] + \sum_{V \in \mathcal{V}(r)} C[E[V]] : (\alpha, e')}\ (\textsc{sum})$$

Upward references substitutions

- $(\texttt{subst-r'}_{\parallel})$

$$\dfrac{R;\Gamma \vdash M : (\alpha_1, e_1) \qquad \dfrac{R;\Gamma \vdash r_i : \mathrm{Reg}_{r_i} A_i \qquad \forall V \in \mathcal{V}(r_i) :\ R;\Gamma \vdash V : (A_i, \emptyset) \qquad R;\Gamma \vdash N : (\alpha_2, e_2)}{R;\Gamma \vdash N[\mathcal{V}]_{\uparrow} : (\alpha_2, e_2)}\ (\textsc{subst-r})}{R;\Gamma \vdash M \parallel (N[\mathcal{V}]_{\uparrow}) : (\mathbf{B}, e)}\ (\textsc{par})$$

Then we have

$$\dfrac{\dfrac{R;\Gamma \vdash r_i : \mathrm{Reg}_{r_i} A_i \qquad \forall V \in \mathcal{V}(r_i) :\ R;\Gamma \vdash V : (A_i, \emptyset) \qquad R;\Gamma \vdash M : (\alpha_1, e_1)}{R;\Gamma \vdash (M[\mathcal{V}]_{\downarrow}) : (\alpha_1, e_2)}\ (\textsc{subst-r}) \qquad R;\Gamma \vdash N : (\alpha_2, e_2)}{R;\Gamma \vdash (M[\mathcal{V}]_{\downarrow}) \parallel N : (\mathbf{B}, e)}\ (\textsc{par})$$

$$\dfrac{R;\Gamma \vdash r_i : \mathrm{Reg}_{r_i} A_i \qquad \forall V \in \mathcal{V}(r_i) :\ R;\Gamma \vdash V_i : (A_i, \emptyset) \qquad R;\Gamma \vdash (M[\mathcal{V}]_{\downarrow}) \parallel N : (\mathbf{B}, e)}{R;\Gamma \vdash (M[\mathcal{V}]_{\downarrow}) \parallel N[\mathcal{V}]_{\uparrow} : (\mathbf{B}, e)}\ (\textsc{subst-r})$$

- $(\texttt{subst-r'}_{\texttt{app-left}})$

$$\dfrac{R;\Gamma \vdash r_i : \mathrm{Reg}_{r_i} A_i \qquad \forall V \in \mathcal{V}(r_i) :\ R;\Gamma \vdash V : (A_i, \emptyset) \qquad R;\Gamma \vdash M : (A \xrightarrow{e_2} \alpha, e_3)}{R;\Gamma \vdash M[\mathcal{V}]_{\uparrow} : (A \xrightarrow{e_2} \alpha, e_3)}\ (\textsc{subst-r})$$

$$\dfrac{R;\Gamma \vdash (M[\mathcal{V}]_{\uparrow}) \ : (B \xrightarrow{e_1} \alpha, e_2) \qquad R;\Gamma \vdash N : (A, e_3)}{R;\Gamma \vdash (M[\mathcal{V}]_{\uparrow})\ N : (\alpha, e)}\ (\textsc{app})$$

$$\dfrac{R;\Gamma \vdash r_j : \mathrm{Reg}_{r_j} B_j \qquad \forall U \in \mathcal{U}(r_j) :\ R;\Gamma \vdash U : (B_j, \emptyset) \qquad R;\Gamma \vdash (M[\mathcal{V}]_{\uparrow})\ N : (\alpha, e)}{R;\Gamma \vdash (M[\mathcal{V}]_{\uparrow})\ N[\mathcal{U}]_{\lambda} : (\alpha, e)}\ (\textsc{subst-r})$$

That we turn to

$$\dfrac{R;\Gamma \vdash M : (A \xrightarrow{e_1} \alpha, e_2) \qquad \dfrac{R;\Gamma \vdash r_i : \mathrm{Reg}_{r_i} A_i \qquad \forall V \in \mathcal{V}(r_i) :\ R;\Gamma \vdash V : (A_i, \emptyset) \qquad R;\Gamma \vdash N : (A, e_3)}{R;\Gamma \vdash N[\mathcal{V}]_{\downarrow} : (A, e_3)}\ (\textsc{subst-r})}{R;\Gamma \vdash M\ (N[\mathcal{V}]_{\downarrow}) : (\alpha, e)}\ (\textsc{app})$$

$$\dfrac{R;\Gamma \vdash r_k : \mathrm{Reg}_{r_k} C_k \qquad \forall W \in (\mathcal{V} + \mathcal{U})(r_k) :\ R;\Gamma \vdash W : (C_k, \emptyset) \qquad R;\Gamma \vdash M\ (N[\mathcal{U}, \mathcal{V}]_{\downarrow}) : (\alpha, e)}{R;\Gamma \vdash M\ (N[\mathcal{V}]_{\downarrow})[\mathcal{U}, \mathcal{V}]_{\lambda} : (\alpha, e)}\ (\textsc{subt-r})$$

$$\dfrac{R;\Gamma \vdash r_k : \mathrm{Reg}_{r_k} C_k \qquad \forall W \in (\mathcal{U} + \mathcal{V})(r_k) :\ R;\Gamma \vdash W : (C_k, \emptyset) \qquad R;\Gamma \vdash M\ (N[\mathcal{V}]_{\downarrow})[\mathcal{U}, \mathcal{V}]_{\lambda} : (\alpha, e)}{R;\Gamma \vdash M\ (N[\mathcal{V}]_{\downarrow})[\mathcal{U}, \mathcal{V}]_{\lambda}[\mathcal{V}]_{\uparrow} : (\alpha, e)}\ (\textsc{subst-r})$$

- $(\texttt{subst-r'}_{\texttt{app-right}})$: same as $(\texttt{subst-r'}_{\texttt{app-left}})$

- $(\texttt{subst-r'}_{\top})$ Assume that $\mathbf{S} = M_1 + \ldots + M_n + []$, with $n \geq 0$, then we have

$$\dfrac{R;\Gamma \vdash r_j : \mathrm{Reg}_{r_j} A_j \qquad \forall V \in \mathcal{V}(r_j) :\ R;\Gamma \vdash V : (A_j, \emptyset) \qquad R;\Gamma \vdash M : (\alpha, e)}{R;\Gamma \vdash M[\mathcal{V}]_{\uparrow} : (\alpha, e)}\ (\textsc{subst-r})$$

$$\dfrac{(\forall i \ : 1 \leq i \leq n)\ R;\Gamma \vdash M_i : (\alpha, e) \qquad R;\Gamma \vdash M[\mathcal{V}]_{\uparrow} : (\alpha, e)}{R;\Gamma \vdash S[M[\mathcal{V}]_{\uparrow}] : (\alpha, e)}\ (\textsc{sum})$$

We just have to remove the (`subst-r`) rule to get the result :

$$\frac{(\forall i \ : 1 \leq i \leq n) \ R;\Gamma \vdash M_i : (\alpha, e) \qquad R;\Gamma \vdash M : (\alpha, e)}{R;\Gamma \vdash S[M] : (\alpha, e)} \ (\text{SUM})$$

□

4.2.3 Progress

Well-typed normal forms of λ_{cES} may not be a parallel composition of values. Indeed, get(r) itself is an example of a typable normal form which is not a value. The progress theorem states that the only reason for which a term may get stuck is the presence of an orphan read with no corresponding assignment. Normal forms are thus either values, or some application of values together with at least one such stuck read.

Lemma 4.2.7. Progress
Let $R \vdash \mathbf{M} : (A, e)$ be a well-typed term that does not reduce further. Then $\mathbf{M}$ is of the form $\sum_{i=1}^{n}(M_1^i \parallel \ldots \parallel M_{l_i}^i)$ where the M_j^i are either values or terms of the grammar $M_{\text{norm}} ::= \text{get}(r) \mid (M_{\text{norm}} \ V)[\mathcal{V}]_\lambda \mid (V \ M_{\text{norm}})[\mathcal{V}]_\lambda \mid (M_{\text{norm}} \ M_{\text{norm}})[\mathcal{V}]_\lambda$. □

Proof. Lemma 4.2.7
By recurrence on $\mathbf{M}$. First assume that $\mathbf{M}$ is a simple term M :

- If M is a value or $M = \text{get}(r)$, this is true
- If $M = M'[\sigma]$, then there is always a (`subst`) rule that can be applied whatever M' is.
- If $M = M'[\mathcal{V}]_\downarrow$, we can use a (`subst-r`) rule unless $M' = M''[\sigma]$. But then we are back to the previous case. Hence M' reduces, and so does M.
- If $M = P \ Q[\mathcal{V}]_\lambda$, we have the type derivation

$$\frac{\dfrac{R;\vdash P \ Q[\mathcal{V}]_\lambda : (\alpha, e)}{R;\vdash P \ Q : (\alpha, e)} \ (subst - r)}{R;\vdash P : (A \xrightarrow{e_1} \alpha, e_2) \qquad R;\vdash Q : (A, e_3)} \ (app)$$

  If $P \to P'$ by any rule excepted (`subst-r'`$_\top$), $M = E[P]$ with $E = [.] \ Q[\mathcal{V}]_\lambda$ and thus M also reduces. The same applies if Q reduces. Assume now that $P \to P'$ by (`subst-r'`$_\top$), meaning that $P = P'[\mathcal{V}]_\uparrow$. Then we can apply (`subst-r'`$_{\texttt{app-left}}$) to M. Similarly, we can apply (`subst-r'`$_{\texttt{app-right}}$)

if $Q = Q'[\mathcal{V}]_\uparrow$. Thus by induction P and Q are themselves either values, of the form M_{norm}, or a parallel of values and M_{norm} forms. It is immediate that neither P or Q can be of the form $M_1 \parallel M_2$ (they would have to be typed by **B** and coudn't be applied or applied to), thus P and Q are either values or of the form M_{norm}.
Now, if both are values (note that P and Q must be closed terms), the only way for P to have type $A \xrightarrow{e_1} \alpha$ is to be an abstraction, but then M would reduce by (β_v). Thus one of them is of the form M_{norm} and so is M.

- if $M = M'[\mathcal{V}]_\uparrow$ then it reduces by ($\texttt{subst-r'}_\top$).
- if $M = P \parallel Q$, assume that $P \to P'$ by any rule excepted ($\texttt{subst-r'}_\top$), then so would M, taking $E = [.]$ and $C = [.] \parallel Q$. The same holds for Q.

  Now, if P or Q is of the form $P'[\mathcal{V}]_\uparrow$, then M reduces by ($\texttt{subst-r'}_\parallel$). Thus P and Q does not reduce, and by induction, are of the form $\|_i M_i$ with M_i a value or of the form M_{norm}, hence so is M.

Finally, if $\mathcal{M} = \sum M_i$, then $\mathcal{M}$ is normal if and only if each M_i is and the induction hypothesis immediately gives the result. □

4.3 Termination

Our main result is a finitary, interactive proof of strong normalization for λ_{cES}. This section is devoted to the presentation of the problem in the context of references, the explanation of why the existing solutions do not apply to our setting and what we propose instead.

Shortcoming of Existing Solutions Introduced by Tait in 1967 [53], reducibility is a widely used, versatile technique for proving strong normalization of lambda-calculi. The core of this technique is to define inductively on types τ a set $\mathbf{SC}(\tau)$ of well typed terms, called *strongly computable terms*, satisfying a series of properties. One proves that terms in $\mathbf{SC}(\tau)$ are strongly normalizing (Adequacy) and (the most difficult part) that all well typed terms of a type τ are actually in $\mathbf{SC}(\tau)$.

When adapting this technique to a type and effect system, the main difficulty is that the definition is not obviously inductive anymore. To define $\mathbf{SC}(A \xrightarrow{e} \alpha)$, we need to have defined the types of references appearing in e. But e can itself contain a reference of type $A \xrightarrow{e} \alpha$: in the Landin's fixpoint example shown in the introduction of this chapter, the looping term has the type $\texttt{Unit} \xrightarrow{\{r\}} \texttt{Unit}$

while r has the same type. The role of stratification is to induce a well-founded ordering on types so that the definition becomes consistent.

The solution offered by stratification of the type system is however not enough for λ_{cES}. In Boudol [8], where the technique is introduced, concurrency is explicitly controlled by threads themselves that are guaranteed to be the only process in execution during each slice of execution. In λ_{cES}, reduction steps are performed in arbitrary threads such that stores may be affected by others between two atomic steps in a particular thread.

To overcome this issue, for the language presented in Section 4.4.1, Amadio [6] strengthens the condition defining **SC** sets by asking that they also terminate under infinite stores of the form $(r \Leftarrow V_1 \parallel \ldots \parallel r \Leftarrow V_n \parallel \ldots)$ with (V_i) an enumeration of all the elements of $\mathbf{SC}(\alpha)$. In this setting, infinite stores are static top-level constructions: once saturated, they are invariant by any new assignment. In a term $(M_1 \parallel M_2 \parallel S)$ with S being such a store, any memory operation of M_2 is completely invisible to M_1 and one can prove separately the termination of each thread.

However, this solution is not easily transposable to λ_{cES}. First of all, the rule ($\texttt{subst-r}_{\texttt{get}}$) produces all the possible values associated to a store. The corresponding $\text{get}(r)[\mathcal{V}]_{\downarrow}$ would reduce to an infinite sum $\text{get}(r) + \sum_i V_i$ where, even if each summand terminates, there is also for any positive integer n a summand that takes at least n steps to reach normal form. The total sum is not terminating anymore. Secondly, unlike static top-level stores, reference substitutions are duplicated, erased and exchanged in an interactive way between threads.

Our Solution To prove strong normalization of λ_{cES}, we change gears. With explicit substitutions, assignments and reads are a way of exchanging messages between threads or subterms. Apart from the termination of each term in isolation, the key property we need is that threads cannot exchange an infinite amount of messages.

We formalize this condition by strengthening the definition of strongly computable terms. We force them to also be *well-behaved*. A well-behaved term must only emit a finite number of upward substitutions containing strongly computable terms when placed in a "fair" context. A fair context is a context that would only send strongly computable reference substitutions (albeit potentially infinitely many).

These notions are defined in Section 4.3.1, while the strong normalization result is spelled out in Section 4.3.2.

4.3.1 Technical Definitions

Remark 4.3.1. In the following, we do not want to deal with the clumsiness of handling sums of terms everywhere. If a reduction sequence is seen as a tree, where branching points correspond to ($\texttt{subst-r}_{\texttt{get}}$) and the children to all the summands produced by this rule, then by König's lemma it is finite if and only if all its branches are finite. We will thus use the alternative non-deterministic reduction $\to_{\texttt{nd}}$ (Definition 4.1.9), such that a sequence of reductions $\to_{\texttt{nd}}$ corresponds to a branch in the original reduction system. Proving the termination of $\to_{\texttt{nd}}$ is then sufficient, thanks to the König's lemma. In the rest of the chapter, we only consider simple terms (non-sums) and the $\to_{\texttt{nd}}$ reduction, that we will just write $\to$.

The purpose of the following Definition 4.3.2 is to formalize the interaction of a subterm with its context as a play against an opponent that can non-deterministically drop downward substitutions at the top level or absorb upcoming substitutions. This is summarized in the condition (**WB**) of Definition 4.3.5.

Definition 4.3.2. Environment Reduction
Let $\vdash M : (\alpha, e)$ be a well typed term. Let $(\mathcal{V}_i)$ be a sequence of reference substitutions such that $M[\mathcal{V}_i]_\downarrow$ is well typed that we write $\vdash (M, (\mathcal{V}_i))$. We call a $(M, (\mathcal{V}_i))$-reduction a finite or infinite reduction sequence starting from M where each step is either a $\to_{\texttt{nd}}$, or an interaction with the environment defined by the additional rules $M[\mathcal{V}]_\uparrow \to_\uparrow M$ and $M \to_\downarrow M[\mathcal{V}_i]_\downarrow$.

The sets of strongly computable terms are defined by induction on a pair (α, e) of type and effect, with respect to the following well-founded ordering introduced in [8]:

Definition 4.3.3. Type and effect ordering

We define the size of a type $|\alpha|$ by:

$$\begin{array}{lcl} |\texttt{Unit}| & = & 1 \\ |\mathbf{B}| & = & 1 \\ |A \xrightarrow{e} \alpha| & = & |A| + |\alpha| + 1 \end{array}$$

For two types and effect (α_1, e_1) and (α_2, e_2) such that $R \vdash (\alpha_1, e_1)$ and $R \vdash (\alpha_2, e_2)$, $(\alpha_1, e_1) \prec_R (\alpha_2, e_2)$ if and only if:

1. $\texttt{Reg}_R(\alpha_1, e_1) \subsetneq \texttt{Reg}_R(\alpha_2, e_2)$, or

2. $\mathtt{Reg}_R(\alpha_1, e_1) = \mathtt{Reg}_R(\alpha_2, e_2)$ and $|\alpha_1| < |\alpha_2|$

Proposition 4.3.4. [8, Boudol] Well-ordering of type and effect
The $\prec_R$ order is well-founded, that is there is no infinite sequence $(\alpha_i, e_i)_{i\in\mathbb{N}}$ with $\forall i \in \mathbb{N}, (\alpha_{i+1}, e_{i+1}) \prec_R (\alpha_i, e_i)$.

We can now define the notion of strongly computable terms as follows, by induction with respect to the $\prec_R$ order.

Definition 4.3.5. Strongly Computable Terms
The set $\mathbf{SC}_R(\alpha, e)$ of strongly computable terms of type (α, e) is defined as follows:

Base type. Assume that $\alpha = \mathtt{Unit} \mid \mathbf{B}$. Then $M \in \mathbf{SC}_R(\alpha, e)$ if it is

well-typed $R \vdash M : (\alpha, e)$

(SN) Strongly normalizing under reference substitutions: For all $\mathcal{V}$ verifying $\forall r, \mathcal{V}(r)$ defined $\implies$ $r \in e$, $\mathcal{V}(r) \subseteq \mathbf{SC}_R(R(r), \emptyset)$, the term $M[\mathcal{V}]_\downarrow$ is strongly normalizing.

(WB) Well Behaved: For any $(\mathcal{V}_i)$ such that $\vdash (M, (\mathcal{V}_i))$ and verifying $\forall i, \forall r, \mathcal{V}_i(r)$ defined $r \in e, \mathcal{V}_i(r) \subseteq \mathbf{SC}_R(R(r), \emptyset)$, for any $(M, (\mathcal{V}_i))$-reduction $M = M_0 \to \ldots \to M_n \to \ldots$, there exists $n_0 \geq 1$ such that for all $k \geq 1$:

1. If M_{k-1} is of the form $N[\mathcal{U}]_\uparrow$ with $M_{k-1} \to_\uparrow M_k$ then $\forall r, \mathcal{U}(r)$ defined $\implies$ $\mathcal{U} \subseteq \mathbf{SC}_R(R(r), \emptyset)$,
2. If $k \geq n_0$ then $M_{k-1} \to M_k$ is not a $(\to_\uparrow)$ step.

Inductive case. M belongs to $\mathbf{SC}_R(A \xrightarrow{e_1} \alpha, e)$ provided that $R \vdash M : (A \xrightarrow{e_1} \alpha, e)$. Then for all $V \in \mathbf{SC}_R(A, \emptyset)$, we have $M\ V \in \mathbf{SC}_R(\alpha, e \cup e_1)$.

Notation 4.3.6. By abuse of notation, we shall omit in the following one or more of the R, α, e when it is obvious from the context and just write $M \in \mathbf{SC}$. Moreover, we also write $\mathcal{V} \subseteq \mathbf{SC}$ to mean that for all r for which $\mathcal{V}$ is defined, we have $\mathcal{V}(r) \subseteq \mathbf{SC}_R(R(r), \emptyset)$.

Remark 4.3.7. The condition (**SN**) requires terms to be strongly normalizing when put under any finite reference substitution of strongly computable terms. The finiteness is sufficient, thanks to the presence of condition (**WB**). This rather technical condition is the well-behaved requirement: it says that there are at most n_0 $(\to_\uparrow)$ steps.

4.3.2 Strong Normalization for $\lambda_{\mathbf{cES}}$

We are now ready to state and sketch the proof of strong-normalization for λ_{cES}. The easy part is the adequacy result, stated as follows.

Lemma 4.3.8. Adequacy If $M \in \mathbf{SC}(\alpha, e)$ then M is strongly normalizable. □

Proof. Lemma 4.3.8
We will prove additionally by induction on types that for any value type A, there exists $V_\alpha \in \mathbf{SC}(A, \emptyset)$.

- For $\alpha = \mathtt{Unit} \mid \mathbf{B}$, if $M \in \mathbf{SC}(\alpha, f)$ did not terminate, then neither would $M[\mathcal{V}]_\downarrow$. If $\alpha = \mathtt{Unit}$, then $* \in \mathbf{SC}(\mathtt{Unit}, \emptyset)$ is a value populating $\mathbf{SC}(\alpha, \emptyset)$.
- For $\alpha = A \xrightarrow{e_1} \alpha'$, if $M \in \mathbf{SC}(\alpha, e')$ did not terminate, neither would $M\ V$ for $V \in \mathbf{SC}(A, \emptyset)$, which exists since the latter set is not empty by induction hypothesis. For $V_{\alpha'} \in \mathbf{SC}(\alpha', \emptyset)$, then $\lambda x.V_{\alpha'}$ is a value populating $\mathbf{SC}(\alpha, \emptyset)$

□

The heart of our result is the opposite result, the soundness:

Proposition 4.3.9. Soundness
Let $R; x_1 : A_1, \ldots, x_n : A_n \vdash P : (\alpha, e)$. Let σ be a variable substitution with $\mathrm{dom}(\sigma) \subseteq \{x_1, \ldots, x_n\}$ and $\forall x \in \mathrm{dom}(\sigma), \sigma(x) \in \mathbf{SC}_R(A_i, \emptyset)$. Then $P[\sigma] \in \mathbf{SC}_R(\alpha, e)$.

The proof of Proposition 4.3.9 is rather technical and require the introduction of auxiliary lemmas to be worked out. We first give a high-level sketch of some representative cases first, to focus on the important ideas and intuitions. The reader may refer to Section 4.3.3 for a complete proof.

Sketch of the proof of Proposition 4.3.9.
The proof is performed by induction on the structure of the term P. To show how the proof works, we focus on two representative cases, the abstraction and the parallel composition.

Abstraction Let us treat the case $P = \lambda x.M$ with $R, x : A_1 \vdash \lambda x.M : (\alpha, e_1)$. By induction, for any $V \in \mathbf{SC}_R(A_1, \emptyset)$, $e' \supseteq e_1$, $M[x \mapsto V] \in \mathbf{SC}_R(\alpha, e')$. For $e \subseteq \mathrm{dom}(R)$, we want to show that $P \in \mathbf{SC}_R(A_1 \xrightarrow{e_1} \alpha, e)$. Let $\alpha = A_2 \xrightarrow{e_2} \ldots \xrightarrow{e_{n-1}} A_n \xrightarrow{e_n} \beta$ be the expansion of the type of P, where β is either $\mathtt{Unit}$ or $\mathbf{B}$. If we unfold the recursive definition of $\mathbf{SC}$ sets, proving that $P \in \mathbf{SC}(\alpha)$ amounts to check that $\Lambda(P, \mathcal{U}, (N_i)) := P\ V_1\ \ldots\ V_n[\mathcal{U}]_\downarrow$ satisfies (**SN**) and (**WB**) for all strongly computable $V_1, \ldots, V_n$ and $\mathcal{U}$ with suitable types.

The only possible reduction in Λ are either (`subst-r`) ones (excluding (`subst-r`$_{\texttt{get}}$)), or the reduction $P \to \lambda x.(M[\sigma])$. Assume that Λ has an infinite reduction sequence. As (`subst-r`) rules alone are strongly normalizing, then the reduction $P \to (\lambda x.M[\sigma])$ must happen at some point. Similarly, after this reduction, the only possible reductions excluding (`subst-r`) rules are the β_V-reduction $(\lambda x.M[\sigma])\ V_1 \to M[\sigma][x \mapsto V_1]$, followed by the merging $M[\sigma, x \mapsto V_1]$.

To sum up, an infinite reduction sequence starting from Λ must have the form

$$\begin{array}{lll} \Lambda & \to_1 & \Lambda_1 \to_2 \dots \\ & \to_{i_1} & (\lambda x.M[\sigma])\ V_1\ \dots\ V_n \to_{i_1+1} \dots \\ & \to_{i_2} & M[\sigma][x \mapsto V_1]\ V_2\ \dots\ V_n \to_{i_2+1} \dots \\ & \to_{i_3} & M[\sigma, x \mapsto V_1]\ V_2\ \dots\ V_n \to_{i_3+1} \dots \end{array}$$

We omitted a possible bunch of floating reference substitutions for the sake of readability. Reduction steps verify $(\forall i\ : 1 \leq i \leq i_3), i \notin \{i_1, i_2, i_3\} \implies \to_i$ is a (`subst-r`) rule. We can bound Λ_{i_3}: $\Lambda_{i_3} = M[\sigma, x \mapsto V_1]\ V_2\ \dots\ V_n \sqsubseteq M[\sigma, x \mapsto V_1]\ V_2\ \dots\ V_n[\mathcal{V}]_\downarrow$, but by induction hypothesis $M[\sigma, x \mapsto V_1] \in \mathbf{SC}(\alpha, e_1 \cup e)$, thus Λ_{i_3} is strongly normalizing. This contradicts the fact that the reduction sequence (Λ_i) is infinite. Hence, Λ is strongly normalizing. The (**WB**) is proved similarly: before producing any upward substitution, Λ must perform the reduction step $\to_{i_1}, \to_{i_2}$ and $\to_{i_3}$. From there it is bounded by a (**WB**) term, which implies that it is itself (**WB**).

Parallel We now treat a case that shows the usage of the (**WB**) condition. Assume that $P = M_1 \parallel M_2$. Let $\Gamma = x : A_1, \dots, x_n : A_n$. Since $R; \Gamma \vdash P : (\mathbf{B}, e)$, there exists $\alpha_1, \alpha_2, e_1, e_2$ such that $R; \Gamma \vdash (\alpha_1, e_1)$, $R; \Gamma \vdash M_2 : (\alpha_2, e_2)$, $e_1 \cup e_2 \subseteq e$. Take $e' \supseteq e$. By induction, $M_i \in \mathbf{SC}_R(\alpha_i, e')$. We will show that $M_1 \parallel M_2$ is strongly normalizing: as in the previous case, the proof that it is well-behaved follows the same technique, and we focus on strong normalization. Note that for some appropriate $V_1^i, \dots, V_n^i$, $M_i\ V_1^i\ \dots\ V_n^i$ is strongly normalizing. This entails in particular that M_i itself is strongly normalizing (see the proof of Lemma 4.3.8).

Let $\mathcal{V}$ be a reference substitution of strongly computable terms (whose domain is included in e'), we have to prove that $\Lambda = M_1 \parallel M_2[\mathcal{V}]_\downarrow$ is strongly normalizing. Assume that Λ has an infinite reduction. The reducts of Λ have the forms $\Lambda = M_1 \parallel M_2[\mathcal{V}]_\downarrow \to \dots \to M_1^1 \parallel M_2^1 \to \dots \to M_1^n \parallel M_2^n \to \dots$ where we omitted possible reference substitutions at the top level.

While each M_i do terminate in isolation as strongly computable terms, the possibility of an infinite exchange of substitutions prevent us from using (**SN**) directly to deduce the strong normalization of P. This is the precise role of (**WB**): the sequence of reductions from M_i^k to M_i^{k+1} inside its context can be mapped to an environment reduction of M_i. The environment abstracts the role of the adverse thread M_{1-i} which can generate new downward reference substitutions at the top level. We adopt the following strategy :

1. Use (**WB**) to show that the exchange of substitutions between the two threads M_1 and M_2 must come to an end, and that all the exchanged substitutions are strongly computable.

2. For each M_i, gather all the substitutions (a finite number according the previous step) it receives during the reduction of Λ and merge them into one substitution $\mathcal{X}_i$

3. Show that after a finite number k of steps, when the two threads do not exchange reference substitutions anymore, we can bound each reduct M_i^k inside Λ by a reduct of $M_i[\mathcal{X}]_{\downarrow}$.

Since the bounding terms are strongly normalizing by (**SN**), so are the M_i^ks by Proposition 4.1.18, and the reduction of Λ must be finite from this point. □

Putting together Proposition 4.3.9 (with $n = 0$) and Lemma 4.3.8, we can easily prove the strong normalization result for λ_{cES}.

Theorem 4.3.10. Termination
Let $R \vdash M : (\alpha, e)$ be a well-typed closed term. Then M is strongly normalizing.

Proof. Theorem 4.3.10
Let $R \vdash M : (\alpha, e)$. By Proposition 4.3.9, $M \in \mathbf{SC}(\alpha, e)$. By Lemma 4.3.8, M is strongly normalizing. □

Since the reduction is locally confluent, we deduce the confluence of the language.

Corollary 4.3.11. Confluence
The reduction is confluent on typed terms.

Proof. Corollary 4.3.11
By Newman's lemma. □

4.3.3 Proof of Proposition 4.3.9

We start by giving an explicit (non inductive) characterization of strongly computable terms:

Lemma 4.3.12. Characterization
Let

- $\alpha = A_1 \xrightarrow{e_1} \ldots \xrightarrow{e_{n-1}} A_n \xrightarrow{e_n} \beta$ with $\beta = \text{Unit} \mid \mathbf{B}$
- $\vdash M : (\alpha, e)$
- $V_i \in \mathbf{SC}(A_i, \emptyset)$
- $\mathcal{U} \subseteq \mathbf{SC}$
- $f = e \cup e_1 \cup \ldots \cup e_n$

with $\text{dom}(\mathcal{U}) \subseteq f$. We define

$$\Lambda(M, \mathcal{U}, (V_i)) = M\ V_1\ \ldots\ V_n[\mathcal{U}]_\downarrow$$

Then $M \in \mathbf{SC}(\alpha, e)$ if and only if $\Lambda(M, \mathcal{U}, (V_i))$ is (**SN**) and $\Lambda(M, \bot, (V_i))$ is (**WB**) for all $\mathcal{U}, (V_i)$ satisfying the above conditions. In the following, we may conveniently omit some of the parameters $(M, \mathcal{U}, (V_i))$ of Λ.

Proof. By induction on types. □

Then next series of lemmas aims to prove an inclusion relation between strongly computable terms, namely that $e \subseteq e' \implies \mathbf{SC}_R(\alpha, e) \subseteq \mathbf{SC}_R(\alpha, e')$. This result is stated in Lemma 4.3.16. The idea behind is that adding possible reference substitutions in $\Lambda(M)$ for a term $M \in \mathbf{SC}_R(\alpha, e)$ does not change the normalization (or well-behaving), as either these references are already accounted for by some effect in α, or they do not belong to $\mathtt{Eff}_R(\alpha, e)$ and the corresponding substitutions can not be used at all by M.

The following lemma shows that the effects appearing in the type of a term determine which reference the term may act on.

Lemma 4.3.13. Let $R; \Gamma \vdash M : (\alpha, e)$, then any occurrence $\text{get}(r)$ inside M verifies $r \in \mathtt{Eff}(\alpha, e)$.

Proof. By induction on the typing derivation. □

The following lemma holds the technical content of the proof of Lemma 4.3.16. In Section 4.1.4, we introduced preorders $\sqsubseteq, \sqsubseteq_{\mathcal{V}}$ for comparing the possible behaviors on terms. We showed in Proposition 4.1.18 that if $M \sqsubseteq N$, then

everything that can be done by M can be mimicked in N. Here, we prove a converse result in a very specific case. We take a term $M \sqsubseteq_{\mathcal{V}} N$, and make the additional assumption that for any $\mathrm{get}(r)$ in M, $r \notin \mathrm{dom}(V)$. In others words, $\mathcal{V}$ is composed of reference assignments that can never be used by M. In this case, then the converse assertion about behaviors also holds, that is everything that can be done by N can also be done by M, as the additional substitutions in N can not be used for at all.

Lemma 4.3.14. Let $R \vdash M : (\beta, e)$ a well-typed term of λ_{cES} with $\beta = \mathtt{Unit} \mid \mathbf{B}$. Let $\mathcal{V}$ be a reference substitution such that $\mathrm{dom}(\mathcal{V}) \cap e = \emptyset$. Assume that $M \sqsubseteq_{\mathcal{V}} N$. Then if $N \to N'$, there exists M' such that $M \to^n M'$ with $M' \sqsubseteq_{\mathcal{V}} N'$. If $\to$ is not a (`subst-r`) rule, or is a ($\mathtt{subst\text{-}r}_{\mathtt{get}}$) rule, then $n \geq 1$.

Proof. The proof is similar to Proposition 4.1.18: most of the developments are totally symmetric in M and N. The only different case is the case of the ($\mathtt{subst\text{-}r}_{\mathtt{get}}$). This means that a $\mathrm{get}(r)[\mathcal{V}, \mathcal{W}]_{\downarrow}$ is reduced in N, while only $\mathrm{get}(r)[\mathcal{W}]_{\downarrow}$ can be formed in M. However, by Lemma 4.3.13, $r \in e$, thus the value substitution the first get is $V \in (\mathcal{V}, \mathcal{W})(r) = \mathcal{W}(r)$ by the hypothesis on the domain of $\mathcal{V}$. Then this substitution may also be performed on $\mathrm{get}(r)[\mathcal{W}]_{\downarrow}$ in M. □

Corollary 4.3.15. Let $R \vdash M : (\beta, e)$ a well-typed term of λ_{cES} with $\beta = \mathtt{Unit} \mid \mathbf{B}$. Let $\mathcal{V}$ be a reference substitutions such that $\mathrm{dom}(\mathcal{V}) \cap e = \emptyset$. Assume that $M \sqsubseteq_{\mathcal{V}} N$. Then M is strongly normalizing if and only if N is. Moreover, M is (**WB**) if and only if N is.

Proof. The proof is identical to the proof of Corollary 4.1.19. □

We can now states our monotonicity property about strongly computable sets.

Lemma 4.3.16. Let $M \in \mathbf{SC}_R(\alpha, e)$, then for any $e', e \subseteq e' \subseteq \mathrm{dom}(R)$, $M \in \mathbf{SC}_R(\alpha, e')$.

Proof. Using the characterization of Lemma 4.3.12, we must show that any $\Lambda(M, \mathcal{V}, (V_i))$ with $\mathcal{V}$ and (V_i) adapted to $\mathbf{SC}_R(\alpha, e')$ is (**SN**) (and (**WB**) for $\mathcal{V} = \bot$). Let $f = \mathrm{dom}(\mathcal{V}) \setminus e$, by projecting $\mathcal{V}$ on e, we have that $\Lambda(M, \mathcal{V} \restriction_e, (V_i)) \sqsubseteq_{\mathcal{V}\restriction_f} \Lambda(M, \mathcal{V}, (V_i))$. But $\Lambda(M, \mathcal{V} \restriction_e, (V_i))$ is (**SN**) because $M \in \mathbf{SC}_R(\alpha, e)$. By Corollary 4.3.15, so is $\Lambda(M, \mathcal{V}, (V_i))$. The (**WB**) condition is treated similarly. □

The following lemma gives a list of technical properties:

Lemma 4.3.17. Auxiliary results for soundness
Let $M \in \mathbf{SC}_R(\alpha, e)$ and $\mathcal{V}$ with $r \in \mathrm{dom}(V) \implies \mathcal{V} \subseteq \mathbf{SC}_R(R(r), \emptyset$.

1. Consider $\Lambda(M, \mathcal{U}, (U_i))$ for appropriate $\mathcal{U}$ and (U_i) (as given in Lemma 4.3.12). The possible immediate reducts of Λ may have the following form:

   (a) $M\ V_1\ \ldots\ V_{n-1}[\mathcal{V}]_\lambda$ $(V_n[\mathcal{V}]_\downarrow)$ if the substitution is pushed down

   (b) $M'\ V_1\ \ldots\ V_n[\mathcal{V}]_\downarrow$ with $M \to M'$ if a reduction happens inside M

   (c) $N\ V_2\ \ldots\ V_n[\mathcal{V}]_\downarrow$ if $M = \lambda x.M'$ interacts with V_1 by a β_V reduction.

   As each V_i is inert, and propagating of reference substitutions is a terminating process, an infinite reduction sequence of Λ must at some point either reduce inside M as in (b), propagate the substitution $\mathcal{V}$ in M or perform a β_V reduction as in (c).

2. Let $R, x_1 : A_1, \ldots, x_n : A_n \vdash N : (\alpha, e)$, σ be a variable substitution with $\mathrm{dom}(\sigma) \subseteq \{x_1, \ldots, x_n\}$ and assume $N[\sigma] \to M$. Then $N[\sigma]_\downarrow \in \mathbf{SC}_R(\alpha, e)$.

3. $M[\mathcal{V}]_\downarrow \in \mathbf{SC}$

4. $M[\mathcal{V}]_\uparrow \in \mathbf{SC}$

5. If $M \to M'$, then $M' \in \mathbf{SC}$

6. $\mathcal{V}\{\sigma\} \subseteq \mathbf{SC}$

Proof. 1. The V_i are inert, and if one does not perform a reduction involving M by either reducing inside, propagating reference substitutions or performing a β_V reduction, then the only possible reduction are the propagation of $\mathcal{V}$, which terminates.

2. Consider an infinite reduction of $\Lambda(N[\sigma])$. By the previous point, the subterm M' must be reduced in at some point. Since the variable substitution forbids any other reduction than propagating itself first, M is the only possible reduct. Then, then term we obtain after reducing $N[\sigma]$ to M in head position is bounded by a reduct of $\Lambda(M)$ which is (**SN**) by hypothesis on MM. Thus $\Lambda(M')$ has no infinite reduction sequence and $\Lambda(M')$ is (**SN**).

   Similarly, consider (for suitable $(\mathcal{V}_i)$) a $(M', (\mathcal{V}_i))$ reduction. If $N[\sigma]$ is never reduced, then no upward substitution is ever produced. If $N[\sigma]$ is reduced at some point, it is reduced to M and thus produces a finite amount of **SC** upward substitutions from here since M is (**WB**). Hence $N[\sigma]$ is (**WB**).

3. We have the following relation: $\Lambda(M[\mathcal{V}]_\downarrow, \mathcal{U}, (N_i)) \sqsubseteq \Lambda(M, (\mathcal{U}, \mathcal{V}), (N_i))$ (for appropriate $\mathcal{U}, (N_i)$). As the latter is (**SN**) since M is **SC**, so is the former. For (**WB**), we can easily map a $(M', (\mathcal{W}'_i))$ reduction to a $(M, (\mathcal{W}_i))$ by just appending $\mathcal{V}$ to $(\mathcal{W}_i)$ and start with a $\rightarrow_i$ reduction. Since $M \in \mathbf{SC}$, M' is (**WB**).

4. Let $M' = M[\mathcal{V}]_\uparrow$. We proceed by induction on the length of the type α. For base type, it is clear that M' is (**WB**) if and only if M is (it either produces the same upward substitutions as M, or $\mathcal{V}$) and $M'[\mathcal{U}]_\downarrow$ has exactly the same reductions as $M[\mathcal{U}]_\downarrow$ excepted for commutations of upward and downward substitutions, and a possible ($\texttt{subst}_\top$). Since M is (**SN**), so is M'.

   Now, for $\alpha = A \xrightarrow{e_1} \alpha'$, consider an infinite reduction of $\Lambda(M', \mathcal{U}, (V_i))$. If the upward substitution is never moved upward, we can map this to an infinite reduction of $\Lambda(M, \mathcal{U}, (N_i))$ for the same reasons as above, but the latter is (**SN**). Thus at some point the upward substitution must go up, so that the head term becomes $M''\ (V'_1[\mathcal{V}]_\downarrow)[\mathcal{W}]_\lambda[\mathcal{V}]_\uparrow$ where $\mathcal{W}$ is either $\mathcal{V}, \mathcal{U}$ or $\mathcal{V}$, M'' is either a reduct of $M[\mathcal{U}]_\downarrow$ or a reduct of M, and V'_1 is either V or $V_1[\mathcal{U}]_\downarrow$.

   By hypothesis, $M \in \mathbf{SC}_R(A \xrightarrow{e_1} \alpha', e)$, thus $P = M''\ V_1 \in \mathbf{SC}_R(\alpha', e \cup e_1)$. By induction hypothesis, $P[\mathcal{V}]_\uparrow \in \mathbf{SC}_R(\alpha', e \cup e_1)$. We can then bound the considered reduct of Λ by a reduct of $\Lambda(P[\mathcal{V}]_\uparrow, (\mathcal{U}, \mathcal{V}), (V_2, \ldots, V_n))$ which is strongly normalizing.

5. This is straightforward from the definition of **SC** sets.

6. Combining 2. and 4., as for $r \in \mathrm{dom}(U), V \in \mathcal{U}(r), V\{\sigma\}$ is the immediate reduct of $U[\sigma]$ using a ($\texttt{subst}$) rules.

□

We can finally give a complete proof of soundness:

Proof. Proposition 4.3.9
Assume that $R; x_1 : A_1, \ldots, x_n : A_n \vdash M : (\alpha, e)$. We have to prove that for any substitution σ with $\mathrm{dom}(\sigma) = \{x_1, \ldots, x_n\}$ and $(\forall i \ : 1 \leq i \leq n),\ \sigma(x_i) \in \mathbf{SC}_R(A_i, \emptyset)$, then $M[\sigma] \in \mathbf{SC}_R(\alpha, e)$. We perform the proof by induction on the structure of M, and use the characterization of Lemma 4.3.12 to prove that terms are strongly computable.

- $M = x : x[\sigma]$ reduces to $\sigma(x) \in \mathbf{SC}_R(A_i, \emptyset) \subseteq \mathbf{SC}_R(A_i, e)$ (by hypothesis and), and we apply 2. of Lemma 4.3.17.

- $M = * : *[\sigma] \to *$, and we also apply 2. of Lemma 4.3.17. $*$ is trivially (**WB**) and (**SN**).

- $M = \lambda y.M' : M[\sigma] \to \lambda x.(M'[\sigma])$. We have $\alpha = B \xrightarrow{e_1} \alpha'$. By 2. of Lemma 4.3.17, it suffices to prove that $\lambda x.(M'[\sigma])$ is strongly computable. By 1. of Lemma 4.3.17, for a reduction of a corresponding Λ to be infinite, a β_V reduction between $\lambda x.M'[\sigma]$ and V_1 must occur. The head term has then the form $M'[\sigma][x \mapsto V][\mathcal{W}]_\downarrow$ with $\mathcal{W}$ being either $\bot$ or $\mathcal{V}$. $M'[\sigma][x \mapsto V] \to M'[\sigma, x \mapsto V]$ which is in $\mathbf{SC}_R(B, e_1 \cup e)$ by induction hypothesis.

- $M = \mathrm{get}(r) : \mathrm{get}(r)[\sigma] \to \mathrm{get}(r)$. By 2. of Lemma 4.3.17, it suffices to show that $\mathrm{get}(r) \in \mathbf{SC}_R(\alpha, e)$. Consider an infinite reduction of $\Lambda(\mathrm{get}(r))$. By 1. of Lemma 4.3.17, the $\mathrm{get}(r)$ must be reduced at some point and from this point it is either replaced by $V \in \mathcal{V}(r) \in \mathbf{SC}_R(R(r_i) = \alpha, \emptyset) \subseteq \mathbf{SC}_R(\alpha, e)$ and from this point the reduct is bounded by a reduct of a strongly normalizing $\Lambda(V, \mathcal{V}, (V_{ii}))$, or it just get rid of the downward substitution $\mathcal{V}$ and from this point the only possible reductions left are propagation of the initial $\mathcal{V}$ which must terminates.

- $M = M'[\tau]$, then $M[\sigma] \to M'[\tau, \sigma]$. M' has a typing judgement of the form $x_1 : A_1, \ldots, x_n : A_n, y_1 : B_1, \ldots, y_m : B_m \vdash M' : (\alpha, e)$. By induction, $\tau(y_k)[\sigma] \in \mathbf{SC}_R(B_i, \emptyset)$. As $\tau(y_k)[\sigma] \to \tau(y_k)\{\sigma\}$, By 5. of Lemma 4.3.17, the latter is in $\mathbf{SC}_R(B_i, \emptyset)$. Thus we can apply the induction hypothesis on M': $M'[\tau, \sigma] \in \mathbf{SC}_R(\alpha, e)$.

- $M = M'[\mathcal{U}]_\downarrow : M[\sigma] \to M'[\sigma][\mathcal{U}\{\sigma\}]_\downarrow$. By induction, $M'[\sigma] \in \mathbf{SC}_R(\alpha, e)$ and by 6. of Lemma 4.3.17 , $r \in \mathrm{dom}(\mathcal{V}) \implies V \in \mathcal{U}\{\sigma\}(r) \in \mathbf{SC}_R(R(r_i), \emptyset)$. We conclude by parts 3. and 2. of Lemma 4.3.17.

- $M = M'[\mathcal{V}]_\uparrow$: We proceed in the same way, by parts 2., 3., and 4. of Lemma 4.3.17.

- $M = M_1\ M_2[\mathcal{V}]_\lambda : M[\sigma] \to M' = (M_1[\sigma])\ (M_2[\sigma])[\mathcal{V}\{\sigma\}]_\lambda$. By inversion of typing rules, $R; \Gamma \vdash M_1 : (A \xrightarrow{e_1} \alpha, f_1)$ and $R; \Gamma \vdash M_2 : (A, f_2)$ with $f_1 \cup e_1 \cup f_2 \subseteq e$. By induction and 6. of Lemma 4.3.17, $M_1[\sigma] \in \mathbf{SC}_R(A \xrightarrow{e_1} \alpha, f_1) \subseteq \mathbf{SC}_R(A \xrightarrow{e_1} \alpha, e)$, $M_2[\sigma] \in \mathbf{SC}_R(A, f_2) \subseteq \mathbf{SC}_R(A, e)$, and $r \in \mathrm{dom}(\mathcal{V}) \implies \mathcal{V}\{\sigma\} \subseteq \mathbf{SC}_R(R(r), \emptyset)$.

  Assume the existence of an infinite reduction of sequence $\Lambda(M', \mathcal{U}, (V_i))$ for appropriate $\mathcal{U}$ and (V_i). Assume that no β_V reduction between a reduct of $M_1[\sigma]$ and $M_2[\sigma]$ happens. That is, the reducts of $\Lambda(M')$ are of the form $M_1^k\ M_2^k[\mathcal{W}_k]_\lambda\ V_1\ \ldots\ V_n$. Notes that the M_i^k are not exactly reducts of $M_i[\sigma]$: indeed, the two subterms may freely exchange reference

substitutions. This is where the (**WB**) comes into play: if we abstract the context of the M_1^ks, that is [.] $M^k[\mathcal{W}_k]_\lambda\ V_1\ \ldots\ V_n$, which can drop some downward substitutions coming from the reduction of M_2^k at any time, this precisely corresponds to an environment reduction. That is, the sequence $M_1[\sigma], M_1^1, \ldots, M_1^k, \ldots$ is precisely an environment reduction for some sequence of substitution $\mathcal{X}_1^k$. This is also the case form the point of view of $M_2[\sigma]$. As the downward reference substitutions dropped on M_i^k corresponds either to $\mathcal{U}$ or to a substitution previously generated by M_{1-i}^k, they are all strongly computable. Thus the environment reduction satisfies the required hypothesis to apply (**WB**), and we deduce that only a finite number of substitutions is exchanged between the M_1^ks and the M_2^ks. After $k \geq n$ steps of reduction for some n, the M_1^ks reduces in isolation. If we take the trace of all the substitutions it received from the environment before this point, and merge them into one big substitution $\mathcal{X}_1$, then we can see that M_1^n is bounded by a reduct of $M_1[\sigma][\mathcal{X}_1]_\downarrow$. Since $M_1[\sigma]$ is strongly computable, in particular $M_1[\sigma][\mathcal{X}_1]_\downarrow$ is strongly normalizing (cf Lemma 4.3.8). Thus at some point, M_1^k can not reduce anymore. We can do the same for $M_2[\sigma]$, and we see that for the reduction of Λ to be infinite, a β_v reduction must eventually happens between a M_1^k and M_2^k.

Hence there is a k_0 such that $M_1^{k_0} = (\lambda x.P)$ and $M_2^{k_0} = U$, and the reduct of Λ is of the form $(\lambda x.P)\ U[\mathcal{Y}]_\lambda\ V_1\ \ldots\ V_n$ where $\mathcal{Y}$ is composed of substitutions of $\mathcal{X}_1$, $\mathcal{X}_2$ and $\mathcal{U}$, which are all strongly computable. The β_V reduction leads to the term $(P[x \mapsto U][\mathcal{Y}]_\downarrow)\ V_1\ \ldots\ V_n$. This term is bounded by a reduct of $M_1[\sigma]\ U\ V_1\ \ldots\ V_n[\mathcal{X}_1, \mathcal{X}_\in, \mathcal{U}]_\downarrow$. But the latter term is strongly normalizing, because $M_1[\sigma] \in \mathbf{SC}_R(A \xrightarrow{e_1} \alpha, e)$ and $U \in \mathbf{SC}_R(A, e)$ by 3. and 4. of Lemma 4.3.17. Hence Λ has no infinite sequence reduction, and is (**SN**).

We apply a similar reasoning to prove that it is (**WB**).

- $M = M_1 \parallel M_2$: this is handled as in the previous case (this case is simpler as there is not β_V reduction involved).

□

4.4 λ_{C} and $\lambda_{\mathbf{cES}}$

In this section, we study the relation between λ_{C} and λ_{cES}. Although they are close one to another, λ_{C} is not a proper sub-language of λ_{cES} because of the disappearance of stores and of the set(r, V) construct in λ_{cES}. We can however

give a straightforward translation of λ_{C} in λ_{cES} (Definition 4.4.2). We prove a simulation theorem for this translation, with respect to the non-deterministic version of the reduction of λ_{cES} (cf Definition 4.1.9). Since λ_{cES} features explicit non-deterministic sums and a confluent reduction on sums, while λ_{C} has a non-deterministic reduction defined on simple terms, switching to the non-deterministic version of the reduction of λ_{cES} seems to be the natural way to relate the two languages through a simulation.

A simulation between non-deterministic languages is however weaker, in some sense, than in a setting where both languages are confluent. To explain why, take two ARS $(X, \to_a)$ and $(X, \to_b)$ with the same set of terms (to simplify, we take a special case of a translation where both languages are the same and the translation is the identity). We also assume that the normal forms in the two ARS are the same, and that they are integers, representing the result of a computation. Assume there is a simulation between the two: if $x \to_a^* x'$, then $x \to_b^* x'$. In particular, this simulation ensures that if x reduces by $\to_a$ to a normal form, $x \to_a^* n \in \mathbb{N}$, then x reduces by $\to_b$ to the same normal form: $x \to_b^* n$. If both ARS are confluent, by unicity of normal forms, this is in particular the only possible normal form that x can reach by $\to_b$ reductions: if $x \to_b^* n' \in \mathbb{N}$, this implies $n = n'$. On the other hand, if $(X, \to^b)$ is not confluent, then nothing prevents x to reduces to many other integers in addition to n. The simulation is then not completely satisfying anymore, as it does not strongly support the claim that the two systems are behaving in the same way. If one chose a meaningless but permissive reduction (for example $\to^b = X^2 \setminus \{(n,t) \mid (n,t) \in \mathbb{N} \times X\}$) then the ARS $(X, \to_b)$ vacuously simulates $(X, \to_a)$ for any choice of $\to_a$. Simulation states that $\to_b$ can do as much as $\to_a$, but it does not prevent it from doing *much more*.

To strengthen the claim that the two languages do behave similarly, we show an adequacy theorem for typed terms, which role is precisely to show that λ_{cES} can not do much more than λ_{C}. If one looks at the possible results of a computation and discards pathological terms to only keep (parallel of) values, these values are essentially the same in for a term of λ_{C} and its translation in λ_{cES}.

We start with a presentation of λ_{C} based on [41].

4.4.1 The concurrent λ-calculus $\lambda_{\mathbf{C}}$

λ_{C} is a call-by-value λ-calculus extended with:

- a notion of threads and an operator $\|$ for parallel composition of threads,

- two terms $\mathrm{set}(r, V)$ and $\mathrm{get}(r)$, to respectively assign a value to and read from a reference,
- special threads $r \Leftarrow V$, called stores, accounting for assignments.

When $\mathrm{set}(r, V)$ is reduced, it turns to the unit value $*$ and produces a store $r \Leftarrow V$ making the value available to all the other threads. A corresponding construct $\mathrm{get}(r)$ is reduced by choosing non deterministically a value among all the available stores. For example, assuming some support for basic arithmetics consider the program $(\lambda x.x + 1)\ \mathrm{get}(r) \parallel \mathrm{set}(r, 0) \parallel \mathrm{set}(r, 1)$. It consists of 3 threads: two concurrent assignments $\mathrm{set}(r, 0)$ and $\mathrm{set}(r, 1)$, and an application $(\lambda x.x + 1)\,\mathrm{get}(r)$. This programs admits two normal forms depending on which assignment "wins": the term $1 \parallel * \parallel * \parallel r \Leftarrow 0 \parallel r \Leftarrow 1$ and the term $2 \parallel * \parallel * \parallel r \Leftarrow 0 \parallel r \Leftarrow 1$. In this language, the stores are global and cumulative: their scope is the whole program, and each assignment adds a new binding that does not erase the previous ones. Reading from a store is a non deterministic process that chooses a value among the available ones. References are able to handle an unlimited number of values and are understood as a typed abstraction of possibly several concrete memory cells. This feature allows λ_{C} to simulate various other calculi with references such as variants with dynamic references or communication [41]. The language is endowed with an appropriate type and effects system ensuring termination.

Terms

-values	V	::=	$x \mid * \mid \lambda x.M$
-terms	M	::=	$V \mid M\ M \mid \mathrm{get}(r) \mid \mathrm{set}(r, V) \mid M \parallel M$
-stores	S	::=	$r \Leftarrow V \mid (S \parallel S)$
-programs	P	::=	$M \mid S \mid (P \parallel P)$

Reduction

As in λ_{cES}, the $\parallel$ operator is subject to structural rules, namely associativity and commutativity, given in Table 4.4. Once again, contexts are similar to the ones of λ_{cES}: we can reduce in any thread at top-level, and the reduction is an usual call-by-value one, excepted that we chose a version which is neither left-to-right nor right-to-left Table 4.5. Reductions rule are given in Table 4.6. There are three of them:

- β_v is a call-by-value β-reduction rule
- (`get`) replaces a $\mathrm{get}(r)$ occurrence by V, where $r \Leftarrow V$ is an available assignment at top-level

- (`set`) generates a new assignment $r \Leftarrow V$ at top-level

Notation 4.4.1. For terms M, V of λ_{C}, we denote by $M\{x/V\}$ the usual implicit substitution:

- $x\{x/V\} = V$
- $y\{x/V\} = y$ when $y \neq x$
- $\lambda y.M\{x/V\} = \lambda y.(M\{x/V\})$
- $(M\ N)\{x/V\} = (M\{x/V\})\ (N\{x/V\})$
- $\mathrm{get}(r)\{x/V\} = \mathrm{get}(r)$
- $\mathrm{set}(r, U)\{x/V\} = \mathrm{set}(r, U\{x/V\})$

$$\begin{array}{rcl} P \parallel P' & = & P \parallel P' \\ (P \parallel P') \parallel P'' & = & P \parallel (P' \parallel P'') \end{array}$$

Table 4.4: Structural Rules of λ_{C}

$$\begin{array}{rcl} E & ::= & [.] \mid E\ M \mid M\ E \\ C & ::= & [.] \mid (C \parallel P) \mid (P \parallel C) \end{array}$$

Table 4.5: Evaluation Contexts of λ_{C}

Reduction rules			
(β_v)	$C[E[(\lambda x.M)\ V]$	$\rightarrow$	$C[E[M\{x/V\}]]$
(`get`)	$C[E[\mathrm{get}(r)]] \parallel r \Leftarrow V$	$\rightarrow$	$C[E[V]] \parallel r \Leftarrow V$
(`set`)	$C[E[\mathrm{set}(r, V)]]$	$\rightarrow$	$C[E[*]] \parallel r \Leftarrow V$

Table 4.6: Reduction rules of λ_{C}

Typing rules

The stratification conditions and subtyping rules are exactly the same as for λ_{cES}. We only give the typing rules of terms in Figure 4.4.

4.4.2 Simulation

We start by defining a translation of a term of λ_{C} in λ_{cES}. $\mathrm{set}(r, V)$, which is not a construct of λ_{cES}, is translated to $*[\mathcal{V}]_{\uparrow}$ where $\mathcal{V} = \{r \mapsto [V]\}$. Stores should be converted to reference substitutions. However, they may be placed at many intermediate locations in the structure of a term. We chose a canonical representation, where the downward reference substitutions corresponding to stores are pushed down to the maximum. Then, as reference substitutions vanishes on values, the only remaining ones are the λ-substitutions at application, and the one that are followed by a $\mathrm{get}(r)$.

$$\frac{R \vdash \Gamma, x : A}{R; \Gamma, x : A \vdash x : (A, \emptyset)}\ (var) \qquad \frac{R \vdash \Gamma}{R; \Gamma \vdash * : (\mathtt{Unit}, \emptyset)}\ (unit) \qquad \frac{R \vdash \Gamma \qquad r : A \in R}{R; \Gamma \vdash r : Reg_r A}\ (reg)$$

$$\frac{R; \Gamma, x : A \vdash M : (\alpha, e)}{R; \Gamma \vdash \lambda x.M : (A \xrightarrow{e} \alpha, \emptyset)}\ (lam) \qquad \frac{R; \Gamma \vdash M : (A \xrightarrow{e_1} \alpha, e_2) \qquad R; \Gamma \vdash N : (A, e_3)}{R; \Gamma \vdash M\ N : (\alpha, e_1 \cup e_2 \cup e_3)}\ (app)$$

$$\frac{R; \Gamma \vdash \mathrm{Reg}_r A}{R; \Gamma \vdash \mathrm{get}(r) : (A, \{r\})}\ (get) \qquad \frac{R; \Gamma \vdash r : \mathrm{Reg}_r A \qquad R; \Gamma \vdash V : (A, \emptyset)}{R; \Gamma \vdash set(r, V) : (\mathrm{Unit}, \{r\})}\ (set)$$

$$\frac{r : A \in R \qquad R; \Gamma \vdash V : (A, \emptyset)}{R; \Gamma \vdash r \Leftarrow V : (\mathbf{B}, \emptyset)}\ (store)$$

$$\frac{R; \Gamma \vdash P : (\alpha, e) \qquad R; \Gamma \vdash S : (\mathbf{B}, \emptyset)}{R; \Gamma \vdash P \parallel S : (\alpha, \emptyset)}\ (par1) \qquad \frac{i = 1, 2 \qquad R; \Gamma \vdash P_i : (\alpha_i, e_i)}{R; \Gamma \vdash P_1 \parallel P_2 : (\mathbf{B}, e_1 \cup e_2)}\ (par2)$$

Figure 4.4: Typing rules for λ_{C}

Definition 4.4.2. Translation of λ_{C} in λ_{cES}
Let M be a term and S be a store of λ_{C}. S can be written as

$$S = r_1 \Leftarrow V_1^1 \parallel \ldots \parallel r_1 \Leftarrow V_{k_1}^1 \parallel \ldots \parallel r_n \Leftarrow V_1^n \parallel \ldots \parallel r_n \Leftarrow V_{k_n}^n$$

Let $\mathcal{V}_S : r_i \mapsto [V_1^i, \ldots, V_{k_i}^i]$. We allow S to be the empty store ϵ, in which case $\mathcal{V}_S$ is the nowhere defined function. We define the translation of M under S by

- $\overline{*}^S = *$
- $\overline{\overline{\lambda x.M'}}^S = \lambda x.\overline{M'}^S$
- $\overline{\overline{\mathrm{set}(s, V)}}^S = *[s \mapsto [\overline{V}^S]]_{\uparrow}$
- $\overline{\overline{N\ N'}}^S = \overline{N}^S\ \overline{N'}^S[\mathcal{V}_S]_{\lambda}$
- $\overline{\overline{\mathrm{get}(s)}}^S = \mathrm{get}(s)[\mathcal{V}_S]_{\downarrow}$ or just $\mathrm{get}(s)$ if $S = \epsilon$
- $\overline{\overline{N \parallel N'}}^S = \overline{N}^S \parallel \overline{N'}^S$

In fact, $\overline{M}^S$ is the normal form reached from $M[\mathcal{V}_S]_{\downarrow}$ using only downward structural rules (that is, all (`subst-r`) rules excepted ($\mathtt{subst\text{-}r}_{\mathtt{get}}$)). For any program $P = M \parallel S$, we define $\overline{P} = \overline{M}^S$. The case where P does not contain any store is considered as if $S = \epsilon$.

Since the type systems of λ_{cES} and λ_{C} are very close, the translation preserves the fact of being well-typed:

Proposition 4.4.3. Preservation of typing
Let P be a well-typed program of λ_{C}. Then $\overline{P}$ is a well-typed program of λ_{cES}.

The reduction of λ_{cES} has the limitation of not being able to reduce under abstractions, even to propagate explicit substitutions. The variable substitutions propagation under abstractions will thus be delayed until the corresponding lambda will be applied. To cope with this subtlety, we introduce a relation on terms of λ_{cES} that expresses that a term M is related to N if M is the same as N up to the propagation of pending substitutions frozen under lambdas. If we would allow substitutions to be reduced under lambdas, M would reduces to N.

Definition 4.4.4. Substitution relation
We define $\rightsquigarrow_s$ as :

- $x \rightsquigarrow_s x$, $\mathrm{get}(r) \rightsquigarrow_s \mathrm{get}(r)$, $* \rightsquigarrow_s *$
- $M \parallel N \rightsquigarrow_s M' \parallel N'$ iff $M \rightsquigarrow_s M'$ and $N \rightsquigarrow_s N'$
- $\lambda x.M \rightsquigarrow_s \lambda x.M'$ if $M = N[\sigma_n][\ldots][\sigma_1]$ such that $N\{\sigma_1, \ldots, \sigma_n\} = M$.
- $M[\mathcal{V}]_\downarrow \rightsquigarrow_s M'[\mathcal{V}']_\downarrow$ or $M[\mathcal{V}]_\uparrow \rightsquigarrow_s M'[\mathcal{V}']_\uparrow$ iff $M \rightsquigarrow_s M'$ and $\mathcal{V} \rightsquigarrow_s \mathcal{V}'$
- $M\ N[\mathcal{V}]_\lambda \rightsquigarrow_s M'\ N'[\mathcal{V}']_\lambda$ iff $M \rightsquigarrow_s M'$, $N \rightsquigarrow_s N'$ and $\mathcal{V} \rightsquigarrow_s \mathcal{V}'$
- $M[\sigma] \rightsquigarrow_s M'[\sigma']$ iff $M \rightsquigarrow_s M'$ and $\sigma \rightsquigarrow_s \sigma'$

Where we extended point-wise the definition of $\rightsquigarrow_s$ to functions and multisets.

We can now state the simulation result modulo $\rightsquigarrow_s$:

Theorem 4.4.5. Simulation
Let P, Q be λ_{C} programs such that $P \to^* Q$. Then

$$\overline{P} \to^*_{\mathtt{nd}} M \rightsquigarrow_s \overline{Q}$$

The following subsection proves Theorem 4.4.5 with the help of auxiliary lemmas.

Proof of Simulation

Lemma 4.4.6 indicates that a reduction may be recasted in a larger context.

Lemma 4.4.6. Context reduction
Let M, N be terms of λ_{cES} such that $M \to^*_{\mathtt{nd}} N$ without using ($\mathtt{subst\text{-}r}_\top$), then for any context C and E, $C[E[M]] \to^*_{\mathtt{nd}} C[E[N]]$ using the same reduction rules.

Lemma 4.4.7 states that the translation is modular with respect to contexts. Lemma 4.4.8 shows that the translation commutes with implicit substitutions.

Lemma 4.4.7. Context decomposition
Let P be a program of λ_{C},

- If $P = C[E[M]]$ with $C = [.]$ or $C = [.] \parallel N$, then $\overline{P} = P$
- If $P = C[E[M]]$ with $C = [.] \parallel N \parallel S$, then $\overline{P} = \overline{E}^{S}[\overline{M}^{S}] \parallel \overline{N}^{S}$

Proof. Lemma 4.4.7
This is an easy induction following the definition of $\overline{}^{S}$. □

Lemma 4.4.8. Substitution for $\overline{}^{S}$
Let M, M' be terms, V a value and S a store of λ_{C}. Then

$$\overline{M\{x/V\}}^{S} = \overline{M}^{S}\{x/V\}$$

Proof. Lemma 4.4.8
By induction □

The absorption lemma (Lemma 4.4.9) (resp. diffusion lemma (Lemma 4.4.10)) studies how the translation of a term interacts with new downward reference substitutions (resp. upward reference substitutions).

Lemma 4.4.9. Absorption
Let M be a term and $S = S' \parallel r \Leftarrow V$ be a store of λ_{C}, C and E be context of λ_{cES}, and $\mathcal{V} = \{r \mapsto [V]\}$. Then

$$C[E[\overline{M}^{S'}[\mathcal{V}]_{\downarrow}]] \to^{*}_{\mathtt{nd}} C[E[\overline{M}^{S}]]$$

Proof. Lemma 4.4.9
By induction on M, using the definition of $\overline{}^{S}$ and the (`subst-r`) reductions. □

Lemma 4.4.10. Diffusion
Let N be a term, E a context, $S = S' \parallel r \Leftarrow V$ a store of $\lambda_{\mathrm{C}}$and $\mathcal{V} : r \mapsto [V]$. Let M be a term of λ_{cES}. Then

$$\overline{E}^{S'}[M[\mathcal{V}]_{\uparrow}] \to^{*}_{\mathtt{nd}} \overline{E}^{S}[M][\mathcal{V}]_{\uparrow} \tag{4.1}$$

$$M[\mathcal{V}]_{\uparrow} \parallel \overline{N}^{S'} \to^{*}_{\mathtt{nd}} (M \parallel \overline{N}^{S})[\mathcal{V}]_{\uparrow} \tag{4.2}$$

$$\overline{E}^{S'}[M[\mathcal{V}]_{\uparrow}] \parallel \overline{N}^{S'} \to^{*}_{\mathtt{nd}} \overline{E}^{S}[M] \parallel \overline{N}^{S}[\mathcal{V}]_{\uparrow} \tag{4.3}$$

using only rules (`subst-r`) or (`subst-r'`).

Proof. Lemma 4.4.10

- For (Equation (4.1)), we work by induction on E. As we will need Lemma 4.4.6, we add the induction hypothesis that the rule ($\texttt{subst-r'}_\top$) is not used.
  - $\overline{E}^{S'} = [.]$ nothing to do
  - $\overline{E}^{S'} = \overline{E'}^{S'}\ \overline{M'}^{S'}[\mathcal{U}]_\lambda$
    By IH, $\overline{E'}^{S'}[M[\mathcal{V}]_\uparrow] \to^*_{\texttt{nd}} \overline{E'}^{S}[M][\mathcal{V}]_\uparrow$ using no ($\texttt{subst-r'}_\top$) rules. We use Lemma 4.4.6 and get
    $$\overline{E}^{S'}[M[\mathcal{V}]_\uparrow] \to^*_{\texttt{nd}} (\overline{E'}^{S}[M][\mathcal{V}]_\uparrow)\ \overline{M'}^{S'}[\mathcal{U}]_\lambda$$
    We then apply ($\texttt{subst-r'}_{\texttt{lapp}}$) to get
    $$\overline{E}^{S'}[M[\mathcal{V}]_\uparrow] \to^*_{\texttt{nd}} (\overline{E'}^{S}[M])\ (\overline{M'}^{S'}[\mathcal{V}]_\downarrow)[\mathcal{W}]_\lambda[\mathcal{V}]_\uparrow$$
    Using Lemma 4.4.9, we get that the right part of the application reduces to $\overline{M'}^{S}$. Finally, by definition of $\overline{}^{S}$, the right hand side is $\overline{E}^{S}[M][\mathcal{V}]_\uparrow$.
  - Other cases are treated alike
- To prove (Equation (4.2)), we apply ($\texttt{subst-r'}_\parallel$). We then use Lemma 4.4.9 on $\overline{N}^{S'}[\mathcal{V}]_\downarrow$ to get $\overline{N}^{S}$ and Lemma 4.4.6 with $C = M \parallel [.]$ to get the desired result.
- Finally, we prove (Equation (4.3)) by combining the two results. We first use the strengthened version of (*Equation* (4.1)), so we can apply Lemma 4.4.6 and get
  $$\overline{E}^{S'}[M[\mathcal{V}]_\uparrow] \parallel \overline{N}^{S'} \to^*_{\texttt{nd}} M[\mathcal{V}]_\uparrow \overline{E}^{S}[M] \parallel \overline{N}^{S'}$$
  We use (Equation (4.2)) with $M' = E[M]$ to get the result.

□

Lemma 4.4.11 states that explicit substitutions implement implicit substitutions, up to the substitution order.

Lemma 4.4.11. Explicit substitution
Let $P = C[E[M[\sigma]]]$ with M containing no explicit substitutions nor upward reference substitution. Then

$$P \to^*_{\texttt{nd}} P' \rightsquigarrow_s C[E[M\{V/x\}]]$$

Proof. Lemma 4.4.11 By induction on M.

- Neutral
  If $M = *$, $M = \mathrm{get}(r)$ or $M = y$ with $y \neq x$, then $M\{V/x\} = M$. If we apply ($\mathtt{subst}_{\mathtt{unit}}$) (resp. ($\mathtt{subst}_{\mathtt{get}}$),($\mathtt{subst}_{\mathtt{var}}$)) we also get that
  $$C[E[M[\sigma]]] \to_{\mathtt{nd}} C[E[M]]$$
- $M = x$
  $M\{V/x\} = V$. By ($\mathtt{subst}_{\mathtt{var}}$),
  $$C[E[x[\sigma]]] \to_{\mathtt{nd}} C[E[V]]$$
- $M = \lambda y.M'$
  $M\{V/x\} = \lambda y.(M'\{V/x\})$. By ($\mathtt{subst}_{\lambda}$),
  $$C[E[\lambda y.M'[\sigma]]] \to_{\mathtt{nd}} C[E[\lambda y.(M'[\sigma])]] \rightsquigarrow_s C[E[\lambda y.(M'\{\sigma\})]]$$
- ...
  The remaining cases are treated similarly, the reduction rules being designed to make this lemma work.

□

We prove now the main technical result from which the simulation theorem follows easily:

Proposition 4.4.12. One-step simulation
Let P, Q be λ_{C} programs such that $P \to Q$. Then
$$\overline{P} \to^*_{\mathtt{nd}} M \rightsquigarrow_s \overline{Q}$$

Proof. Proposition 4.4.12
By case analysis on the reduction rule applied :

β_v We have $P = E[(\lambda x.M)\ V] \parallel N \parallel S$ and $Q = E[M\{V/x\}] \parallel N \parallel S$. Let By Lemma 4.4.7, $\overline{P} = \overline{E}^S[\lambda x.M\ V[\mathcal{V}_S]_\lambda] \parallel \overline{N}^S$ and $\overline{Q} = \overline{E}^S[\overline{M\{V/x\}}^S] \parallel \overline{N}^S$. Then, by one application of β_v in λ_{cES}, we have $\overline{P} \to_{\mathtt{aux}} Q_0 = \overline{E}^S[M[\mathcal{V}_S]_\downarrow[\sigma]] \parallel \overline{N}^S$. We apply Lemma 4.4.9 with to deduce $Q_0 \to^* Q_1 = \overline{E}^S[\overline{M}^S[\sigma]] \parallel \overline{N}^S$. Using Lemma 4.4.11 and Lemma 4.4.8, we get $Q_1 \to^* Q_2 \rightsquigarrow_s \overline{E}^S[\overline{M\{V/x\}}^S] \parallel \overline{N}^S = \overline{Q}$.

get We have $P = E[\mathrm{get}(r)] \parallel N \parallel S$ with $S = S' \parallel r \Leftarrow V$, and $Q = E[V] \parallel N \parallel S$. By Lemma 4.4.7, $\overline{P} = \overline{E}^S[\mathrm{get}(r)[\mathcal{V}]_\downarrow] \parallel \overline{N}^S$ and $\overline{Q} = \overline{E}^S[V] \parallel \overline{N}^S$. The rule ($\mathtt{subst\text{-}r}_{\mathtt{get}}$) gives $\overline{P} \to_{\mathtt{nd}} \overline{E}^S[V] \parallel \overline{N}^S = \overline{Q}$.

set We have $P = E[\mathrm{set}(r, V)] \parallel N \parallel S'$ and $Q = E[*] \parallel N \parallel S$ with $S = S' \parallel r \Leftarrow V$. By Lemma 4.4.7, $\overline{P} = \overline{E}^{S'}[*[\mathcal{V}]_{\uparrow}] \parallel \overline{N}^{S'}$ and $\overline{Q} = \overline{E}^{S}[*] \parallel \overline{N}^{S} = \overline{E[*] \parallel N}^{S}$. We apply Lemma 4.4.10 to get $P \to^* (\overline{E}^{S}[*] \parallel \overline{N}^{S})[\mathcal{V}]_{\uparrow}$ and then use the rule ($\texttt{subst-r}_{\top}$) to get rid of the toplevel reference substitution.

□

The only missing part is the following lemma allowing us to extend the result of Proposition 4.4.12. It shows that the substitution order is a relation that is compatible with reduction.

Lemma 4.4.13. Subsitution order is compatible with reduction
$\rightsquigarrow_s$ is compatible with reduction, meaning that if $M \rightsquigarrow_s N$ and $M \to_{\mathtt{nd}} M'$, then $N \to_{\mathtt{nd}} N'$ with $M' \rightsquigarrow_s N'$

Proof. Lemma 4.4.13
By case analysis on the reduction rule applied in $M \to_{\mathtt{nd}} M'$. As $\rightsquigarrow_s$ only relates to subterms that are under an abstraction, and the reduction do not reduce under abstractions, the reduction and the substitution order do not interact. □

Proof. Simulation(Theorem 4.4.5)
By induction on the length of the reduction $P \to^* Q$, using Proposition 4.4.12 and Lemma 4.4.13. □

4.4.3 Adequacy

The adequacy theorem states that the summands of the normal form of $\overline{M}$ are either the translation of a normal form $T = V_1 \parallel \ldots \parallel V_n$ that is a reduct of the original M, or garbage, that is a non-value term that corresponds to execution paths which deadlocked. We can recognize this garbage, thus eliminate it: with this additional operation, the summands of the normal form of $\overline{M}$ coincide with the values that are reachable by M.

Theorem 4.4.14. Adequacy
Let P be a well-typed term of λ_{C}. If $\overline{P} \to^* \sum_i M_i$ such that $\sum_i M_i$ is a normal form, then for any i, $P \to^* P_i$ with $M_i \rightsquigarrow_s M_i' \sqsubseteq \overline{P_i}$. That is, any term appearing in the normal form of the translation of P is $\sqsubseteq$-bounded (up to the substitution order) by the translation of a reduct of P.

In particular, applied to values, this gives the sought property for λ_{cES}:

Corollary 4.4.15. Let P be a well-typed term of λ_{C}. Assume that $\overline{P} \to^* M' + \mathbf{M}''$ where M' is a normal form.

- If $M' = V \parallel N$, then $P \to^* U \parallel Q$ with $V \rightsquigarrow_s \overline{U}$.
- In particular, if $M' = \ \|_i V_i$, then $P \to^* \ \|_i U_i$ with $V_i \rightsquigarrow_s \overline{U}_i$

The only problematic case is the non deterministic reduction ($\texttt{subst-r}_{\texttt{get}}$) which creates new summands. The proof consist in showing that these summands are actually limited in what they can do. Formally, they are bounded by the initial term that is being reduced, in the sense of the $\sqsubseteq$ preorder . The following lemma states that indeed the case of deterministic reduction is trivial:

Lemma 4.4.16. Values preservation
Let M be a term of λ_{cES}, and $M \to^*_{\texttt{nd}} M'$ without using ($\texttt{subst-r}_{\texttt{get}}$). We define $\text{NF}(M) = \{T \mid T \text{ normal and } M \to^*_{\texttt{nd}} T\}$. Then $\text{NF}(M) = \text{NF}(M')$

Proof. Lemma 4.4.16
If we do not use ($\texttt{subst-r}_{\texttt{get}}$), the two reduction (the standard $\to$ and the non-deterministic $\to_{\texttt{nd}}$) coincide: we have in particular $M \to^* M'$ by the standard reduction. Assume that $M \to^*_{\texttt{nd}} T$ where T is a normal form. Then $M \to^* T + \ldots$ through the standard reduction. By confluence, $M' \to^* T + \ldots$. This precisely means that $M' \to^*_{\texttt{nd}} T$. □

The following lemma states the adequacy result for a one-step reduction. The general theorem is then deduced by induction.

Lemma 4.4.17. Let $M = \overline{P}$ be the translation of a well-typed λ_{C} term such that $M \to_{\texttt{nd}} M'$. Then there exists M'', N', such that $M' \to^*_{\texttt{nd}} M'' \rightsquigarrow_s M''_s \sqsubseteq \overline{N'}$ and $N \to^{\{0,1\}} N'$, with all reductions from M' to M'' not being ($\texttt{subst-r}_{\texttt{get}}$).

Proof. Lemma 4.4.17
Given the form of a term which is the translation of a λ_{C} term, the only redexes in M are either premises of a (`subst-r'`) rule, the (β_v) rule or the ($\texttt{subst-r}_{\texttt{get}}$) rule. In the first case, it corresponds to a reducible set whose reduction can be carried on in M' by pushing the upward substitution to the top and pushing down the corresponding generated downward substitutions, to obtain the translation of N' (as in the simulation of a (`set`) reduction). We can proceed the same way with (β_v) : this corresponds to a β-redex in N, where N' is the result of reducing it, and M'' is obtained by pushing down the generated substitutions (variable and references). As we saw in simulation, this gives the desired result but up to $\rightsquigarrow_s$.

Finally, if the reduction rule is ($\texttt{subst-r}_{\texttt{get}}$), then either it chose one of the available values and it corresponds exactly to a get reduction $N \to N'$, or it threw away available values, in which case $N = N'$, $M' = M''$ and clearly $M'' \sqsubseteq \overline{N'}$. □

Proof. Theorem 4.4.14
Since P is well-typed, M is well-typed by Proposition 4.4.3. By Theorem 4.3.10, M is strongly normalizing. We can thus proceed by induction on $\eta(M)$, the length of the longest reduction starting from M. If $\eta(M) = 0$, i.e. M is a normal form, this is trivially true.
To prove the induction step, consider a reduct M' of M. We use Lemma 4.4.17 to get $M' \to^* M'' \sqsubseteq \overline{P'}$ for some reduct P' of P, such that the reduction to M'' doesn't use ($\texttt{subst-r}_{\texttt{get}}$). Thus, by Lemma 4.4.16, $\mathrm{NF}(M'') = \mathrm{NF}(M')$. By induction on M'', $\forall T \in \mathrm{NF}(M''), \exists Q, P \to^* Q$ and $T \sqsubseteq \overline{Q}$. But this is true for any reduct M', and we have $\mathrm{NF}(M) = \bigcup_{M \to M'} \mathrm{NF}(M')$, hence this is true for M. □

4.5 Summary

In this chapter, we presented a lambda-calculus with concurrence and references, featuring explicit substitutions for both variables and references. We discussed the issues explicit substitutions raise with respect to termination and explained how standard techniques fail to address them.

The main contribution of the chapter is a solution to this problem. Reminiscent of Game Semantics, the proof technique we apply is interesting in its own right. Based on an interactive point of view, it is reasonable to expect that the general methodology we present can be extended to other settings, such as proof nets or concurrent calculi.

This work we open the way to the encoding of a calculus with references into differential proof nets, which is done is Chapter 5.

4.6 Discussion

Comparison with Other Languages One may wonder how λ_{cES} compares to other concurrent calculi and especially the language λ_{C} presented in Section 4.4.1. In particular, λ_{cES} is almost an explicit substitution version of λ_{C}. Indeed, it turns out that we can define a translation of λ_{C} to λ_{cES}. The weak reduction of λ_{cES} prevents variable substitutions from percolating under abstractions, and translated terms may evaluate to closures as $\lambda x.(M[\sigma])$ instead of the expected $\lambda x.(M\{x_1/\sigma(x_1), \ldots, x_n/\sigma(x_n)\})$ if $\mathrm{dom}(\sigma) = \{x_1, \ldots, x_n\}$. Up to this difference there is a simulation of λ_{C} in λ_{cES} (Theorem 4.4.5).

More generally, we followed the design choice of adopting cumulative stores, while many languages in the literature and in practice follow an erase-on-write semantics. Remarkably, our choice makes the version with explicit substitutions

asynchronous, as various upward and downward substitutions may be reduced arbitrarily without the need of any scheduling. Another point that justifies its introduction is that such calculi simulate a lot of other paradigms, such as erase-on-write or communication channels for example, as mentioned in [41]. This means that the termination of the cumulative store version implies the termination of the aforementioned variants. To illustrate our point, let us quickly sketch how a calculus with explicit substitutions with an erase-on-write semantics could be devised. First, encode $\mathrm{set}(r, V)$ as $((\lambda x.*[r \mapsto [V]]_{\uparrow})\,*)$. Then, when an upward substitution becomes reducible, apply all possible downward and upward structural rules until it is not possible anymore. Finally, instead of merging reference substitutions, the upper one erases the lower one. Its termination follows immediately from the one of λ_{cES}.

Explicit substitutions and Linear Logic Proof nets are strongly connected to systems with explicit substitutions (see e.g. [2]). λ_{cES} anticipates the translation presented in Chapter 5: its constructs are already inspired by the approach of [14] and [56]. λ_{cES} can be seen as a calculus-side version of the nets we will be translating to. The translation and simulation of λ_{C} in λ_{cES} could be described as a compilation from a *global shared memory* model to a *local message passing* one. The correctness of this compilation requires that a well-typed strongly normalizing term in the initial language is also strongly normalizing in the target language, and this is what this chapter achieves.

Chapter 5

Encoding a concurrent λ-calculus in nets

In this chapter, we put the devices developed in Chapter 3 to use by presenting a translation of λ_{cES} - the language introduced in Chapter 4 - in the nets described in Chapter 2. The translation builds on ideas from [56] for the encoding of references, and from [14] for concurrency, which inspired the routing areas.

We build on the standard call-by-value translation of the λ-calculus in proof nets [42]. In this translation, terms are interpreted as nets, where the interface of the resulting net corresponds to its output (the result of the whole term) as well as free variables. This embodies the different ways a term can interact with an environment, such as a when it is put inside a context. Free variables, that may be substituted by values, are the *inputs*. The eventual result that the term returns is the *output*.

In λ_{cES}, references add new possibilities of interaction. Each reference r on which a well-typed term may act, appearing in the effect e of its typing judgement, adds a new input and output capability. The input corresponds to reads from the reference r through a $\text{get}(r)$, that can substituted by a downward reference substitutions. The output corresponds to assignments, implemented by the upward reference substitutions generated by the term. Thus for each such reference, the translation of a term M of λ_{cES} will have a new pair of input/output free wires in addition to the wires of free variables and the output. These are the wires that are combined using routing areas. After the presentation of the translation, we give a simulation, a termination and an adequacy theorem.

In order to translate well-typed terms of λ_{cES} in nets, we must also translate types and effects: this is where the monadic translation of [56] comes into play. It gives a technique to encode stratified type and effects to formulas of multiplicative linear logic. This allows us to label the input and output wire of a reference r by a standard LL formula.

Overview In Section 5.1, we explain the translation in detail. We start by translating type and effects in Section 5.1.1. We explain how effects are combined using routing areas in Section 5.1.2. Then, Section 5.1.3 explain how terms of λ_{cES} are translated to nets.

The following sections are dedicated to the study of the properties of the translation. In Section 5.2, we state and prove a simulation theorem .

In Section 5.3 shows a termination and adequacy theorem. In Section 5.3.1, we prove that the translation of a term is weakly normalizing. In Section 5.3.3, we prove that the values we obtain by normalizing a term and the ones obtained by normalizing its translation are essentially the same.

After discussing the chapter in Section 5.5, Section 5.6 finally concludes with perspective of future work.

5.1 The Translation

5.1.1 Translating types and effects

Before translating terms, we need to translate the types from the type and effects system of λ_{cES} to plain LL formulas. We use the approach of [56], a monadic translation, explained in the following. Before translating to LL, let us first try to take a type with effects and translate it to a pure simple type. Let $e = \{r_1, \ldots, r_n\}$ be an effect (a finite set of references), assume we can assign a simple type R_i to each reference r_i. The type a store representing the current state of the memory would be $S_e = R_1 \times \ldots \times R_n$. We transform a term M of type A producing effects to a pure term which takes the initial state of the store, and returns the value it computes together with the new state of the store after this computation. Using curryfication for the arrow type, we define the following translation:

$$\begin{array}{rcl} T_e(\alpha) & = & S \to S \times \alpha \\ T_e(A \xrightarrow{e} \alpha) & = & A \to (S \to (S \times \alpha)) \cong A \times S \to S \times \alpha \end{array}$$

From there, we go to LL types by implementing the pair type $A \times B$ as $!A \otimes !B$, and the usual call-by-value translation for the arrow $(A \to B)^\bullet = !(A^\bullet \multimap B^\bullet)$ [42]. We still have to determine each R_i first. Using the previous formula, we may associate an LL type variable X_{r_i} to each reference and plug everything in to obtain the following equations (where A_i is the type given to r_i by the reference context):

$$
\begin{aligned}
\mathrm{Unit}^\bullet &= !1 \\
(A \overset{\{s_1,\ldots,s_m\}}{\to} \alpha)^\bullet &= !((A^\bullet \otimes X_{s_1} \ldots \otimes X_{s_m}) \multimap (X_{s_1} \otimes \ldots \otimes X_{s_m} \otimes \alpha^\bullet)) \\
X_{r_i} &= A_i{}^\bullet
\end{aligned}
$$

This system is solvable precisely because the type system is stratified [56], and we can thus translates all the types of λ_{C} to plain LL types. The behavior type $\mathbf{B}$ will be translated to types of the form $A_1 \parr \ldots \parr A_n$, as the translation indeed remembers the types of each threads.

5.1.2 Combining effects

The monadic translation allows to translate a type and effect to a plain LL type. Each reference appearing in the effects e of the type of a term will have two corresponding wires in the translation, one for the input and the other for the output. When the effects of two subterms must be combined, mainly in the application case and the parallel case, these wires are connected through routing areas. More precisely, we use two specific routing areas which we introduce below: the γ area and the δ area. In the following, $E_i = \{1, \ldots, i\}$ and R_i is the binary relation defined on E_i by $k \; R_i \; l \iff k \neq l$.

The γ area is defined by $(E_3, E_3, R_\gamma = R_3)$. It is composed of 3 pairs of input and outputs grouped by label. Each such pair represents a plug to which translated terms will be connected. The definition of R_γ expresses that the input and the output of a plug are not connected, as a component should not receive the data it sent himself: this would be the analog of a short-circuit. All others inputs and output are connected.

The δ area is an analog structure with 4 plugs: (E_4, E_4, R_δ). It is designed to handle the application $M \; N$ which includes three potential sources of effects :

1. The effects e_1 produced by reducing M to $\lambda x.M'$
2. The effects e_2 produced by reducing N to value V_N
3. The effects e_3 produced by reducing $M'[V_N/x]$ to the final result V

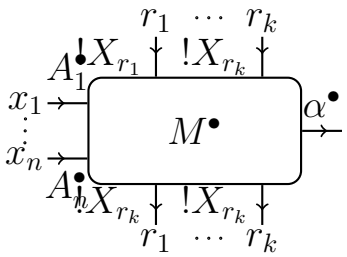
The reduction of λ_{C} imposes that e_1 and e_2 happen before e_3, while e_1 and e_2 may happen concurrently. For $1 \leq i \leq 3$, the plug (i, i) of δ corresponds to the effects e_i. The last one is the external interface for future connections. We easily accommodate δ to implements the sequentiality constraint by removing the couples $(3, 1)$ and $(3, 2)$ from R_4 to form R_δ. Indeed e_1 and e_2 happens before e_3 thus cannot observe any assignment made by the latter.

To implement it, we just cut the corresponding wires. We see that the formalism of routing areas allows us to easily encode the order of effects.

We are now ready to give the translation of terms.

5.1.3 Translating terms

The general form of the translation of a term $x_1 : A_1, \ldots, x_n : A_n \vdash M : (\alpha, \{r_1, \ldots, r_k\})$ is given by



We distinguish three different types of free wires:

Output wire The right wire, labelled by $\alpha^\bullet$, corresponds to the result of the whole term.

Variable wires Each wire on the left corresponds to a variable of the context. The explicit substitution of a variable x for a term $V^\bullet$ is obtained by connecting the output wire of $V^\bullet$ to the wire of x.

References wires The wires positioned at the top are input wires corresponding to references and have a similar role as variable wires, while the wires at the bottom corresponds dually to the output. References wires will be connected by routing areas.

We give the different cases of the translation above. get(r) and reference substitutions demonstrate how reference wires interact with the rest. The abstraction shows how effects are thunked in a function's body following the monadic translation. Application and parallel use the routing areas to handle effects scheduling. Variable substitutions are handled in a traditional way, using a cut (or, in nets, just a connection).

In all constructions, the different variable wires corresponding to multiple occurrences in subterms of a variable appearing in the context are unified through a contraction.

$$\frac{R;\Gamma, x : A \vdash M : (\alpha, e) \qquad R;\Gamma \vdash V : (A, \emptyset)}{R;\Gamma \vdash M[\sigma] : (\alpha, e)}\ (subst)$$

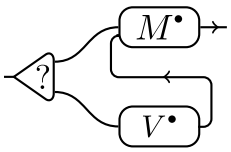
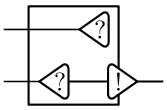


Figure 5.1: Translation of variable reference substitutions

$$\frac{}{\Gamma, x : A \vdash x : (A, \emptyset)}\ (var)$$



$$\frac{}{\Gamma \vdash * : (\mathtt{Unit}, \emptyset)}\ (unit)$$

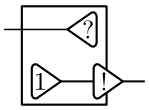


Figure 5.2: Translation of variable and $*$

Var and Unit (Figure 5.2) The var rule is encoded as an axiom, while the constant $*$ is encoded as a !1. The axiom is eta expanded, hence it is surrounded by an exponential box. The wires corresponding to variables that are not free in the term but appear in the context are provided by weakenings. Note that the weakenings are willingly put inside the exponential boxes: thanks to the closed box constraint of the reduction, this reflects the sequentiality of the term as it requires that all free variables are effectively substituted before a variable or a $*$ may be itself moved around. This is required for both adequacy and termination.

Variable substitution (Figure 5.1) A variable explicit substitution is naturally built by connecting the output wire of a substituted value to the corresponding variable wire.

Get and reference substitutions (Figures 5.3 to 5.5) An upward reference substitutions $M[\mathcal{V}]_{\uparrow}$ packs the output of the translation of $\mathcal{V}$ with the output reference wire corresponding to r of M using a cocontraction.

One important remark is that in the translation of a reference substitution $\mathcal{V}$, an additional exponential layer is added around the translation of V. In a call-by-value language, the non determinism is strict in the sense that non-deterministic terms must be evaluated before any copy. For example, the term $(\lambda x.f\ x\ x)\ \mathrm{get}(r)[r \mapsto [V_1, V_2]]_{\downarrow}$ can reduce either to $f\ V_1\ V_1$ or $f\ V_2\ V_2$ but not to $f\ V_1\ V_2$. Differential LL rather implements call-by-name semantics of the

$$\frac{R \vdash \Gamma}{R;\Gamma \vdash \mathrm{get}(r) : (A, \{r\})}\ (get)$$

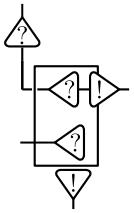


Figure 5.3: Translation of $\mathrm{get}(r)$

$$\frac{R;\Gamma \vdash M : (\alpha, e) \qquad R;\Gamma \vdash V \in \mathcal{V}(r) : (R(r), \emptyset)}{R;\Gamma \vdash M[\mathcal{V}]_{\downarrow} : (\alpha, e)}\ (\textit{subst-r} \downarrow)$$

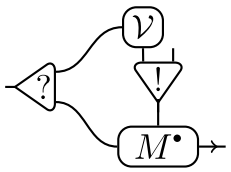


Figure 5.4: Translation of downward reference substitutions

$$\frac{R;\Gamma \vdash M : (\alpha, e) \qquad R;\Gamma \vdash V \in \mathcal{V}(r) : (R(r), \emptyset)}{R;\Gamma \vdash M[\mathcal{V}]_{\uparrow} : (\alpha, e)}\ (\textit{subst-r} \uparrow)$$

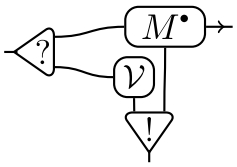


Figure 5.5: Translation of upward reference substitutions

latter example as hinted at by the $\overset{\texttt{ba}}{\to}$ rule which expresses that duplication and non-determinism should commute. The mismatch is due to two different usages we want to make of the ! modality:

- The first one allows to discriminate what nets can be the target of structural rules, which implements substitution. In call-by-value, the only terms that can be substituted are values. The ! is introduced by the translations of values, using $!_p$, and eliminated at usage - when applied to another term - for each copy by a dereliction.
- The second usage relates to the differential part. The bang denotes resources that may be packed non deterministically by a cocontraction. The choice is made when a dereliction is met.

But as we noted, these two usages are in contradiction: a non-deterministic packing should not be allowed to be substituted. Technically, the dereliction corresponding to the place of usage and the dereliction corresponding to the non-deterministic choice should not be the same. This is the reason of the additional ! layer introduced by an exponential box around $V^\bullet$. The corresponding dereliction is found in the translation of get(r). The get(r), dual of the set, takes a resource from the corresponding input reference wire and redirects it to the output wire. It outputs a coweakening on the reference wire as it does not produce any assignment. As mentioned in the previous case, a dereliction is added on the input wire to force the non-deterministic choice and strip the exponential layer added by the set.

Abstraction (Figure 5.6) The abstraction thunks the potential effects of the body M in the pure term $\lambda x.M$. Following the monadic translation, the input effects are tensorized with the bound variable, and the output effects with the

$$\frac{R; x : A, \Gamma \vdash M : (\alpha, e)}{R; \Gamma \vdash \lambda x.M : (A \xrightarrow{e} \alpha, \emptyset)}\ (lam)$$

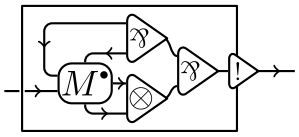


Figure 5.6: Translation of $\lambda x.M$

$$\frac{R; \Gamma \vdash M : (A \xrightarrow{e_1} \alpha, e_2) \qquad R; \Gamma \vdash N : (A, e_3) \qquad R; \Gamma \vdash V \in \mathcal{V}(r) : (R(r_i), \emptyset)}{R; \Gamma \vdash M\ N[\mathcal{V}]_\lambda : (\alpha, e = e_1 \cup e_2 \cup e_3)}\ (app)$$

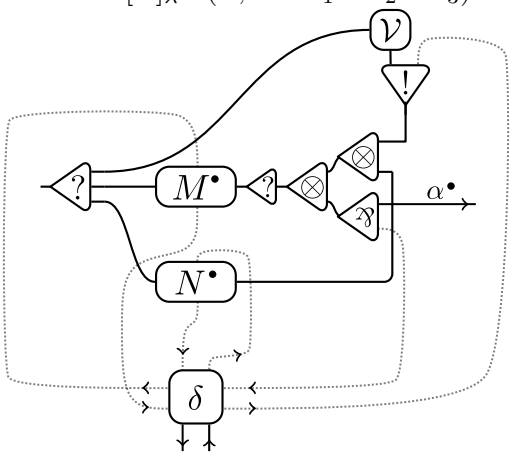


Figure 5.7: Translation of $M\ N$

output of M. Finally the whole term is put in an exponential box as it is a value.

Application and parallel (Figure 5.7) The application puts the routing area at use. Using the same terminology as in the introduction of this section, we see the effects e_1 and e_2 coming respectively from the evaluation of M and N, and e_3, liberated by the body of the function being applied, plugged accordingly on the δ area. The parallel operator is similar to the application, but simpler. The two outputs of the terms are connected using a ⅋. Unlike application, there is no constraint on the order of effects, thus reference wires are connected through a symmetric γ area.

Note that the translation of a value V is an exponential box. This box is

$$\frac{i = 1, 2 \qquad R; \Gamma \vdash P_i : \alpha_i}{R; \Gamma \vdash P_1 \parallel P_2 : \mathbf{B}}\ (par)$$

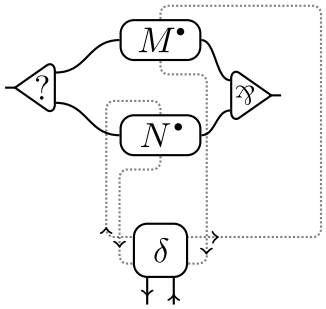


Figure 5.8: Translation of parallel

closed if and only if the corresponding value has no free variables.

We now study the properties of this the translation, starting with simulation.

5.2 Simulation

The simulation theorem states that a reduction step in the source language is reproduced by zero or more reduction in nets.

Theorem 5.2.1. Simulation for λ_{cES}
Let $\vdash M : (\alpha, e)$ be a closed well-typed term of λ_{cES}. If $M \to^* N$, then $M^\bullet \Rightarrow^* N^\bullet$.

To derive this result, we define a notion of typed context and a translation of contexts to nets.

5.2.1 Nets contexts

Reduction rules of λ_{cES} are defined up to contexts (see Table 4.3). To prove simulation, it is natural and convenient to give contexts a proper treatment, by providing a notion of typed context and a translation of these contexts to nets. In both cases, this boils down to add a special treatment for the hole, and use the material already developed for terms otherwise. We first extend typing to context: the only additional rule we need is the one for the hole [.]. The other constructors of contexts are the same as the constructors of standard terms, hence we can the same typing rules of terms.

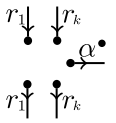
Definition 5.2.2. Hole typing rule

$$\frac{R \vdash (\alpha, e)}{R, [.] : (\alpha, e) \vdash [.] : (\alpha, e)}\ (hole)$$

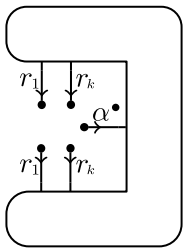
[.] can be seen as a special kind of variable in the typing derivation. But unlike variables, it can be substituted by something else than a value. Using the standard typing rules of λ_{cES}, we can build type derivations of contexts. A context can then be translated to a net, with an interface determined by α and e, labelling free ports. This is what we call net contexts. The substitution of a net in a net context consists in plugging the translation of a term with a compatible type in this interface.

Definition 5.2.3. Hole translation
We define the translation of a typed hole $R \vdash [.] : (\alpha, e)$ as

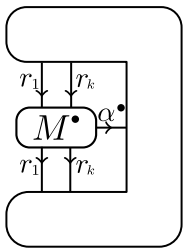
$$\frac{R \vdash (\alpha, e)}{R, [.] : (\alpha, e) \vdash [.] : (\alpha, e)}\ (hole)$$



We just extend the translation with a case for the hole. We can then carry on and use the usual term translation to build the translation of a context $(C[E])^\bullet$, which is a net of the form



The substitution of $M^\bullet$, or of any net with a compatible interface for that matter, in $(C[E])^\bullet$ is defined by just connecting the free wires of the substituted net to the corresponding free wires of the context hole :



The fundamental property of net contexts and net substitutions is that the substitution commutes with the translation, in the following sense :

Proposition 5.2.4. Nets substitution
Let $\vdash [.] : (\alpha, e)$ be a typed hole, $[.] : (\alpha, e) \vdash C[E] : (\beta, e')$ a typed context, and $\vdash M : (\alpha, e)$ a term. Then :

$$(C[E[M]])^\bullet = (C[E])^\bullet[M^\bullet]$$

The very definition of net reduction immediately entails that if $M^\bullet \Rightarrow^* N^\bullet$ then $(C[E])^\bullet[M^\bullet] \Rightarrow^* (C[E])^\bullet[N^\bullet]$. Together with Proposition 5.2.4, this ensures that we can focus on the case where $C = E = [.]$, as the general case follows seamlessly.

From here, we check that each reduction rule of λ_{cES} can be simulated on the net side, relying on the definition of the reduction and the behavior of routing areas. We will prove the following one-step simulation lemma:

Lemma 5.2.5. One-step simulation
Let $\vdash M : (\alpha, e)$ be a closed well-typed term of λ_{cES}. If $M \to N$, then $M^\bullet \Rightarrow^* N^\bullet$.

To do so, we proceed by case analysis on the reduction rule applied in the reduction $M \to N$.

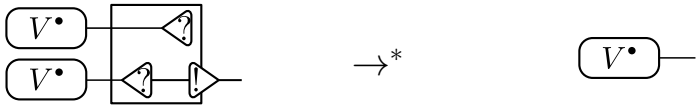
5.2.2 Variable substitutions reductions

Let us show the case of a (`subst`) rule. Thanks to the previous theorem, we can assume that contexts are empty without loss of generality. We consider only closed terms. Indeed, since reduction contexts S, C, E do not bind variables, all the terms appearing in the premise of a rule are thus closed terms, and we can omit their context. For each rule, we write the translation of the premise followed by its reduction in nets, which matches the conclusion.

The fundamental rule that performs the substitution is ($\texttt{susbt}_{\texttt{var}}$).

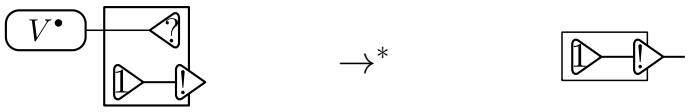
($\texttt{subst}_{\texttt{var}}$) ($\sigma(x)$ undefined)

($\texttt{subst}_{\texttt{var}}$) ($\sigma(x)$ defined)



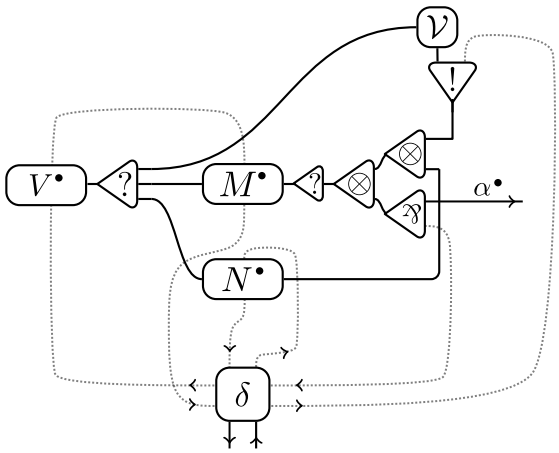
When reaching a get(r) or a $*$, the substitution simply vanishes.

($\texttt{subst}_{\texttt{unit}}$)

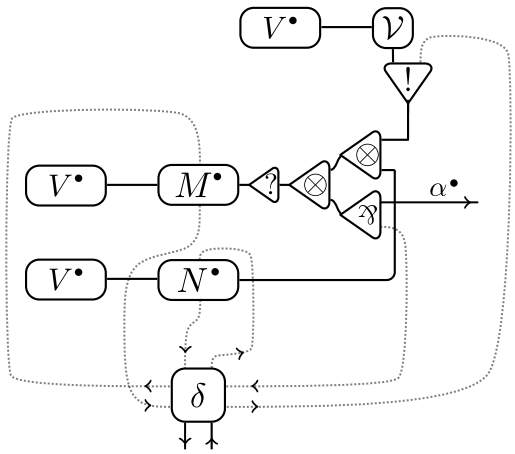


The ($\texttt{subst}_{\texttt{app}}$), ($\texttt{subst}_{\texttt{subst-r}}$) and ($\texttt{subst}_{\texttt{subst-r'}}$) perform a duplication and propagate the substitution inside reference substitutions.

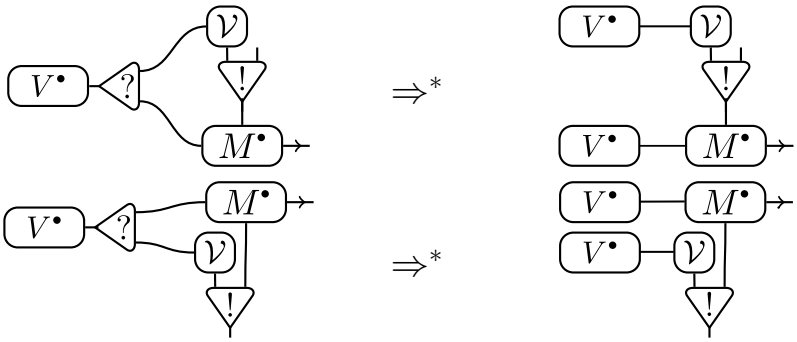
$(\mathtt{subst}_{\mathtt{app}})$



$\Rightarrow^*$



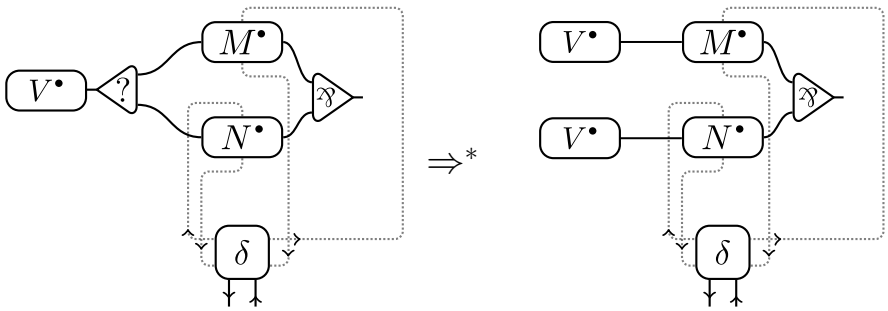
$(\mathtt{subst}_{\mathtt{subst\text{-}r}})$



$(\mathtt{subst}_{\mathtt{subst\text{-}r'}})$

The $(\mathtt{subst}_{\|})$ just duplicate the variable substitution to the two threads

$(\mathtt{subst}_{\|})$



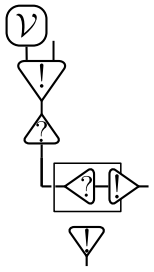
Finally, the ($\texttt{subst}_{\texttt{merge}}$) distributes the outer substitution to both the term and the inner substitution.

($\texttt{subst}_{\texttt{merge}}$)

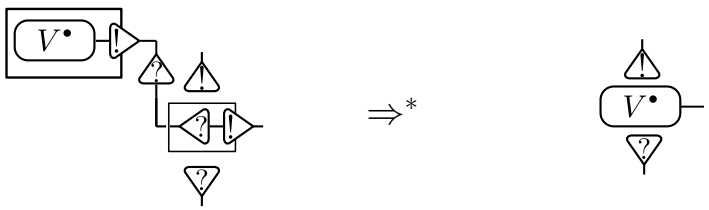
5.2.3 Downward reference substitutions reductions

The propagation of references substitutions relies on the behavior of routing area, and especially Proposition 3.2.7. The fundamental case is the non deterministic reduction happening when reducing a get(r) whose redex is

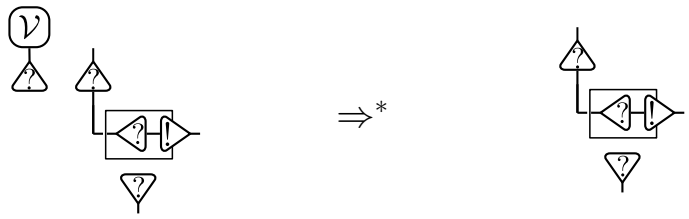
($\texttt{subst-r}_{\texttt{get}}$)



Then for each V in the image of $\mathcal{V}$, there will be exactly one summand of the following form

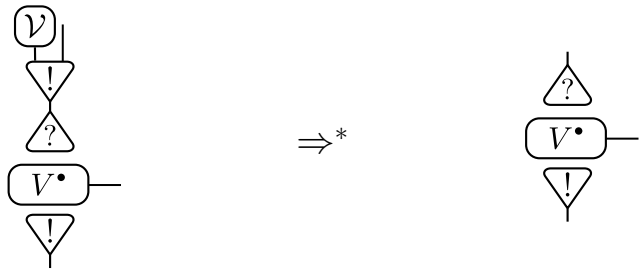


The remaining term reduces to the translation of get(r):

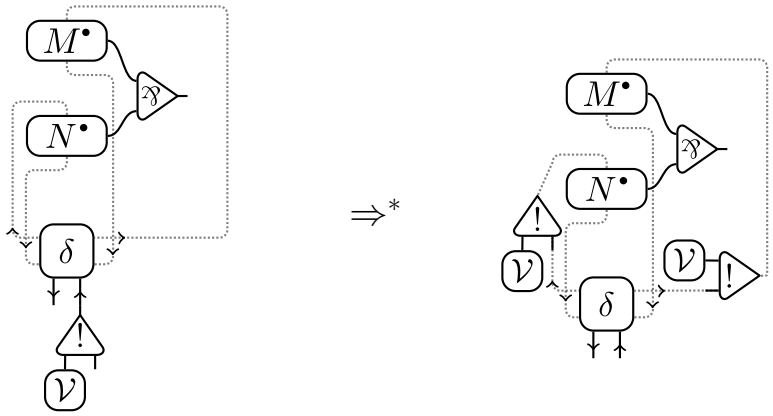


($\texttt{subst-r}_{\texttt{val}}$), ($\texttt{subst-r}_{\parallel}$) and ($\texttt{subst-r}_{\texttt{app}}$) are just direct application of the Proposition 3.2.7.

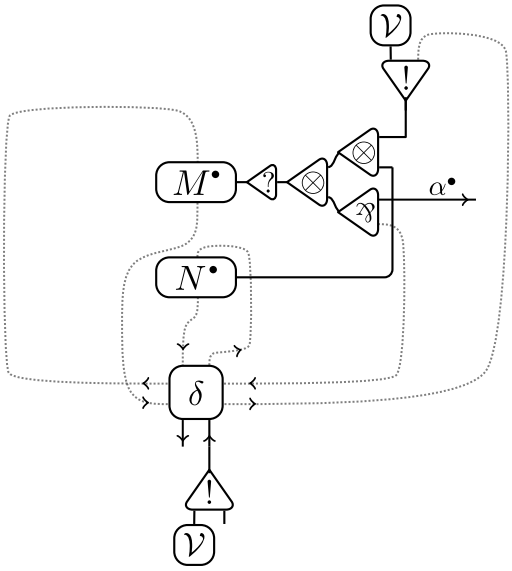
($\texttt{subst-r}_{\texttt{val}}$)



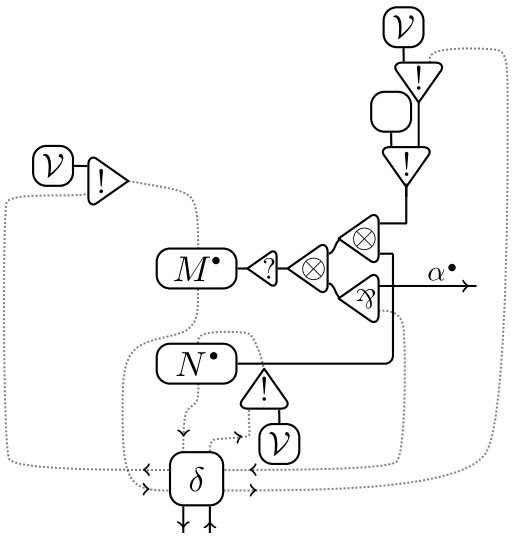
($\texttt{subst-r}_{\parallel}$)



($\texttt{subst-r}_{\texttt{app}}$)

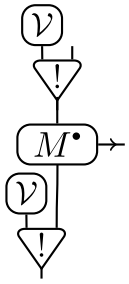


$\Rightarrow^*$

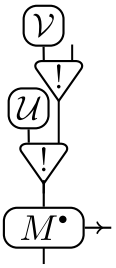


($\texttt{subst-r}_{\texttt{merge}}$) and ($\texttt{subst-r}_{\texttt{subst-r'}}$) amount to nothing in nets, as the translation already identifies the redex and the reduct of these rules.

($\texttt{subst-r}_{\texttt{subst-r'}}$)



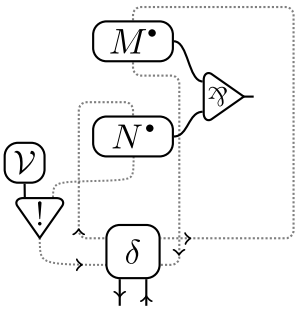
($\texttt{subst-r}_{\texttt{merge}}$)



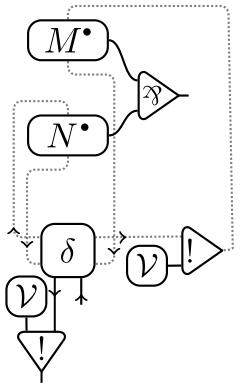
5.2.4 Upward reference substitutions reduction

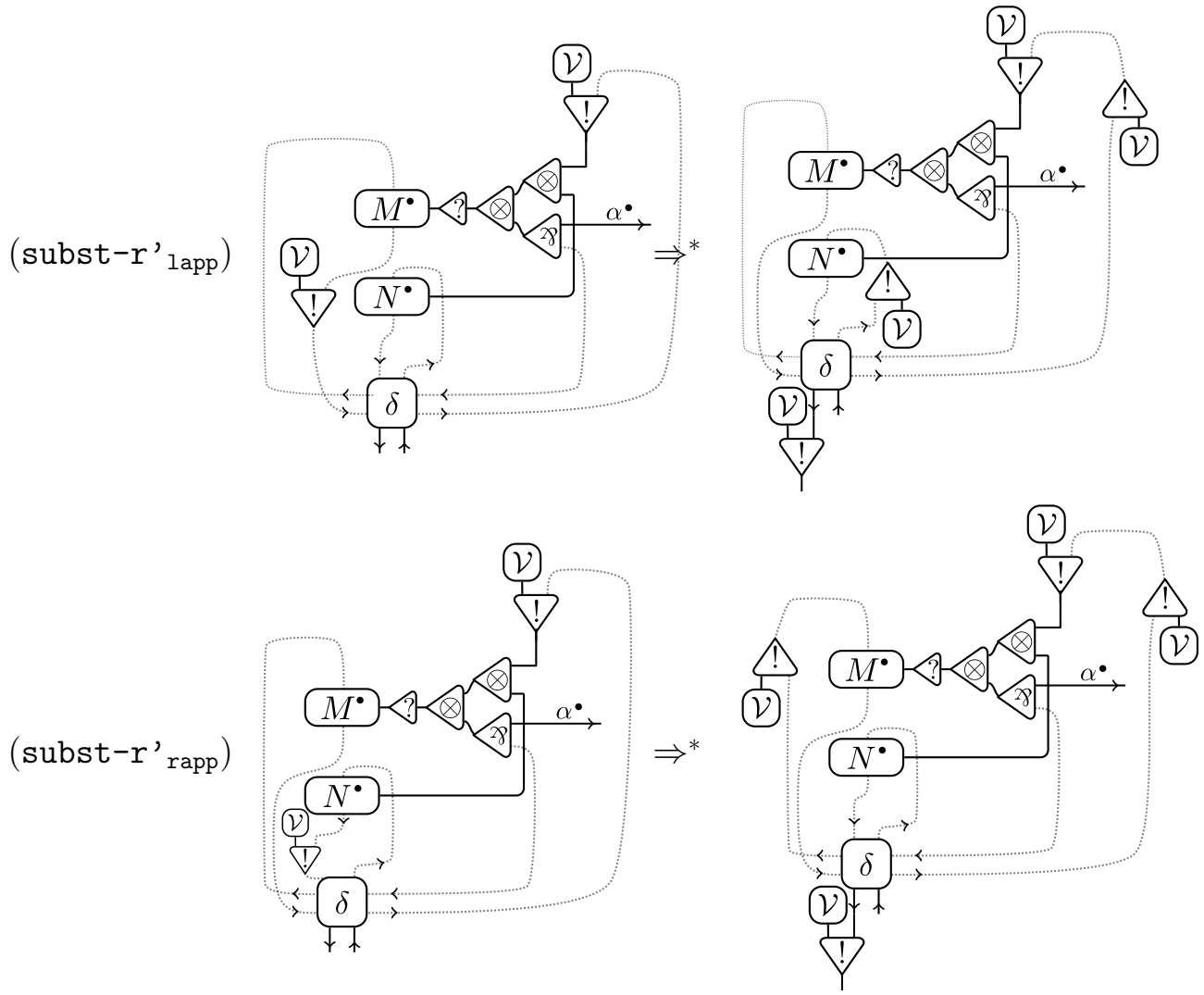
As for downward substitutions, the main ingredient is the Transit lemma, applied to our specific routing area δ and γ.

($\texttt{subst-r'}_{\|}$)



$\Rightarrow^*$





5.3 Termination and Adequacy

5.3.1 Termination

In this section we show that the nets that are the translation of a well-typed terms of λ_{cES} are weakly normalizing. Nets are indeed more liberal in their reduction: for example, they reduce an expression corresponding to a β-redex $(\lambda x.M)\ N$ even if N is not a value (take a get(r)) for example. There also administrative reductions, corresponding to merging of routing areas, which are invisible on the calculus side. As a result, the translation of a normal form of λ_{cES} my not be a normal net. The goal of this section is to show that however, the translation of a normal form of λ_{cES} can be reduced in a few steps to a normal net that can be related to the original term.

Proposition 5.3.1. Let $\vdash N : (\alpha, e)$ be a closed well-typed term of λ_{cES} which is a normal form. Then $N^\bullet$ is strongly normalizing.

Together with simulation, this gives the following theorem:

Theorem 5.3.2. Weak normalization
Let $\vdash M : (\alpha, e)$ be a closed well-typed term of λ_{cES}. Then $M^\bullet$ is weakly normalizing.

Proof. Theorem 5.3.2
Let $\vdash M : (\alpha, e)$ be a closed well-typed term of λ_{cES}. By Theorem 4.3.10, it is strongly normalizing, hence $M \to^* N$ where is normal. By simulation, $M^\bullet \Rightarrow^* N^\bullet$. By Proposition 5.3.1, $N^\bullet$ is strongly normalizing, and in particular $M^\bullet \Rightarrow^* N^\bullet \Rightarrow^* \mathcal{N}$ where $\mathcal{N}$ is a normal form. □

The next section gives the detailed proof of Proposition 5.3.1.

5.3.2 Proof of Proposition 5.3.1

To prove Proposition 5.3.1, we proceed in two stages: first, we extend λ_{cES} to a language $\lambda_{\text{cES}}+$ that is able to perform more reductions than λ_{cES}, precisely the kind of generalized β reduction mentioned above. The extension of the translation and the simulation theorem for $\lambda_{\text{cES}}+$ are straightforward. Then, we show that $\lambda_{\text{cES}}+$ also terminates, give an explicit grammar for its normal forms, and show that the translation of these normal forms are strongly normalizing nets.

Working on the calculus

We add a labelled reference substitution variable substitution to λ_{cES} : $M[|\mathcal{V}|]_\downarrow$ | $M[|x/N|]$. The first one, $M[|\mathcal{V}|]_\downarrow$, corresponds to a downward reference substitution where $\mathcal{V}$ has no free variables. The second one corresponds to a variable substitution of a term that is not a value. Both can appear during the reduction in nets but are not accounted for in λ_{cES}. $M[|x/N|]$ cannot be reduced, either in N or by (`subst`) rules (whether N is a value or not). The labelled variable substitution forbids any reduction under it, except for the labelled reference substitutions. We extend the reduction rules (`subst-r`) to the labelled substitution $[|\mathcal{V}|]_\downarrow$. They can be performed anywhere (included under a variable substitution) except under abstraction, as defined by the following reduction contexts: $J ::= [.] \mid J\ M \mid M\ J \mid J[\mathcal{V}]_\downarrow \mid J[\sigma] \mid J[|x/N|]$.

We add a generalized β-rule which fires such labelled substitution :

$$(\beta') \quad (\lambda x.M)\ N[\mathcal{V}]_\lambda \quad \to \quad M[|\mathcal{V}|]_\downarrow[|x/N|]$$

This corresponds to the additional reduction nets can do. A β redex can always be fired in nets even if the argument is not a value. But then, the corresponding term do need to be reduced before any duplication, erasure or substitution. Let us now show termination and describe the normal forms of $\lambda_{\text{cES}}+$:

Definition 5.3.1. F-normal forms
Let M be a normal form of the form of λ_{cES}. It belongs to the grammar M_{norm} (cf Lemma 4.2.7). Then M reduces to a sum of terms in $\lambda_{\text{cES}}+$, where each summand belongs to the following grammar of F-normal forms :

$$F_{\text{norm}} \quad ::= \quad \text{get}(r) \mid F_{\text{norm}}\ V[\mathcal{V}]_\lambda \mid F_{\text{norm}}\ F_{\text{norm}}[\mathcal{V}]_\lambda \mid M[|x/F_{\text{norm}}|]$$

where the M is in λ_{cES} (contains no labelled substitution).

Proof. By induction on the structure of M, with the additional hypothesis that only the additional rules are performed (no upward substitution) :

- $M = \text{get}(r)$: ok
- $M = M_{\text{norm}}\ V[\mathcal{V}]_\lambda$: by induction, M_{norm} reduces to a sum of F_{norm}. Take one such summand N, then $M \to^* N_{\text{norm}}\ V[\mathcal{V}]_\lambda$ (because no rule (`subst-r'`$_\top$) was used)
- $M = M_{\text{norm}}\ M'_{\text{norm}}[\mathcal{V}]_\lambda$: we proceed as in the previous case
- $M = V\ M_{\text{norm}}[\mathcal{V}]_\lambda$: by induction, M_{norm} reduces to some N in F_{norm} grammar, so $M \to^* V\ N[\mathcal{V}]_\lambda$. Then, by inversion of typing rules, V is of the form $\lambda x.P$ and we can apply the new β-reduction to get $P[|\mathcal{V}|]_\downarrow[|x/N|]$. By pushing down $[|\mathcal{V}|]_\downarrow$ in P, we can reduce it to a sum of P_is where each P_i do not contain labelled substitutions. Then $M \to^* \sum_i P_i[x/N]$

□

Lemma 5.3.3. F-normal forms are normal.

Proof. By induction :

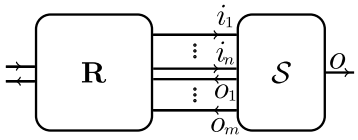
- $M = \text{get}(r)$: ok
- $M = F_{\text{norm}}\ V[\mathcal{V}]_\lambda$: by induction, F_{norm} is normal, and values are not F-normal form, so the application cannot create any β-redex.
- $M = M_{\text{norm}}\ M'_{\text{norm}}[\mathcal{V}]_\lambda$: same as the previous case
- $M = M[|x/F_{\text{normal}}|]$: the explicit substitution prevents any reduction except (`subst-r`) ones for labelled reference substitution, but by definition, M does not contain any.

□

Working on the nets

We will see in the following that the translation of a F-normal form has not many possible reductions left. However, a few steps may remain to eventually reach a normal form. The first step is to collect and merge all the routing areas that are created and connected during the translation. Doing so, we separate the net between a part that closely follows the translated term structure, and a big routing area which connects various subterms to enable communication through references between them. Once this is done, a few starving get(r) may interact with the routing area by generating sums, but without being actually substituting. The following definition give the shape obtained after the merging of routing:

Definition 5.3.4. Separability
Let $\mathcal{R}$ be the translation of a $\lambda_{\text{cES}}+$ term, then $\mathcal{R}$ is say to be *separable* if it can be reduced to the following form :



where $\mathbf{R}$ is a routing area and $\mathcal{S}$ a net with free wires labelled by $i_1, \ldots, i_n, o_1, \ldots, o_m, O$, satisfying :

(a) There is no redex in $\mathcal{S}$

(b) $i_1, \ldots, i_n$ are either connected to the auxiliary port of a $\otimes$ cell, to the auxiliary port of a cocontraction or to the principal port of a dereliction

(c) $o_1, \ldots, o_m, O$ are either connected to a $\parr$ cell, or to the principal door of an open box

The translation of term with at least one free variable is separable. The reason is that constructors such as application, substitution, parallel composition, etc. preserve separability. Moreover, the translation of values with free variables are obviously separable. The weakenings, which correspond to free variables, materializes as auxiliary doors of exponential boxes, and block further reduction.

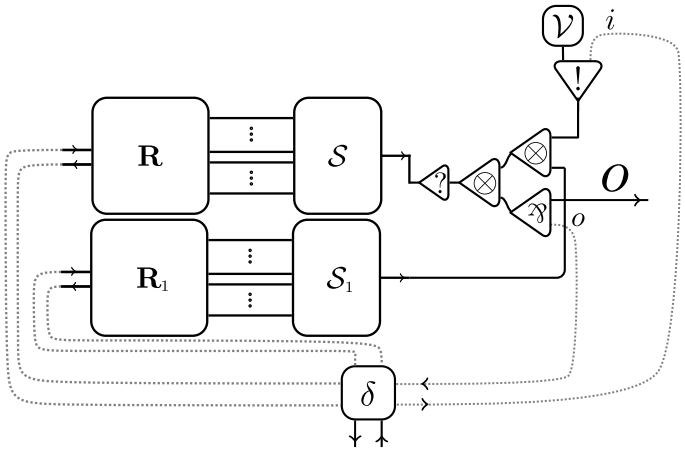
Lemma 5.3.5. Open terms separability
Let $\Gamma, x : A \vdash M : (\alpha, e)$ a λ_{cES} well-typed term, then its translation $M^\bullet$ is separable.

Proof. Lemma 5.3.5
We proceed by induction on terms.

Values As stated above, we can first observe that the translation of $*$, of a variable x and an abstraction all satisfy the separability conditions. Indeed, they are composed of a box with at least one auxiliary door, and the inside of the box is a normal form (by induction for abstraction and trivially for others). As they are pure terms, $m = n = 0$.

Get It is almost the same as values, except that the output o corresponding to the reference of get(r) is connected to a dereliction, which is allowed in **(b)**.

Application By induction, $M^\bullet$ and $N^\bullet$ are separable, thus can be decomposed in the following way :



We can merge the 3 routing areas $\mathbf{R}, \mathbf{R}_1$ and δ. All the inputs or outputs previously connected to one of the small areas immediately satisfy the conditions **(b,c)** by IH. The two remaining wires i and o are respectively connected to the auxiliary port of a par and the auxiliary port of a cocontraction, thus satisfy **(b,c)**. $\mathcal{S}$ and $\mathcal{S}_1$ are normal by IH, and the translation of $\mathcal{V}$ is easily seen as normal. O is the conclusion of a par, hence satisfies **(b,c)**. It only remains to see that the whole net excepted the routing areas is normal, but the only redex that could appear is the connection of a dereliction to $M^\bullet$. By **(c)**, the output of $M^\bullet$ is either the conclusion of an open box or a par which do not form a redex. The condition **(a)** is verified.

Parallel Similar to application

Variable substitution As for application, we apply the IH on $M^\bullet$ and $V^\bullet$, merge the routing areas, and just check that the connection of $V^\bullet$ to $M^\bullet$ cannot create new redexes using **(b)** on the output wire of $V^\bullet$.

Reference substitutions Again, the same technique is applied.

□

Finally, we can state the separability of normal forms:

Lemma 5.3.6. Normal form separability
The translation of a F-normal form is separable

It is proved as Lemma 5.3.5, by induction on the syntax of F-normal forms. We can eventually prove from this last lemma:

Lemma 5.3.7. Termination of $\lambda_{\text{cES}}+$
The translation of a F-normal form is strongly normalizing.

Proof. Lemma 5.3.7
We apply Lemma 5.3.6 to reduce the translation of a F-normal form F to a routing area $\mathbf{R}$ connected to $\mathcal{S}$ satisfying separability conditions. $\mathcal{S}$ and $\mathbf{R}$ are normal. The potential redexes must involve a wire at the interface of $\mathbf{R}$ and $\mathcal{S}$. The inputs of $\mathbf{R}$ are connected to $\mathcal{S}$, either to a par or to the conclusion of an open box and thus can't form any redex with (co)weakenings and (co)contractions of $\mathbf{R}$.

The outputs of $\mathbf{R}$ are either connected to the auxiliary port of a tensor, the auxiliary port of a cocontraction or to a dereliction. Only the latter may form a new redex. If the output of $\mathbf{R}$ is a coweakening, then everything reduces to $\mathbf{0}$. If it is just a wire, then the dereliction is connected through this wire to an input of $\mathbf{R}$ which again can't be part of any redex. The only remaining case is when the output of $\mathbf{R}$ is a cocontraction tree. Then we can perform the non-deterministic $\overset{\mathtt{ba}}{\to}$ reductions, and we go back to exactly the two previous cases as the dereliction is finally connected to a leaf of the tree of an input.

Hence, after we performed finitely many $\overset{\mathtt{ba}}{\to}$ reductions, we finally get a normal form, which is either $\mathbf{0}$, or a sum of the previous net $\mathbf{S}$ connected to simpler routing areas $\mathbf{R}_i$. □

From this last lemma follows Proposition 5.3.1.

5.3.3 Adequacy

The adequacy theorem states that the summands of the normal form of $M^\bullet$ are either the translation of a normal form $T = V_1 \parallel \ldots \parallel V_n$ (where $V_1, \ldots, V_n$ are values of λ_{cES}) that is a reduct of the original M, or garbage, that is a non correct net that corresponds to execution paths which deadlocked. We can recognize this garbage, thus eliminate it. With this additional operation, the summands of the normal form of $M^\bullet$ coincide with the translation of (parallel

of) values that are reachable by M. We stated a similar adequacy theorem between λ_{C} and λ_{cES} (Theorem 4.4.14) in Section 4.4.3.

Theorem 5.3.8. Adequacy
Let M be well-typed term of λ_{cES}. We write $\mathrm{Val}(M) := \{T = \|_i\ V_i \mid M \to^* T + M'\}$. Similarly, for a net $\mathcal{R}$ with normal form $\mathcal{N}$, we define $\mathrm{Val}(\mathcal{R}) = \{\mathcal{S} \mid \mathcal{N} = \mathcal{S} + \mathcal{S}',\ \mathcal{S}$ is the normal form of some $(\|_i\ V_i)^\bullet\}$. Then

$$\mathrm{Val}(M)^\bullet = \mathrm{Val}(M^\bullet)$$

where $\mathrm{Val}(M)^\bullet = \{N \mid V \in \mathrm{Val}(M),\ N$ is the normal form of $V^\bullet\}$.

Proof. By simulation, we know that if $M \to^* \sum_i M_i$ where the M_is are normal forms, then $M^\bullet \to^* \sum_i M_i^\bullet$. As parallel of values are translated to normal forms in nets, we have in particular $\mathrm{Val}(M)^\bullet \subseteq \mathrm{Val}(M^\bullet)$.

To check the converse, we have to check that the translations of the remaining M_is (the ones that are not parallel of values) cannot be reduced (as simple nets) to value nets, or do not generate new summands that themselves reduce to value nets. We can extract this fact from the proof of termination (cf Section 5.3.2) that shows that in the translation of a term which is not a value, the only remaining reduction are either the merging of routing areas, a reduction to $\mathbf{0}$, or a sequence of $\overset{\mathsf{ba}}{\to}$ which conserve the overall structure of a term. In particular, this does not produce any value during its reduction to a normal form in nets. □

5.4 Summary

In this chapter, we illustrated the use of the tools introduced in Chapter 2 and Chapter 3, a differential net system (Chapter 2) and routing areas (Chapter 3), by constructing an encoding from the concurrent λ-calculus with explicit substitutions λ_{cES} introduced in Chapter 4 in nets.

We use the monadic translation of Tranquilli [56] to encode types from the stratified type and effect system to standard formulas of multiplicative LL. We then translate terms as nets, with free wires corresponding both to variables and effect (references). We combine the "pure λ-calculus" part following the standard call-by-value translation in LL [42], and combine the wires corresponding to references through routing areas. These areas are specifically designed to encode the order of the evaluation of effects in λ_{cES}. This provides a modular translation of well-typed term of λ_{cES} in nets.

We then prove a simulation theorem, showing that nets implements the reduction of λ_{cES}. We show that the translation of a term is weakly normalizing. In the section on adequacy, we prove that the values in the normal form of the

translation of a term (which exists thanks to weak normalization and is unique by confluence) corresponds to the value in the normal form of the term in λ_{cES}: if we discard terms that deadlocked, the term and its translation computes the same values.

5.5 Discussion

Termination We believe that the result of weak normalization for the translation of well-typed terms of λ_{cES} may extend to a strong normalization result. This was the idea behind the (T_1) of Section 2.4: it ensures that weak normalization is actually sufficient to get strong normalization. The problem we encountered is that we use the $\xrightarrow{\mathtt{0}}$ reduction (or at least on of its avatar $\xrightarrow{\mathtt{n}}$, mentioned in Remark 3.1.7) in the reduction concerning routing areas such as application of the transit lemma , which potentially breaks the (T_1) property (Definition 2.4.3). However we use it in a restricted manner: we use it just to implement the neutrality of coweakening with respect to cocontraction (as does $\xrightarrow{\mathtt{n}}$). In practice, we always erase a summand that was produced by the previous duplication of an initial term as two summands, by a $\xrightarrow{\mathtt{ba}}$ reduction. This kind of reduction should not break the (T_1) property, as any potential looping subnets that is erased is still present in other summands.

Concurrency and correctness We mentioned from Chapter 2 that the translation of concurrent program may produce nets that do not respect the correctness criterion of differentials LL. A simple example is the following well-typed term of λ_{C}:

$$P = (\lambda x.\text{set}(r, x))\ \text{get}(r) \parallel (\lambda x.\text{set}(r, x))\ \text{get}(r)$$

This term does not reduce further, but it can be modified easily to normalize to a parallel of values: $P \parallel \text{set}(r, *)$, for example, reduces to $* \parallel * \parallel r \Leftarrow * \parallel r \Leftarrow * \parallel r \Leftarrow *$. This is a core example of a program that is translated to a net which has a cycle in one of its switching graphs (cf Section 3.1.3 for a review of the correctness criterion). The cycle correspond to a seemingly cyclic dependency in the term P: in the left thread, the x depends to the value $\text{get}(r)$ for which it will be substituted. Then, the $\text{set}(r, x)$ depends on this x. From the outside, the fact that the set is executed after the get is not visible. Thus, the right thread just see a term that is likely both read from and write to the reference r. Because of this, there is a dependence between the right $\text{get}(r)$ and the left $\text{set}(r, x)$. As the program is totally symmetric, we can see the emergence of a cycle: from the left $\text{get}(r)$, to the left $\text{set}(r, x)$, to the right $\text{get}(r)$, to the

right set(r, x), to the left get(r). This is not an actual cyclic dependency it the languages between some of the junctions are in fact mutually exclusives: the first get(r) being reduced must feed on an external assignment (as the program P alone can not reduce further), breaking the dependency with the adverse set(r, V). But this kind of dynamic property is hard to accounted in nets. This seems not to be specific to the languages we choose, as this example can be transposed to the π-calculus process $P' = a(x).\overline{a}\langle x\rangle \mid a(x).\overline{a}\langle x\rangle$ and thus already affect the translation presented in [14].

5.6 Perspective

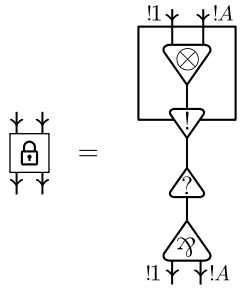
Typed and untyped terms The stratification constraint of the type system seems rather restrictive, especially from a practical point of view. It enforces strong normalization, which is a questionable feature for a realistic programming language. Tranquilli [56] showed that the stratification constraint precisely corresponds to the solvability of the equation given by the monadic translation, without resorting to recursive types. If one allows recursive types in nets, we can encode non stratified programs, and even untyped programs. The simulation result should also holds in this setting as it mainly relies on the operational semantic of nets, which does not depend on typing nor correctness. On the other hand, it is not obvious that an adequacy result would be achievable, and if it is, how. For this reason, we restricted ourselves to a fragment where these kind of properties may still be established. Concerning termination, a desirable extension of the termination theorem to the untyped case would be the preservation of strong normalization. This property, especially studied in explicit substitutions systems, ensures that the translation of a strongly normalizing term is itself strongly normalizing.

Synchronous and asynchronous effects In the design of the λ_{cES} language, we made some choices in order to get a more asynchronous system. This make the translation in nets easier. The two main such features are the cumulative stores (an assignment do not erase the previous ones, but rather add up), and the fact that the reduction is neither left-to-right nor right-to-left in the reduction of applications. The latter point is not really limiting, as one can encode a specific order of evaluation. Since the reduction is weak (no reduction under abstraction), it suffices to change an application $M\ N$ to $(\lambda x.x\ N)\ M$ to force a function to be fully evaluated before the argument, and get a left-to-right operational semantics.

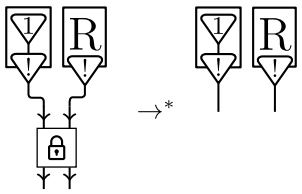
The cumulative stores are a more profound design choice. Usual programming languages with imperative references rather implement an erase-on-write semantics: in λ_C (cf Section 4.4.1), this would correspond to have exactly one store $r \Leftarrow V$ for each reference r, and to replace the (`set`) reduction rule by the following one :

$$(\texttt{set'}) \quad C[E[\mathrm{set}(r, V)] \parallel r \Leftarrow U \quad \rightarrow \quad C[E[*]] \parallel r \Leftarrow V$$

Implementing this kind of behavior in nets requires a form of synchronization. Tranquilli has sketched some ideas about it [54]. He proposes the use of an additional wire for each references that represent a locking signal. $\mathrm{get}(r)$ and $\mathrm{set}(r, V)$ must acquire this lock before reading or writing values. While he resorts to second-order constructs (the idea of a transistor already used by Ehrhard and Laurent to encode prefixing in π-calculus [14]) to implement the lock, we could implement it directly in our nets thanks to our reduction operating only on closed boxes. Here is a proposal of a lock mechanism in nets. This lock has two input wires (top) and two output wires (bottom), where the left input and output correspond to the locking signal while the right input and output correspond to the data we want to tie with the locking signal:



The locking signal does not carry any information, but only encode the precedence of memory operations: this is why it is typed by $\texttt{Unit}^\bullet = !\mathbf{1}$. The lock multiplexes the locking signal with the data with a tensor, and this pair is then enclosed by an exponential box. This box only serve as a synchronization device, and is directly connected to a dereliction in order to open it, followed by a par which demultiplexes the two inputs. It the reduction were not restricted to closed boxes, then the lock could be instantly reduced to two wires and would just act as the identity on both its inputs, independently of what it is connected to. However, with a reduction restricted to closed boxes, the lock's box must be closed (id est being without auxiliary ports) before being opened. This means that as long as both the inputs are not connected to a closed exponential box, then the lock prevent any interaction between what is connected to its inputs and what is connected to its outputs. When both its inputs are closed exponential boxes, the lock can be removed through the following reduction:



Here is a sketch of how $\mathrm{set}(r, V)$ and $\mathrm{get}(r)$ would be encoded using this lock in an erase-on-white setting:

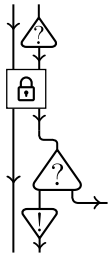


Figure 5.9: Encoding of $\mathrm{get}(r)$

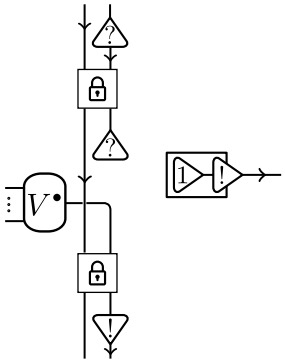


Figure 5.10: Encoding of $\mathrm{set}(r, V)$

The left part corresponds of each encoding to the locking signal, while the right part correspond to the data being effectively exchanged through references. Both encoding start with a dereliction connected to the incoming data, stripping away the additional exponential layer of values stored in references (cf the paragraph on get and reference substitutions in Section 5.1.3). Then, in both case, the lock imposes that the $\mathrm{get}(r)$ or the $\mathrm{set}(r, V)$ acquire the locking signal, meaning that they must receive a !1 on the left input, before anything else can happen. For $\mathrm{set}(r, V)$, a second lock prevent $V^\bullet$ from being emitted asynchronously, as the first lock have not control on it. When this condition is met, the lock can be removed and the reductions corresponding to copying/sending data may take place. As long as the lock vanishes, the locking signal is consequently re-emitted through the appropriate wire, making it available for the next memory operation to acquire it.

Note that in our current setting of cumulative stores, $\mathrm{get}(r)$ and $\mathrm{set}(r, V)$ sends messages asynchronously: a $\mathrm{get}(r)$ produces a dummy output through a coweakening, while a $\mathrm{set}(r, V)$ ignore the input through a weakening. Here, a term really takes the current state of the whole store as an argument, and return the new state of the store. In particular, $\mathrm{get}(r)$ duplicate the input data, but also re-emits it through the reference output wire. The $\mathrm{set}(r, V)$, by replacing the output with its own data, effectively erases any previous assignment. In this hypothetical system, the additional exponential layer introduced by an exponential box in the translation of reference substitutions is rather introduced here by

a codereliction instead, as in the translation of the finitary π-calculus of [15]. The wire corresponding to data would be all connected through communication areas, that is routing areas that allow everyone to communicate with everyone. On the other side, the connection of signals must be done using specific routing areas that precisely implement the ordering of effects, as our δ area, for example. We believe that λ_{cES} would also be adaptable to such a setting, by mimicking the mechanisms we sketched for nets on the calculus side.

Expressivity of the type system Another direction for future work is to enrich the type system of λ_{cES} and λ_{C}. LL can indeed accommodate a lot of possible extensions: second order (polymorphism), sum types (the additive fragment), recursive types (by adding fixpoint operators), *etc.* One can also add ad-hoc cells and reduction rules in nets to implement additional constants and primitives for the languages, such as integer or floating point arithmetic for example. Depending on the feature, it is not clear yet how it interact with the current setting and if it has a natural representation in nets.

A parallel abstract machine We stressed that we hope to benefit from the ability of nets to enable independent computations to be done in parallel. We think of the GoI abstract machine for PCF introduced in [36] in particular. An extension of GoI for differential nets can be found in [11]. Taking inspiration from it, we could adapt the multi-token machine of [36] to work on nets. This would provide an interesting basis for a prototype implementation. Automatic parallelism, sometimes referred as *implicit parallelism*, is not the only benefit of a such abstract machine. The natural *message-passing* style of execution makes it very easy to slice a program into arbitrary components and execute them on different computing unit. This is demonstrated for example in [16], which introduces a PCF-like language enriched with an explicit construct to defer the computation of subterms on different nodes. The corresponding machine is GoI-based one.

Bibliography

[1] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Levy. Explicit substitutions. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '90, pages 31–46, New York, NY, USA, 1990. ACM.

[2] Beniamino Accattoli. Proof nets and the call-by-value λ-calculus. *Theor. Comput. Sci.*, 606(C):2–24, November 2015.

[3] Beniamino Accattoli, Eduardo Bonelli, Delia Kesner, and Carlos Lombardi. A nonstandard standardization theorem. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 659–670, New York, NY, USA, 2014. ACM.

[4] Beniamino Accattoli and Delia Kesner. The structural λ-calculus. In Anuj Dawar and Helmut Veith, editors, *Computer Science Logic: 24th International Workshop, CSL 2010, 19th Annual Conference of the EACSL, Brno, Czech Republic, August 23-27, 2010. Proceedings*, pages 381–395, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[5] Beniamino Accattoli and Ugo Dal Lago. (leftmost-outermost) beta reduction is invariant, indeed. *Logical Methods in Computer Science*, 12(1), 2016.

[6] Roberto M. Amadio. On stratified regions. In Zhenjiang Hu, editor, *Programming Languages and Systems: 7th Asian Symposium, APLAS 2009, Seoul, Korea, December 14-16, 2009. Proceedings*, pages 210–225, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[7] A. Asperti, V. Danos, C. Laneve, and L. Regnier. Paths in the lambda-calculus. three years of communications without understanding. In *Proceedings Ninth Annual IEEE Symposium on Logic in Computer Science*, pages 426–436, Jul 1994.

[8] Gérard Boudol. Typing termination in a higher-order concurrent imperative language. *Information and Computation*, 208(6):716 – 736, 2010. Special

Issue: 18th International Conference on Concurrency Theory (CONCUR 2007).

[9] Pierre-Louis Curien, Thérèse Hardin, and Jean-Jacques Lévy. Confluence properties of weak and strong calculi of explicit substitutions. *J. ACM*, 43(2):362–397, March 1996.

[10] Vincent Danos and Laurent Regnier. The structure of multiplicatives. *Archive for Mathematical Logic*, 28(3):181–203, Oct 1989.

[11] Marc De Falco. The Geometry of Interaction of Differential Interaction Nets. In *Logic In Computer Science (LICS)*, pages 465–475, Pittsburgh, United States, June 2008. 20 pagee, to be published in the proceedings of LICS08.

[12] Ugo de'Liguoro and Adolfo Piperno. Non deterministic extensions of untyped lambda-calculus. *Inf. Comput.*, 122(2):149–177, 1995.

[13] Thomas Ehrhard and Olivier Laurent. Acyclic solos and differential interaction nets. *Logical Methods in Computer Science*, 6(3):11, 2010.

[14] Thomas Ehrhard and Olivier Laurent. Interpreting a finitary pi-calculus in differential interaction nets. *Information and Computation*, 208(6):606 – 633, 2010. Special Issue: 18th International Conference on Concurrency Theory (CONCUR 2007).

[15] Thomas Ehrhard and Laurent Regnier. Differential interaction nets. *Theoretical Computer Science*, 364(2):166–195, November 2006. 30 pages.

[16] Olle Fredriksson and Dan R. Ghica. Seamless distributed computing from the geometry of interaction. In Catuscia Palamidessi and Mark D. Ryan, editors, *Trustworthy Global Computing*, pages 34–48, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[17] M. J. Gabbay and A. M. Pitts. A new approach to abstract syntax involving binders. In *Logic in Computer Science*, pages 214–224. IEEE Computer Society Press, 1999.

[18] Dan R Ghica. Geometry of synthesis: a structured approach to vlsi design. In *ACM SIGPLAN Notices*, volume 42, pages 363–375. ACM, 2007.

[19] Dan R Ghica and Alex Smith. Geometry of synthesis ii: From games to delay-insensitive circuits. *Electronic Notes in Theoretical Computer Science*, 265:301–324, 2010.

[20] Dan R Ghica and Alex Smith. Geometry of synthesis iii: resource management through type inference. In *ACM SIGPLAN Notices*, volume 46, pages 345–356. ACM, 2011.

[21] Dan R Ghica, Alex Smith, and Satnam Singh. Geometry of synthesis iv: compiling affine recursion into static hardware. In *ACM SIGPLAN Notices*, volume 46, pages 221–233. ACM, 2011.

[22] Jean-Yves Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Univ. Paris 7, 1972.

[23] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50(1):1 – 101, 1987.

[24] Jean-Yves Girard. Towards a geometry of interaction. *Contemporary Mathematics*, 92(69-108):6, 1989.

[25] Jean-yves Girard. Proof-nets: The parallel syntax for proof-theory. In *Logic and Algebra*. Citeseer, 1996.

[26] Timothy G Griffin. A formulae-as-type notion of control. In *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 47–58. ACM, 1989.

[27] Kohei Honda and Olivier Laurent. An exact correspondence between a typed pi-calculus and polarised proof-nets. *Theor. Comput. Sci.*, 411(22-24):2223–2238, 2010.

[28] Naohiko Hoshino, Koko Muroya, and Ichiro Hasuo. Memoryful geometry of interaction: from coalgebraic components to algebraic effects. In *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, page 52. ACM, 2014.

[29] William A Howard. The formulae-as-types notion of construction. 1980.

[30] Eugen Jiresch. Towards a gpu-based implementation of interaction nets. *arXiv preprint arXiv:1404.0076*, 2014.

[31] Wolfram Kahl. A simple parallel implementation of interaction nets in haskell. *arXiv preprint arXiv:1504.02603*, 2015.

[32] Delia Kesner. A theory of explicit substitutions with safe and full composition. *Logical Methods in Computer Science*, 5, 2009.

[33] Jean-Louis Krivine. Typed lambda-calculus in classical zermelo-fraenkel set theory. *Archive for Mathematical Logic*, 40(3):189–205, 2001.

[34] Yves Lafont. Interaction nets. In *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 95–108. ACM, 1989.

[35] U. D. Lago, R. Tanaka, and A. Yoshimizu. The geometry of concurrent interaction: Handling multiple ports by way of multiple tokens. In *2017 32nd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–12, June 2017.

[36] Ugo Dal Lago, Claudia Faggian, Benoit Valiron, and Akira Yoshimizu. Parallelism and synchronization in an infinitary context. In *Proceedings of the 2015 30th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, LICS '15, pages 559–572, Washington, DC, USA, 2015. IEEE Computer Society.

[37] Peter J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, January 1964.

[38] Pierre Lescanne and Jocelyne Rouyer-degli. Explicit substitutions with de bruijn's levels. In *in Rewriting Techniques and Applications, 6th International Conference, Lecture Notes in Computer Science 914*, pages 294–308. Springer, 1995.

[39] Ian Mackie. *Applications of the Geometry of Interaction to language implementation.* PhD thesis, Univ. of London, 1994.

[40] Ian Mackie. Compiling process networks to interaction nets. In *Proceedings 9th International Workshop on Computing with Terms and Graphs, TERMGRAPH 2016, Eindhoven, The Netherlands, April 8, 2016.*, pages 5–14, 2016.

[41] Antoine Madet. *Complexité Implicite de Lambda-Calculs Concurrents.* Theses, Université Paris-Diderot - Paris VII, December 2012.

[42] John Maraist, Martin Odersky, David N. Turner, and Philip Wadler. Call-by-name, call-by-value, call-by-need, and the linear lambda calculus. *Electronic Notes in Theoretical Computer Science*, 1(Supplement C):370 – 392, 1995. MFPS XI, Mathematical Foundations of Programming Semantics, Eleventh Annual Conference.

[43] Damiano Mazza. *Interaction nets : semantics and concurrent extensions*. PhD thesis, 2006. Thèse de doctorat dirigée par Régnier, Laurent Mathématiques discrètes et fondements de l'informatique Aix Marseille 2 2006.

[44] Damiano Mazza. The true concurrency of differential interaction nets. *Mathematical Structures in Computer Science*, FirstView:1–29, 11 2016.

[45] Paul-André Mellies. Typed λ-calculi with explicit substitutions may not terminate. In Mariangiola Dezani-Ciancaglini and Gordon Plotkin, editors, *Typed Lambda Calculi and Applications: Second International Conference on Typed Lambda Calculi and Applications, TLCA '95 Edinburgh, United Kingdom, April 10–12, 1995 Proceedings*, pages 328–334, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.

[46] Koko Muroya and Dan R Ghica. The dynamic geometry of interaction machine: a call-by-need graph rewriter. *arXiv preprint arXiv:1703.10027*, 2017.

[47] Koko Muroya, Naohiko Hoshino, and Ichiro Hasuo. Memoryful geometry of interaction ii: recursion and adequacy. In *ACM SIGPLAN Notices*, volume 51, pages 748–760. ACM, 2016.

[48] Jorge Sousa Pinto. Sequential and concurrent abstract machines for interaction nets. In *International Conference on Foundations of Software Science and Computation Structures*, pages 267–282. Springer, 2000.

[49] John C. Reynolds. Towards a theory of type structure. In *Programming Symposium, Proceedings Colloque Sur La Programmation*, pages 408–423, Berlin, Heidelberg, 1974. Springer-Verlag.

[50] Kristoffer Høgsbro Rose. Explicit cyclic substitutions. In Michaël Rusinowitch and Jean-Luc Rémy, editors, *Conditional Term Rewriting Systems: Third International Workshop, CTRS-92 Point-à-Mousson, France, July 8–10 1992 Proceedings*, pages 36–50, Berlin, Heidelberg, 1993. Springer Berlin Heidelberg.

[51] Ulrich Schöpp. On interaction, continuations and defunctionalization. In *International Conference on Typed Lambda Calculi and Applications*, pages 205–220. Springer, 2013.

[52] François-Régis Sinot, Maribel Fernández, and Ian Mackie. Efficient reductions with director strings. In Robert Nieuwenhuis, editor, *Rewriting*

Techniques and Applications: 14th International Conference, RTA 2003 Valencia, Spain, June 9–11, 2003 Proceedings, pages 46–60, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[53] W. W. Tait. Intensional interpretations of functionals of finite type. *Journal of Symbolic Logic*, 32(2):198–212, 1967.

[54] Paolo Tranquilli. References, Multithreading and Differential Nets.

[55] Paolo Tranquilli. *Nets between determinism and nondeterminism.* PhD thesis, 2009. Thèse de doctorat dirigée par Tortora de Falco, Lorenzo et Bucciarelli, Antonio Informatique Paris 7 2009.

[56] Paolo Tranquilli. Translating types and effects with state monads and linear logic. 14 pages, January 2010.

[57] Paolo Tranquilli. Intuitionistic differential nets and lambda-calculus. *Theoretical Computer Science*, 412(20):1979 – 1997, 2011. Girard's Festschrift.

[58] Alan Mathison Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London mathematical society*, 2(1):230–265, 1937.