Université Paris Diderot — Paris VII
Sorbonne Paris Cité

École Doctorale Sciences Mathématiques de Paris Centre

THÈSE
en vue d'obtenir le grade de
DOCTEUR DE L'UNIVERSITÉ PARIS DIDEROT
en Informatique Fondamentale

Une Dialectica Matérialiste

~

A Materialist Dialectica

Présentée et soutenue par
Pierre-Marie PÉDROT
le 17 septembre 2015

devant le jury composé de :

Hugo Herbelin	Directeur de thèse
Alexis Saurin	Directeur de thèse
Alexandre Miquel	Rapporteur
Thomas Streicher	Rapporteur
Andrej Bauer	Examinateur
Delia Kesner	Examinatrice
Colin Riba	Examinateur

Acknowledgements

> *« À celle-là seule que j'aime et qui le sait bien. »*
>
> Dédicace commode, que je ne saurais trop recommander à mes confrères. Elle ne coûte rien, et peut, du même coup, faire plaisir à cinq ou six personnes.

Alphonse Allais about acknowledgements.

La maîtrise du subtil art des remerciements ne faisant, hélas ! pas partie des cordes de mon fusil-mitrailleur, je commencerai diplomatiquement en recourant à la logique, et dédierai donc cette thèse à toutes les personnes auxquelles elle n'est pas dédiée. C'est ma foi une dédicace tout aussi commode que celle d'Alphonse Allais, sinon plus, et je la recommande tout autant à la palanquée de confrères confrontés au même dilemme : mêmes causes, mêmes effets. Cette adresse générale étant expédiée, tournons-nous dès à présent vers les remerciements, attentions et autres hommages personnels.

Autem P = NP, cujus rei demonstrationem mirabilem sane detexi. Hanc marginis exiguitas non caperet.

Il me faut commencer par remercier chaleureusement Alexis, directeur de thèse sans précédent pour un doctorant du même. Je lui suis infiniment[1] gré de sa bonne humeur, de sa capacité à supporter les pires adversités, de sa volonté infaillible et de son dévouement sans bornes. Sans présumer du fait qu'il a réussi à me supporter quatre longues années durant, exploit périlleux qui immanquablement confine au suicide ou à l'homicide chez toute personne raisonnable[2]. De là à conclure ce que l'on veut en conclure, il y a un fossé ravineux dans lequel aurait pu se noyer le régiment des désillusions du couple d'un thésard fraîchement émoulu et de son directeur primo-directant. D'où, théorème, π est rationnel. Non, je déconne. Le plus sérieusement du monde cette fois, j'affirmerai haut et fort, la main[3] posée sur la poitrine du côté où palpite le myocarde que je ne serais probablement pas arrivé là où je suis sans lui. *Asylum asylum invocat.* Je ne pourrais jamais le remercier à la hauteur de ce qu'il m'a donnné.

Constatant avec une certaine suspicion que cet hommage commence à avoir de faux airs d'éloge funèbre, je propose au lecteur lassé de lire de longues louanges au lyrisme lénifiant de se tourner vers la personne d'Hugo Herbelin, deuxième directeur de thèse et troisième mousquetaire de cette bande des quatre[4]. Expert ès arts arcanes de la théorie

[1] Conjecturons hardiment au moins le cardinal du continu.

[2] On entend évidemment par là toute personne qui satisfait aux prémisses du théorème d'incomplétude de Gödel.

[3] Gauche, *what else*.

[4] Alexis, Hugo, moi-même et la Science. Ou les neuroleptiques. Je ne suis pas encore bien fixé.

des types, disciple chevronné du Curry-Howard et maniant le λ-terme comme nul autre, Hugo Herbelin est un aventurier des temps modernes qui s'ignore, malmené par notre monde gris et post-moderne qui exalte le banquier voleur de peuples et conchie l'intègre homme de science. Il m'a transmis un vaste savoir qui s'étend de la compréhension des modèles de Kripke mais pas que, à la capacité de survivre à une exposition aiguë au entrailles de Coq qui exhalent des effluves de pourriture capables de tuer un thésard imprudent à dix mètres. Je m'efforcerai de réduire cette connaissance au *kôan* suivant.

> Un thésard demanda à Hugo Herbelin : « Coq est-il cohérent ? »
> `Anomaly: Uncaught exception Mu.`

Par malheur, l'histoire omet de raconter si l'étudiant fut illuminé ou bien s'il l'était déjà à l'instant même où il songea à passer un doctorat d'informatique fondamentale[5].

Il convient de remercier Alexandre Miquel et Thomas Streicher d'avoir accepté de lire avec attention ma prose, torture à laquelle je ne me serais assurément pas soumis si d'aventure je m'étais trouvé à leur place. Grâce leur en soit rendue. Profitons au passage pour remercier en particulier Alexandre qui, à force de coups de pied dans une partie charnue que la décence défend de nommer expressément, sut me convaincre de trouver un directeur de thèse en ce lointain été de l'an I du Canapé[6].

J'espère que le lecteur apprécie les notes en bas de page et dans la marge. Sinon, tant pis pour lui.

Un grand merci aux personnes qui ont accepté d'être membre de mon jury avec bienveillance : merci à Delia, à Colin qui sans le savoir m'a lancé sur le sujet de ma thèse au cours de cette fameuse soirée de TYPES 2013, et à Andrej dont la prose m'a inspiré et qui a bien voulu m'inviter à Ljubljana.

Mention spécialement remarquable et remarquablement spéciale à quelques permanents de PPS qui durant ces quatres années m'ont croisé bien trop souvent rôdant dans les couloirs, l'air hagard du sociopathe en quête de victime flottant sur le visage, et qui n'ont pas contacté les autorités compétentes malgré l'apparente imminence du danger. Je remercierai donc Yann, l'homme à la journée de 48 heures, vaillant Achille qui sut tenir tête à ceux du laboratoire d'en haut dont-on-ne-doit-pas-prononcer-le-nom et fin connaisseur des rouages fangeux de l'implacable machine administrative. Coup de chapeau à Pierre L., Sisyphe de la gestion du système Coq et du reste, et qui sous des atours ulmiotes cache une personne de plus haute confiance. Merci à Pierre-Louis, diplomate devant les diplomates et roc cartésien qui a su préserver l'équipe πr^2 des petits caprices de l'Inria. Mes excuses à Matthieu, dont j'ai pourri le *rebase* de sa branche des univers polymorphes un certain nombre de fois. Révérence à Jean, agent double faisant de l'entrisme chez les biologistes avec maestria. Salutations à Juliusz mot-compte-triple, prestidigitateur capable d'apparaître à loisir dans les bureaux des thésards et de semondre le principe glandesque[7] hors du néant éthéré. Remerciements à Paul-André d'être

[5] Par contre, l'histoire précise bien que le rapport de bug qui s'en suivit fut marqué WontFix par un certain `kgoedel`. Ce n'est pas dans le privé qu'on verrait ça.

[6] `GOTO 6.`

[7] Il est de notoriété publique que le principe glandesque se manifeste le plus souvent sous la forme d'un liquide ectoplasmique pétillant et riche en esprit éthylique.

Paul-André[8].

Merci à Odile pour sa gentillesse, pour son efficacité et pour sa résistance aux tourments qu'a pu lui donner un gougnafier tel que moi.

Il me faut maintenant remercier la nuée de thésards désabusés et la horde d'innocents stagiaires qui ont dû un jour ou l'autre supporter mes délires verbiageux et ma logorhée généralement peu diplomatique. Je pense tout particulièrement aux différentes personnes qui se sont vues dans l'obligation horrifiée de partager mon bureau, c'est-à-dire pas mal de populace l'air de rien : Matthias, Pierre B., Lourdes, Guillaume C., Philipp, Iovana, Cyprien, Hugo, Béa, et avec probabilité 1 d'autres gens dont l'identité a été *garbage collectée* trop promptement par ma mémoire défaillante. Je plains sincèrement le noyau dur formé de Pierre, Lourdes et Guillaume qui ont très certainement fomenté à un moment ou à un autre un complot d'assassinat sur ma personne. Je remercie aussi les thésards des bureaux d'à côté et apparentés. Merci donc à Guillaume M., maître de la petite remarque acerbe, Shahin, générateur de bruit aléatoire dans les couloirs sans fin, Ioana, d'une patience sans fond, Flavien, compagnon d'infortune, Étienne, demi-frère de sang, Hadrien, au moins aussi bringue que moi, Thibaut, spectateur impuissant, Matthieu, sauvé des eaux, Pierre V. « où suis-je », Charles l'inénarrable, Marie Ar Ruz, Amina, demi-sœur de sang de l'autre côté, Yann les-balles-sont-des-dollars, Ludovic, Cyrille et Maxime, collectivement connus sous le nom du « bureau qui fait des maths » et tous les autres dont j'ai la flemme de citer le nom. Toute personne se trouvant dans cette dernière catégorie a effectivement le droit moral de m'envoyer un petit colis chargé en poudre de bacille du charbon, sous réserve d'être affranchi selon les tarifs en vigueur[9].

Je remercie l'immortel groupe de travail logique dans toute sa généralité et plus particulièrement, en sus des déjà nommés, Marc, Adrien, Gabriel et Aloïs et la triple Karmeliet du *Pantalon*, compagnons de route d'une époque que les moins de vingt ans ne peuvent pas connaître.

Outre les personnes déjà citées, je tiens à remercier l'équipe de développement de Coq au sens large, qui a su m'apporter une certaine forme d'amusement aussi bien à travers les gros teuyaux de l'interweb que par les nombreuses occasions de se rencontrer. Merci à Arnaud Spiwack, Maxime Dénès, Enrico Tassi, Guillaume Melquiond, Yves Bertot, Assia Mahboubi, Pierre Courtieu, Jason Gross... Je remercie aussi les grands anciens d'avoir donné la vie à un logiciel si fantastique.

Même si cela les laissera prouvablement de marbre, je voudrais remercier Jean-Yves et Jean-Louis, mythiques fondateurs de paradigmes dont cette thèse est une continuation toute linéaire[10].

[8] **UIP** or not **UIP**, that is the question.

[9] Paris et Seine-et-Oise, 1 fr. Départements, 1 fr 50. Étranger, 2 fr.

[10] Celui qui me dira que cet hommage est trop classique recevra mon pied dans le fondement de l'informatique.

Merci à Inria d'être l'inventeur du monde numérique. S'il eût été plus court, toute la face de la terre aurait changé.

Merci à l'Université Paris Diderot pour leurs innovations architecturales audacieuses. Je gage que d'ici peu, quarante siècles nous contempleront du haut du bâtiment Sophie Germain.

Passons à l'instant à des remerciements plus personnels.

Bien évidemment, je remercie mes parents de ne pas m'avoir jeté aux ordures à la naissance, acte puni trop sévèrement par la loi que j'aurais par ailleurs tout à fait compris et pardonné *a posteriori*. Il est bien probable qu'ils le regrettent fortement après toutes ces années passées à résister à la tentation régulière de pratiquer la strangulation sur leur fils indigne. Je leur tiendrais donc grâce de tant d'endurance, et les remercie de leur affection renouvelée.

Je souhaiterais dédier ce paragraphe à Daniel Hirschkoff, un enseignant comme il en existe trop peu et sans lequel je ne serais probablement pas en train d'écrire des remerciements aussi stupides. Car les fonctions sont des valeurs, bordel. Plus généralement, je voudrais étendre cet hommage ému à toutes les personnes qui ont eu une influence décisive sur ma passion pour les sciences. Un merci ineffable à Joël, et un salut cordial à des enseignants qui m'ont marqué, de Patrice à M. Duval en passant par M. Morize.

Merci à Sarica, sans nul doute la meilleure prof d'espagnol du monde.

Je remercie les colocataires qui ont survécu à la cohabitation avec votre serviteur. Chante ô déesse, le courroux du Poulyide Amaury, à propos de la vaisselle non faite. Ainsi fis-tu, Mikaël, du sac de poissons. Philippe, je sais où tu te caches. Laure, allegro ma non troppo. Hélène, qui déteste Paris. Et puis les autres, Alex, Proux, Tim, ombres fugitives du sauna. Ô temps, suspends ton vol !

High-five lolz pour les membres remarqués d'Impega, la mailing-liste ultime, fière et noble descendante très *select* de la liste L3IF. Salut donc bien bas à Gaupy « Sac-à-vin », J.-M. « Mad » Madiot, Gallais « la Rage », Sylvain « Qu'elle était belle ma Berjalie », Matthieu « Pipe au miel », Val « le Cylon », Hugo « la Pute », Marthe « Mangez des pommes », Richard « Caca » Griffon, Yann « l'Entrepreneur », et à notre totem « le pigeon qui mange du caca », déité trop souvent méprisée mais pourtant ô tant porteuse de sens dans un monde à la dérive qui se cherche.

Je voudrais finir en remerciant l'univers, grand oublié des longues litanies des remerciements qu'on trouve usuellement en préface des thèses. Il est clair que sans lui, on s'emmerderait ferme.

Contents

1 Introduction

— If she weighs the same as a duck...
— She's made of wood.
— And therefore?
— A witch!

Monty Python about the consistency of arithmetic.

Who could have thought that Aristotle's rules of reasoning would have found their way to the tools ensuring that planes do not crash because of a programming bug? This simple anecdote shows how highly theoretical works may eventually apply to very concrete problems, even if this requires to wait for two millenia and a half.

With the advent of an ubiquitary computerization of the human society, logic is one of those research fields that have been deeply transformed. Its metamorphosis is not so much due to the increased computing power at our disposal, that makes possible to perform instantly tasks that would have taken ages a few decades ago, as this is the case for physical simulations for instance. It is rather because logic turned out to share an essential kinship with computer science, up to the point that one could safely assert that *computer science is the fabric of logic*. This is not a vacuous claim, and we will support it with evidence.

From the syllogisms of Aristotle to the bug-free programs, there is quite a journey, that we would like to sketch here. As for any science, it is not a regular-paced trip where discoveries would take the rôle of evenly placed milestones. Rather, this is the story of a sudden explosion fueled by a convergence of ideas that would have not been possible any time before.

Alas! Poor reader! For we shall not spare her from the overused but necessary etymological recollection of the word *logic*, which dangerously wanders on the fringes of the realm of *clichés*! Still, we believe that recalling that it stems from the Greek word *logos*, meaning in turn *word*, is an important starting remark. This renders explicit the fact that logic is, in the first place, the science of language. This is how Greek philosophers, amongst who Aristotle, thought of it, and this is even more obvious in Aristotle's magnum opus, the *Organon*, whose purpose was to provide clear rules justifying the truthness of discourses.

We should acknowledge here that Greek philosophers did realize that language was not so amenable as it looked like at first sight. The liar's paradox, for instance, can be traced back to the 4^{th} century BC. In its rawest form, it can be presented as the following sentence.

This sentence is false.

If this sentence were to be true, then it would imply that it is false, and conversely. This is indeed paradoxical, and demonstrates that one should distrust the expressive power of languages when coming to consistency.

Notwithstanding the advances in logic for more than two millenia, that were chiefly achieved for philosophical or theological purposes, let us fast-forward until the birth of logic as a true science, by the end of the 19^{th} century. While up to then, mathematicians were working in a mostly informal parlance, mathematics underwent a drastic change in the course of a handful of decades. Concerned about the scientific rigor of their object of study, mathematicians sought to provide it with a firm, irrefutable ground, in what would turn out to be the quest for the foundations of mathematics. This quest mobilized the efforts of dozens of scholars for more than half a century. Acknowledging them all would be impossible here, so that, apologizing in advance for our unfairness, we will only recall some of them whom we find peculiarly representative.

We should start by naming one of the most important member of this then-nascent trend, Georg Cantor. He bestowed mathematics with the so-called *paradise* of set theory[1], an axiomatic system allowing for a formal presentation of the objects manipulated by the mundane mathematician [26]. Unluckily for him, his original presentation proved to be flawed, and let pass through various proofs of the absurdity. One of the simplest such paradoxes is due to Russell [99], and is actually a set-theoretical variant of the liar paradox, which arises when considering the set

$$X := \{x \mid x \notin x\}$$

because it is easy to see that both $X \in X$ and $X \notin X$, leading to a contradiction. Exactly as for the liar's paradox, this is a case of careless self-reference. Fixing these issues required quite an amount of work and led, amongst others, to the modern set theory as we know it [2].

While set theory aims at describing the mathematical objects in a formal way, it is somehow agnostic about the underlying logic, and in particular about what a proof is. This question, although pertaining to the same foundational quest, is clearly distinct. Albeit already present in germ in Aristotle's syllogisms, the issue would not be tackled formally until Boole and later Frege.

Boole, some three decades before the proper beginning of the foundational quest, proposed a system based on what we would call today Booleans [22]. It would assign to each proposition a truth value, reduced to two possibilities, either true or false, paving the way for electrical circuits. Instead, Frege devised a graphical system, the *Begriffsschrift* [42], allowing to represent a proof as a tree-like structure, in a fashion close to the modern proof theory. The two approaches are sharply different. In Boole's system, proofs do not exist as proper objects, and the logical content of a formula is amalgamated to its *validity*. In this respect, Frege's *Begriffsschrift* is more fine-grained, because it makes

[1] This expression is due to Hilbert.

explicit the proof justifying a formula as a first-class object, emphasizing the notion of *provability* over validity. This contrasted the opposition between truth, a semantical notion, and provability, a syntactical one.

At the very beginning of the 20^{th} century, Hilbert proposed to the world his now famous twenty-three problems for the next hundred years to come. They were open problems of that time whose importance was deemed sufficient enough to be considered as landmarks for the dawning century. Most notably, the second problem was to prove that arithmetic was consistent. In Hilbert's mind, proving it would have somehow marked the end of the quest for the foundations, because arithmetic was then thought to be a *complete* basis for all mathematics. This was at the core of the so-called Hilbert's program for the refoundation of mathematics, embodied by his claim: "In mathematics there is no *ignorabimus*".

Much to Hilbert's dismay, things went awry thirty years later, when Gödel formally proved that this hope was a chimera. Elaborating on the evergreen liar's paradox, Gödel showed that any reasonable consistent logical system featured a formula which was not provable in this system, nor was its negation. As arithmetic fitted the requirements of reasonableness, this theorem washed ashore the idea that arithmetic would encode all mathematics. Such a formula Φ, neither provable, nor disprovable, called *independent* from the system, can be built just like the Greek liar.

$$\Phi := \text{“}\Phi \text{ is not provable.”}$$

The exact construction of the above formula relied on a trick that was novel at that time, but that would seem nowadays obvious to a computer scientist. Indeed, Gödel needed to turn formulae into numbers, a process which we would call in modern terminology *digitizing* and which is at the heart of our everyday's computers. The notion of reasonable, which we did not define, is also worth discussing from the computer scientist's point of view. Informally, a system is reasonable if its proofs can be mechanically checked by a computer. It is quite remarkable that Gödel's proof, published in 1931, even predates the formalization of what a computer is, due to Turing in 1936!

Actually, the picture is even worse. Gödel also proved that the consistency of a reasonable system was independent from this system. This was a hard stroke. To prove the consistency of arithmetic thus required a system strictly more powerful than arithmetic itself. But the consistency of that system would require in turn a more powerful one, and so on, effectively requiring consistent systems *all the way down*.

It seems that Gödel tried to work around this inherent limitation of logic by shifting its point of view over consistency. This is particularly visible in his subsequent work, including the double-negation translation [50] (1933) and the Dialectica translation [51] (published in 1958, but designed in the 30's). Both translations were trying to reduce consistency of arithmetic to a computational property. These results were drawing from the *intuitionism* that Brouwer had been advocating. Rather than a purely formal game consisting in applying trusted rules, Brouwer opposed to Hilbert and defended the fact that logic needed to be rooted in a constructivist approach, effectively building up the

objects it described. Brouwer rejected in particular the principles of classical logic that allowed to create objects that could not be made explicit.

Instead of relying on abstract, static logical principles, this allowed to base the logical content of a proof on the dynamics of an effective procedure hidden inside the latter. Contrarily to axioms, algorithms are objects one can handle and run, giving much more insight on what is going on in the proof. This eventually led to the so-called realizability techniques, as developed by Kleene for instance [65].

The final step of this history needed a few years to fully mature, and is in part due to the growth of programming languages on actual computers from the end of the first half of the 20^{th} century onwards. In 1958, Curry, and then Howard in 1969, realized that a well-chosen representation of proofs of intuitionistic logic was in one-to-one correspondence with a typed subset of programs from the λ-calculus, a programming language that had been designed by Church in the 30's [29], and the common ancestor of the phylum of functional languages. Pushing forward the ideas sketched by intuitionism, it showed that not only programs could be extracted from proofs through realizability, but also that the proofs were *already* programs in their own right. This observation, now known as the Curry-Howard correspondence, sparkled an important paradigmatic shift.

It materialized in several occasions, by the independent discovery of the same object twice, once by a logician, and once by a computer scientist. One of the earlier example postdating the explicitation of the Curry-Howard isomorphism may be found in the parallel definition of Girard's System F [48] and Reynolds's Polymorphic λ-calculus [98]. The two objects are exactly the same, except that System F stems from second-order logic while the Polymorphic λ-calculus was constructed as a programming language.

This identity allowed for the design of objects that were at the same time a programming language and a proof system. Such languages allow to write programs and prove them in one go, ensuring programming developments that fully respect their specification, and thus, bug-free[2]. In addition, the efficiency of computers on tedious tasks allows to tackle problems that were unthinkable of half a century ago, and mathematics will probably be deeply transformed from such a systematical mechanization, enlarging the boundaries of tractability.

One of the first practical implementation of such a hybrid system was the Coq proof assistant [33] which is nowadays used to write critical software as well as proving complicated mathematical statements.

Another consequence of this equivalence, maybe more surprising, manifested itself by a chance remark due to Griffin [53]. Trying to type a programming primitive called `call/cc` from the Scheme programming language, he realized that this primitive actually granted the expressiveness of classical logic. Thus the Curry-Howard correspondence was not restricted to intuitionistic logic, and could be extended by providing extraneous primitives that retained a computational content. Following this discovery, Krivine started to try to implement well-known axioms in a computational way, spawning a fruitful research program, the so-called classical realizability.

[2] As long as the specification itself is correct, of course.

The understanding of mathematical axioms as new programming constructs opened the door to a brand new world. Logic had finally found the reality it was the model of: more than the science of language, logic is the science of *programming* languages.

We consider ourselves to be in the wake of this bicephal tradition, that merges ideas coming from computer science with principles originating in logic. This paradigmatic standpoint allows for an interdisciplinary look at objects from both worlds, and can often give many enlightening hindsights from the other side of the bridge. The central contribution of this thesis is actually an instance of such a methodological manifesto. It consists in looking at a well-known object, Gödel's Dialectica, with a Curry-Howard era computer scientist's look, based on the pioneering work by De Paiva and Hyland [92, 93, 58].

Their work provided a firm categorical framework to synthetize Dialectica-like translations. Strangely enough, the Dialectica translation by itself did not benefit from this categorical apparatus. In particular, a clear understanding of the computational effects at work in the translation remained to be found. What does the program corresponding to the translation of a proof actually do? We would like the present thesis to fill this gap.

If it were to be reduced to a dozen of words, we would like this thesis to be summarized by the following claim.

> "All the while, Gödel's Dialectica translation was a classical realizability program translation, manipulating stacks as first-class objects and treating substitution in a computationally relevant way, ultimately allowing to observe those stacks at variable access time."

Buried under a hefty crust of technicalities inherited from a time where the λ-calculus was but a toy and Howard was not even a teenager, the Dialectica translation actually turns out to be a jewel of the finest kind, exhibiting constructions that are elegantly explained in the classical realizability realm. It features in particular a first-class notion of stacks as concrete objects, while also dealing with such a subtle thing as delayed substitution, as found in the Krivine machine for example. These notions were virtually unknown at the time the Dialectica translation was published, let alone designed, effectively making the work of Gödel even more awing.

The main contributions of this thesis can be stated as follows.

1. A reformulation of the Dialectica translation as a pure, untyped, program translation that respects the underlying operational semantics of the λ-calculus. While the work of De Paiva and Hyland was an important step into this direction, they were not based on computational objects and relied both on an underlying categorical structure and an explicitly typed language. To the best of our knowledge, the issue of the lack of preservation of β-equivalence in the original Dialectica was never even clearly stated as such.

2. A computational description of the translation, in terms of new side-effects described in the Krivine machine. This presentation is heavily inspired by the work of Krivine in classical realizability, and draws many of the folklore ideas of this research topic. This is a novel approach, and it raises actually more questions than it solves, for we now have a new range of intuitions arising from this description.

3. A new formulation for on-demand computation (the *call-by-need* family of reductions) rooted in logical considerations rather than *ad-hoc* hypotheses. This problem is related to the representation of variable substitution in λ-calculus, and echoes with the clever encodings of substitutions in the Krivine machine by means of closures that play an important rôle in the Dialectica translation.

4. An extension of the Dialectica translation to the dependently-typed case, and a study of its limitations. Contrarily to double-negation translations, the Dialectica translation can be adapted easily to cope with type-dependency, as long as the preservation of β-equivalence has been solved. This is not the case though for dependent elimination, and this hints at new variants of the Dialectica.

5. Various relatives of the Dialectica translation appearing when considering distinct linear decompositions. Many of them are folklore, but presenting them in the same place in the style of Oliva [89] allows to highlight interesting common patterns, as well as possible new variants.

The chapters making this thesis up themselves are roughly divided according to the above classification. Let us give here a broad description of their respective contents.

Chapters 2, 3 and 4 are introductory and recall the necessary basis to understand the developments that follow, together with pervasively used objects and notations. More precisely,

- Chapter 2 describes the λ-calculus and related topics, as well as the minimum amount of logic that will be needed in the course of this thesis;

- Chapter 3 introduces linear logic, that we will use as a valuable tool throughout the later developments, and a bit of category theory to grasp the basic concepts of effects in the λ-calculus;

- Chapter 4 deals with the introduction of dependent types from a very high-level standpoint.

Chapter 5 is a short summary of the problems that occur whenever trying to add effects into a dependently-typed system. While it does not provide any real result by itself, it eases the understanding of the implications of a dependent Dialectica, which is given later. It could therefore be considered as an extended introductory chapter.

Chapter 6 explores a fresh presentation of call-by-need λ-calculi through the lens of logic, and in particular linear logic. We show how a new type of contexts, closure contexts, mimicking the analogous closures from the Krivine machine, allows to derive in

a straightforward fashion various call-by-need calculi. This presentation is more canonical and easily lifts to classical settings. This chapter is quite self-contained and mostly independent from the following ones.

The remaining chapters revolve around the Dialectica translation.

Chapter 7 recalls the Dialectica translation as presented by Gödel, although we do it in a slightly modernized way. Its main goal is to show that the translation soundly interprets Heyting's arithmetic, as well as a few additional semi-classical axioms. Although not yielding new results, it features a more proof-theoretical savvy that is often absent from the presentations of the historical Dialectica.

Chapters 8 and 9 are morally part of the same semantical unit. The former attempts to adapt the historical presentation of Dialectica to a Curry-Howard paradigm, but unluckily fails to fulfill all the expected properties of such a presentation. The latter analyzes the reasons for this failure, and fixes them by drawing ideas from De Paiva and Hyland. It then turns to the explanation of the content of the resulting translation by describing its operational behaviour in the Krivine machine. In light of these findings, much of the historical presentation is scrutinized with new programming intuitions in mind.

Chapter 10 discusses variants of the translation obtained through linear logic. It also provides new explanations for the realizers of the semi-classical axioms of Dialectia seen as programming primitives.

Chapter 11 describes how a dependently-typed Dialectica would look like. It explains to which extent the Dialectica translation is naturally dependent, and also where it cannot account for some principles. Typically, the Dialectia readily accommodates the negative fragment, but fails at encoding the dependent elimination rules.

Finally, Chapter 12 aims at giving a partial reconstruction of the Dialectica translation by putting in perspective with related translations, such as forcing. In the course of this chapter, we show that a variant of the Lafont-Reus-Streicher CPS can be naturally obtained by tweaking a call-by-name forcing translation according to the ideas developed by Miquel and Krivine. This observation paves the way for potential future work on the Dialectica.

2 Prolegomena

> Attention ! Ce flim n'est pas un flim sur le cyclimse.
>
> *Georges Abitbol about the flim.*

We will review in this introductory chapter the basic tools and concepts that we will be at the heart of the remaining of this document. It will be the occasion to recall the definition of the often copied but never equaled λ-calculus, as well as giving a rudimentary look at logic seen through the prism of proof-theory. The exposition of both will lead us to the formulation of the Curry-Howard isomorphism, an observation that proved itself to be an incredibly fruitful discovery both for computer science and logic and from which this thesis proceeds.

2.1 The λ-calculus

Our choice to start with the definition of the λ-calculus, the great ancestor of functional programming language, rather than with the basic principles of logical reasoning in a thesis greatly inspired by the works of Gödel may have sounded a bit odd to him, and this may be still be the case for some modern logicians. While we acknowledge this fact, we also advocate that computation is at the heart of logic, up to the point that we even claim that *logic is a corollary of computation*.

The fact that the λ-calculus was designed by a logician, Alonzo Church, is not here to invalidate this statement [29]. The fact that he designed it even before the formulation of Turing completeness does not validate it either. In any case, the λ-calculus was a logical system that turned out to be a full-fledged programming language as well [104]. This duality is deeply rooted in our research field since the dawn of modern proof-theory, and is a constitutive element of our cultural horizon.

2.1.1 The archetypal λ-calculus

In its rawest form, the λ-calculus can be defined by a three-case inductive structure and one reduction rule. This conciseness makes it an object rather simple to manipulate mathematically.

Definition 1 (λ-calculus)**.** The terms of λ-calculus t, u are defined by the following inductive grammar

$$t, u := x \mid t\ u \mid \lambda x.\, t$$

where x stands for a variable, and where the variable x is bound in $\lambda x.\, t$. This means that we implicitly quotient terms by the so-called α-equivalence, i.e.

$$\lambda x.\, t[x] \equiv \lambda y.\, t[y]$$

where x and y are fresh variables and $t[\cdot]$ stands for a term depending on a given variable.

The term $\lambda x.\, t$ is called λ-*abstraction* while $t\ u$ is an *application*. We will often group abstractions together under the same λ for readability, and use the _ placeholder to stand for variable bindings that do not appear in the body of the term they bind.

The λ-calculus is endowed with a rewriting system, called β-reduction, which is generated by the congruence closure of the following rule:

$$(\lambda x.\, t)\ u \to_\beta t[x := u]$$

where $t[x := u]$ stands for the capture-free substitution of x by u in t. The equivalence generated by this oriented reduction is called β-equivalence, and is written as $\equiv_\beta$.

A term is said in normal form whenever it does not reduce.

Designing precisely and correctly what we mean by capture-free substitution is not as easy as it sounds, and many modern programming languages still get it despairingly wrong [1]. There is a plentiful literature on this topic, but for our purpose, it is sufficient to consider that in the substitution

$$t[x := u]$$

one renames by α-equivalence all bound variables of t and u to fresh instances before actually performing the substitution. From now on we will simply forget about these issues and reason implicitly up to α-renaming.

There is a vast quantity of theorems about the λ-calculus. We will quickly state a few ones we believe to be fundamental.

Theorem 1. *The λ-calculus is Turing-complete, that is, it can compute any algorithm.*

Proof. The proof of its Turing-completeness is actually simultaneous to the very definition of Turing-completeness by Turing [104].

□

Theorem 2. *The β-reduction is confluent, i.e. if $t \to_\beta^* r_1$ and $t \to_\beta^* r_2$ then there exists r such that $r_1 \to_\beta^* r$ and $r_2 \to_\beta^* r$.*

Proof. A standard proof is Tait's one, found amongst others in Barendregt's book [17].

□

[1] The Python programming language is a famous case.

2.1.2 Reductions and strategies

When considering the λ-calculus as a programming language, it is quite usual to restrict the β-equivalence according to a given set of constraints. The resulting systems are often called *calculi*, together with a qualificative. There are two standard such restrictions, the so-called *call-by-name* and *call-by-value* λ-calculi.

Definition 2 (Call-by-name)**.** The call-by-name λ-calculus is simply defined by the unrestricted β-reduction.

Definition 3 (Call-by-value)**.** The call-by-value reduction relies on the notion of value. For the time being, a value v is simply a λ-abstraction, that is, $v := \lambda x.\, t$.

The call-by-value λ-calculus is then defined by the congruence closure of the call-by-value $\to_{\beta v}$ rule, given below.

$$(\lambda x.\, t)\ v \quad \to_{\beta v} \quad t[x := v]$$

The equivalence generated by these rules is written $\equiv_{\beta v}$.

The only difference between by-name and by-value reduction is that, in the latter, only values get substituted. This effectively forces the computation of the argument of a function before applying the β-rule. The two reductions are sharply distinct, as witnessed by the following term

$$(\lambda x.\, \lambda y.\, y)\ ((\lambda x.\, x\ x)\ (\lambda x.\, x\ x))$$

which is equivalent to $\lambda y.\, y$ in call-by-name, but not in call-by-value where it does not have any normal form.

The fact that we qualify the usual β-reduction as by-name may startle the reader used to actual programming. Indeed, what is the point of giving two distinct names to one object? The answer comes from the fact that the two become distinct when adding the constraint that the reduction system is a *strategy*.

Definition 4 (Strategy)**.** A strategy s is a deterministic reduction rule $\to_s$, that is, for any $t \to_s r_1$ and $t \to_s r_2$, we have $r_1 = r_2$.

A strategy can be considered as a particular implementation of a reduction: it orders the places where the reduction is to happen in a predictable fashion. The by-name and by-value calculi can be adapted into strategies. A simple way to present their strategy variants is to define them using reduction contexts.

Definition 5 (Context)**.** A context is a term E with a special free variable written $[\cdot]$ that appears once and exactly once in E. We will write $E[t]$ for E where the hole variable has been substituted by t.

Definition 6 (By-name and by-value strategies)**.** The call-by-name strategy is defined, using the inductive context E_n below:

$$E_n := [\cdot] \mid E_n\ t$$

by the following reduction rule.

$$E_n[(\lambda x.\, t)\ u] \to_n E_n[t[x := u]]$$

The left-to-right call-by-value strategy is defined, using the inductive context E_{vlr} below:

$$E_{vlr} := [\cdot] \mid E_{vlr}\ t \mid (\lambda x.\, t)\ E_{vlr}$$

by the following reduction rule.

$$E_{vlr}[(\lambda x.\, t)\ v] \to_{vlr} E_{vlr}[t[x := v]]$$

The right-to-left call-by-value strategy is defined, using the inductive the context E_{vrl} below:

$$E_{vrl} := [\cdot] \mid E_{vrl}\ v \mid t\ E_{vrl}$$

by the following reduction rule.

$$E_{vrl}[(\lambda x.\, t)\ v] \to_{vrl} E_{vrl}[t[x := v]]$$

Proposition 1. *The reductions given in the previous definition are indeed strategies.*

2.1.3 Typing

In the presentation given in Section 2.1.1, the λ-calculus is presented as a mere (higher-order) rewriting system. It is nonetheless usual to restrict the expressive power of this λ-calculus by subjecting it to a typing discipline. The resulting calculi are therefore called typed λ-calculus, while the original calculus is called untyped.

In the more abstract and general way, a type is a syntactic datum that restricts the uses one can make of a λ-term. There are various notions of types in the literature, but we will essentially focus here on the simply-typed variant.

Before going further, it is worth to mention that there are two main fashions to present the typing of λ-terms.

- An intrinsic style, where terms carry the typing information in their syntactical structure. This representation, called Church-style, has various benefits. The most notable one, in general, lies in the fact that it allows for the decidability of typing. Its main drawback is that it forces the term to live in a given type and makes coercion involved.
- An extrinsic style, where terms do not carry any typing information. Such a presentation, called Curry-style, exchanges the benefits and drawbacks of the Church-style, that is, it usually renders typing undecidable while allowing to freely separate computationally relevant terms from their typing derivations.

Most of the time, we will be inclined towards the Curry-style for its simplicity, although the results we present can be adapted to the Church-style.

We recall here one of the simplest typing systems for the λ-calculus, the so-called *simply-typed* system.

Definition 7 (Simple types)**.** The simple types are built over the following grammar:

$$A, B := \alpha \mid A \to B$$

where α ranges over a fixed set of base types.

The typing property is based on a notion of sequent, which associates a type to a term assuming that its free variables are also typed.

Definition 8 (Sequents)**.** An environment Γ is a list of pairs of variables and types, defined by the following inductive grammar.

$$\Gamma, \Delta := \cdot \mid \Gamma, x : A$$

We will be writing $\Gamma_1, \ldots, \Gamma_n$ for each type at position i in Γ.

A typing sequent is a triple of the form $\Gamma \vdash t : A$ where Γ is an environment, t a term and A a type. As for λ-terms, we will be considering that variables in environments are bound, and thus we will freely apply α-equivalence whenever needed.

Finally, typing is defined in terms of typing derivations, which are trees of sequents representing decoration of the underlying term. It is usual to define the typing relation by means of typing rules of the form

$$\frac{\Delta_1 \vdash u_1 : A_1 \qquad \ldots \qquad \Delta_n \vdash u_n : A_n}{\Gamma \vdash t : A}$$

where the sequent below the line is called *conclusion* of the rule, and the (possibly empty) list of sequents above the line are called *premises* of the rule. Several side-conditions can also be present, in general related to the management of free variables.

Definition 9 (Simply-typed λ-calculus)**.** The simply-typed λ-calculus is defined by the following typing rules.

$$\frac{\Gamma \vdash t : B}{\Gamma, x : A \vdash t : B} \qquad \frac{}{\Gamma, x : A \vdash x : A}$$

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.\, t : A \to B} \qquad \frac{\Gamma \vdash t : A \to B \qquad \Gamma \vdash u : A}{\Gamma \vdash t\ u : B}$$

We will often use implictly the following property.

Proposition 2. *The following derivation is derivable:*

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A}$$

where $x : A \in \Gamma$ has the expected meaning.

The main interest of typing systems is that, while being purely static objects, they are compatible with the dynamics of the underlying term, viz. the reduction rules, in the sense that they enjoy a *subject reduction* property.

Theorem 3 (Subject reduction)**.** *If $\Gamma \vdash t : A$ and $t \to_\beta r$ then $\Gamma \vdash r : A$.*

This allows to ensure a handful of properties on well-typed normal forms. For instance, any closed normal term of type $A \to B$ must be a λ-abstraction.

The simply-typed λ-calculus is not Turing-complete, and its expressiveness is very weak. For the sake of completeness, we present here Girard's System F, a.k.a. Reynold's polymorphic λ-calculus, that allows for quantification over types with a minimal amount of additional material. In Curry-style, it chiefly consists in one supplementary type constructor, no new term constructor and two additional typing rules.

Definition 10 (Polymorphic types)**.** The grammar of polymorphic types is given below.

$$A, B := \alpha \mid A \to B \mid \forall \alpha.\, A$$

Here, α is bound in $\forall \alpha.\, A$, so the usual α-equivalence tricks apply.

Definition 11 (System F)**.** System F is made of the simply-typed rules together with the two additional rules below.

$$\frac{\Gamma \vdash t : A \qquad \alpha \text{ fresh}}{\Gamma \vdash t : \forall \alpha.\, A} \qquad \frac{\Gamma \vdash t : \forall \alpha.\, A}{\Gamma \vdash t : A[\alpha := B]}$$

It is well-known, since Girard's thesis, that System F is strongly normalizing, and thus the simply-typed λ-calculus as well.

Theorem 4. *For any term t such that $\vdash t : A$ in System F, then t has a normal form.*

Proof. See for instance the Girafon [47]. □

2.1.4 Datatypes

The simply-typed λ-calculus is not very expressive, and for programming purposes, it is usual to add base datatypes, allowing to build up more complicated types. We will consider two families of such datatypes, sums and products, by adding their nullary and binary variants, namely:

- For the products, the singleton 1 and pair types $A \times B$;
- For the sums, the empty 0 and binary sum $A + B$ types.

They require us to extend both the types and the terms to manipulate them. The resulting calculus, that we will call the $\lambda^{\times +}$-calculus, will be used extensively.

Definition 12 ($\lambda^{\times+}$-calculus)**.** The $\lambda^{\times+}$-calculus terms are the ones from the usual λ-calculus, extended with the following additional structure:

$$\begin{aligned} t, u := \ldots \quad & \mid (t, u) \mid \texttt{match}\ t\ \texttt{with}\ (x, y) \mapsto u \\ & \mid \textbf{inl}\ t \mid \textbf{inr}\ t \mid \texttt{match}\ t\ \texttt{with}\ [x \mapsto u_1 \mid y \mapsto u_2] \\ & \mid () \mid \texttt{match}\ t\ \texttt{with}\ () \mapsto u \mid \texttt{match}\ t\ \texttt{with}\ [\cdot] \end{aligned}$$

Reduction rules are made of β-equivalence extended to the other connectives.

$$\texttt{match}\ ()\ \texttt{with}\ () \mapsto t \to_\beta t$$

$$\texttt{match}\ (t, u)\ \texttt{with}\ (x, y) \mapsto r \to_\beta r[x := t, y := u]$$

$$\texttt{match}\ \textbf{inl}\ t\ \texttt{with}\ [x \mapsto u \mid y \mapsto r] \to_\beta u[x := t]$$

$$\texttt{match}\ \textbf{inr}\ t\ \texttt{with}\ [x \mapsto u \mid y \mapsto r] \to_\beta r[y := t]$$

Types are taken from the simply-typed λ-calculus together with inductive datatypes.

$$A, B := \ldots \mid 1 \mid A \times B \mid 0 \mid A + B$$

Typing rules are the ones from the simply-typed λ-calculus, extended with the rules below.

$$\frac{}{\Gamma \vdash () : 1} \qquad \frac{\Gamma \vdash t : A \qquad \Gamma \vdash u : B}{\Gamma \vdash (t, u) : A \times B}$$

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash \textbf{inl}\ t : A + B} \qquad \frac{\Gamma \vdash t : B}{\Gamma \vdash \textbf{inr}\ t : A + B}$$

$$\frac{\Gamma \vdash t : 1 \qquad \Gamma \vdash u : C}{\Gamma \vdash \texttt{match}\ t\ \texttt{with}\ () \mapsto u : C} \qquad \frac{\Gamma \vdash t : 0}{\Gamma \vdash \texttt{match}\ t\ \texttt{with}\ [\cdot] : C}$$

$$\frac{\Gamma \vdash t : A \times B \qquad \Gamma, x : A, y : B \vdash u : C}{\Gamma \vdash \texttt{match}\ t\ \texttt{with}\ (x, y) \mapsto u : C}$$

$$\frac{\Gamma \vdash t : A + B \qquad \Gamma, x : A \vdash u_1 : C \qquad \Gamma, y : B \vdash u_2 : C}{\Gamma \vdash \texttt{match}\ t\ \texttt{with}\ [x \mapsto u_1 \mid y \mapsto u_2] : C}$$

The `match` terms are called pattern-matching terms, and the other ones are called constructors.

Remark 1. Note that the $\lambda^{\times+}$-calculus does not feature truly recursive datatypes, in particular natural numbers, even though it would not be difficult to add syntactically.

We would need to take some care if we wanted to remain strongly normalizing, but that would be about that. Nonetheless, we will tend to call any algebraic datatype an *inductive type*, be it recursive or not.

We define now some useful terms that will be used pervasively.

Definition 13. We define the following constants.

$$\begin{array}{lcl} \mathrm{fst} & := & \lambda p.\, \texttt{match}\ p\ \texttt{with}\ (x, y) \mapsto x \\ \mathrm{snd} & := & \lambda p.\, \texttt{match}\ p\ \texttt{with}\ (x, y) \mapsto y \end{array}$$

We will also be using the following useful macros defined below

$$\begin{array}{lcl} \lambda().\, t & := & \lambda p.\, \texttt{match}\ p\ \texttt{with}\ () \mapsto t \\ \lambda(x, y).\, t & := & \lambda p.\, \texttt{match}\ p\ \texttt{with}\ (x, y) \mapsto t \\ \lambda\,[\cdot] & := & \lambda p.\, \texttt{match}\ p\ \texttt{with}\ [\cdot] \\ \lambda[x \mapsto u_1 \mid y \mapsto u_2] & := & \lambda p.\, \texttt{match}\ p\ \texttt{with}\ [x \mapsto u_1 \mid y \mapsto u_2] \end{array}$$

where p is fresh.

For the sake of completeness, one should mention that System F is powerful enough to represent those datatypes through the famous impredicative encoding. The trick is to encode a term living in an inductive datatype as any pattern-matching one could apply to it, universally quantifying the return type of this pattern-matching. We give the type encoding below.

$$\begin{array}{lcl} 0 & := & \forall\alpha.\, \alpha \\ 1 & := & \forall\alpha.\, \alpha \to \alpha \\ A + B & := & \forall\alpha.\, (A \to \alpha) \to (B \to \alpha) \to \alpha \\ A \times B & := & \forall\alpha.\, (A \to B \to \alpha) \to \alpha \end{array}$$

The term encoding is straightforward. We will only expose the pair case here to give the flavour of this encoding.

$$\begin{array}{lcl} (t, u) & := & \lambda k.\, k\ t\ u \\ \texttt{match}\ t\ \texttt{with}\ (x, y) \mapsto u & := & t\ (\lambda x\, y.\, u) \end{array}$$

2.2 A minimalistic taste of logic

We detail here the basic requirements of the remaining of this document regarding logic. We will stick to a very syntactical approach here, and will mainly present the inference systems we need for our purposes. Before presenting the technical content, we would like to take a step back and look at our perception of logic from a computer scientist's point of view.

2.2.1 The programmer's bar talk

For centuries, logic has been thought of as the way to justify the validity, in terms of truth, of a given discourse. While the notion of inference rules was already present in Aristotle's *Organon*, it would not be until Frege's *Begriffsschrift* [42] that logic would have been provided with a truly formal reasoning system. In the wake of this tradition, we will tend to consider logic through the angle of syntactical objects, namely sequents derivations, rather than semantical structures such as models.

The derivation notation allows one to concisely express a succession of deduction steps that would be otherwise clumsy to write out in a more informal language. Compare for instance the two following presentations of the same object, a variant of the *modus ponens* rule.

All men are mortal.
Socrates is a man.
Thus Socrates is mortal.

$$\frac{\vdash A \to B \qquad \vdash B \to C}{\vdash A \to C}$$

As famously shown by Gödel [52], the syntactical approach suffers from an inherent limitation that deeply disturbed the mathematicians of that era. Indeed, he demonstrated that any such derivation system, with a computational proof-checking, and expressive enough to model arithmetic was either inconsistent, i.e. proved all formulae, or incomplete, i.e. featured some undecidable formula, which the system neither proved nor disproved. Even worse, the consistency of the system was precisely an instance of such undecidable formulae.

This issue arises from an intrinsic limitation of logic, but to be understood in its rawest, linguistic form. As soon as we get to manipulate infinite enough objects, we hit an expressiveness wall: our language, be it mathematical or not, is finitistic, in the sense that the set of all sentences that mankind will ever utter in some form or another is bound to be denumerable.

This may have sounded terrible to the logicians of the eve of the twentieth century, but we programmers are not afraid anymore of such a limitation. Our programming languages are Turing-complete, and for any logical system reasonable enough, by Gödel's incompleteness theorem, one can find a program whose termination cannot be proven by this system. Somehow, *programming won*.

This is why we claimed that logic was a corollary of computation. From our computer scientist look, logic is the delicate art of formulating systems rich enough to prove that the program we are looking at is indeed correct, but weak enough not to be inconsistent.

2.2.2 Propositional logic

Propositional logic is a very simple family of logic where formulae are restricted to the well-named propositional fragment, which is defined below.

Definition 14 (Propositional formulae)**.** Propositional formulae A, B are inductively defined as follows:

$$A, B := \alpha \mid \top \mid \bot \mid A \to B \mid A \wedge B \mid A \vee B$$

where α ranges over propositional variables.

The even smaller fragment only made of variables and arrow connective is called the minimal fragment, and it will turn out to be important in the next section.

Even before choosing a particular set of rules for our formulae, there are various ways to present the deduction steps of the logic. For a reason that will appear obvious later on, we will stick to the *natural deduction* presentation.

Definition 15 (Natural deduction)**.** An environment Γ is a list of formulae, defined by the following inductive grammar.

$$\Gamma, \Delta := \cdot \mid \Gamma, A$$

A natural deduction sequent is a pair of the form $\Gamma \vdash A$ where Γ is an environment and A a formula.

We now proceed to give the deduction rules. We will stick to the intuitionistic set of rules, resulting in the proof system known as **LJ**, the intuitionistic natural deduction.

Definition 16 (**LJ**)**.** The **LJ** system has the following deduction rules.

$$\frac{}{\Gamma, A \vdash A} \qquad \frac{\Gamma \vdash A}{\Gamma, B \vdash A} \qquad \frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B} \qquad \frac{\Gamma \vdash A \to B \qquad \Gamma \vdash A}{\Gamma \vdash B}$$

$$\frac{}{\Gamma \vdash \top} \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A} \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B} \qquad \frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \wedge B}$$

$$\frac{\Gamma \vdash \bot}{\Gamma \vdash A} \qquad \frac{\Gamma \vdash A}{\Gamma \vdash A \vee B} \qquad \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B}$$

$$\frac{\Gamma \vdash A \vee B \qquad \Gamma, A \vdash C \qquad \Gamma, B \vdash C}{\Gamma \vdash C}$$

Rules featuring a connective absent from the premises in the conclusion are called *introduction* rules for this connective (possibly with an additional qualificative to distinguish them). Dually, rules featuring a connective in a premise that disappears from the conclusion are called *elimination* rules. Two rules are not covered by this categorization, the so-called structural rules. Those are the two first rules, called respectively *axiom* and *weakening*.

Minimal logic is the fragment that only uses the structural rules together with the introduction and elimination rules for the arrow.

Notation 1. We will write $\neg A$ for $A \to \bot$.

Usually, mathematicians use classical logic, which is an extension of intuitionistic logic with an additional axiom featuring some form of classical reasoning. Amongst them, let us cite the excluded-middle and the double-negation elimination, whose schemes are formulated below.

$$\frac{}{\vdash A \vee \neg A} \qquad \frac{}{\vdash \neg\neg A \to A}$$

We call them *schemes* because these axioms are morally universally quantified over A, but as we lack such a way to express it in **LJ**, we rather choose to do so externally, by providing an axiom for each instance of the formula A.

Those two axioms are logically equivalent in **LJ**, and we will call **LK** the system **LJ** equipped with one of those two classical axioms.

In minimal logic, the lack of a notion of falsity prevents us from being able to write negation, so that the preferred way to introduce classical logic in this setting is by means of Peirce's law, which is the following axiom scheme.

$$\frac{}{\vdash ((A \to B) \to A) \to A}$$

This axiom is close to the double-negation axiom, where each instance of $\bot$ would have been replaced by A and B respectively.

2.2.3 First-order logic

First-order logic is an extension of propositional logic where one is allowed to reason about a given set of terms. Those terms are defined through a signature.

Definition 17 (Signature)**.** A signature is a set of pairs of symbols and nonnegative integers, where such an integer is called the arity of the corresponding symbol.

We will tend to qualify the signature according to the intended nature of the symbols it contains. In particular, we will consider in this section term signatures and predicate signatures.

Definition 18 (First-order terms)**.** Assuming a signature, first-order terms are defined by the following inductive grammar:

$$t, u := x \mid f\ t_1\ \ldots\ t_n$$

where in the first case, x ranges over variables, and in the second case, f is a symbol and n its arity from the signature.

Similarly to the term construction, formulae of first-order logic are built as an extension of propositional formulae upon another signature, the predicate signature.

Definition 19 (First-order formulae)**.** Assuming a term signature and a predicate signature, the formulae of first-order logic are those from propositional logic extended with the following structure:

$$A, B := \ldots \mid \forall x.\, A \mid \exists x.\, A \mid P\ t_1\ \ldots\ t_n$$

where x is bound in $\forall x.\, A$ and $\exists x.\, A$, n is the arity of P in the predicate signature, and the $t_1\ \ldots\ t_n$ are terms built from the given term signature.

The base logic is an extension of **LJ** to the first-order setting.

Definition 20 (First-order logic)**.** The inference rules of first-order logic are simply the ones from **LJ**, enriched with the additional rules below to handle the first-order quantifications.

$$\frac{\Gamma \vdash A \qquad x \text{ fresh in } \Gamma}{\Gamma \vdash \forall x.\, A} \qquad \frac{\Gamma \vdash \forall x.\, A}{\Gamma \vdash A[x := t]}$$

$$\frac{\Gamma \vdash A[x := t]}{\Gamma \vdash \exists x.\, A} \qquad \frac{\Gamma \vdash \exists x.\, A \qquad \Gamma, A \vdash C \qquad x \text{ fresh in } \Gamma, C}{\Gamma \vdash C}$$

This specification allows one to describe a vast range of different logical systems by choosing a particular pair of term and predicate signatures, and by considering a particular set of axioms over the resulting formulae. This provides a great flexibility and extensibility. We will nonetheless only focus on the Heyting arithmetic, which is one of the most standard theory of natural integers.

The system we presented is, as **LJ**, an intuitionistic system. One can recover a classical system by adding one of the axioms mentioned before.

2.3 The Curry-Howard isomorphism

The Curry-Howard isomorphism is a revolutionary discovery that was made independently first by Curry and then about ten years later by Howard [57]. Stemming from a seemingly innocuous observation, it paved the way for a deep paradigm shift in logic, effectively bringing together two worlds that did not seem to have so close bounds at first sight: theoretical computer science and the foundations of mathematics.

2.3.1 A significant insignificant observation

The observation from which sprouted all this is somehow obvious, once we know where to look at. Rather than convoluted explanations, we prefer to let the reader glance at the objects below, which are already spelt out derivation rules of minimal logic and typing rules of λ-calculus, put in the same place.

$$\frac{}{\Gamma, A \vdash A} \qquad \frac{\Gamma \vdash A}{\Gamma, B \vdash A} \qquad \frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B} \qquad \frac{\Gamma \vdash A \to B \qquad \Gamma \vdash A}{\Gamma \vdash B}$$

$$\frac{}{\Gamma, x : A \vdash x : A} \qquad \frac{\Gamma \vdash t : A}{\Gamma, x : B \vdash t : A} \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.\, t : A \to B} \qquad \frac{\Gamma \vdash t : A \to B \qquad \Gamma \vdash u : A}{\Gamma \vdash t\ u : B}$$

It is now time to point at the elephant in the room.

Theorem 5. *Minimal logic derivations are in one-one correspondence with simply-typed λ-calculus derivations (up to α-equivalence).*

This historical observation is the Rosetta Stone of modern proof theory. Each property from one side can be put in regard with another one from the other side.

Proposition 3. *The β-reduction from the simply-typed λ-calculus is known under the name of* cut elimination *on the logical side.*

Famous soundness theorems that are a corollary of cut-elimination are therefore no more than variants on the strong normalization property of typed λ-calculi.

Minimal intuitionistic logic is not really expressive. Luckily, the Curry-Howard isomorphism can be lifted to more general systems. For the sake of comprehensiveness, we list a few alias names below, even though we will not present all of the systems.

Calculus	Logical system
Simply-typed λ-calculus	Minimal logic
$\lambda^{\times +}$-calculus	Propositional logic
$\lambda\Pi$-calculus	First-order logic
System F	Second-order logic

The above system equivalences are mostly based on the following equivalence between programming structures and logical features.

Computation	Logic
0	$\bot$
1	$\top$
$A + B$	$A \vee B$
$A \times B$	$A \wedge B$
$A \to B$	$A \to B$
reduction	cut-elimination
normal form	cut-free proof

There would be already a lot to say about all of these equivalences, but we would lack of time. We rather carry on towards the consequences of such a discovery.

2.3.2 From proofs to programs

Many interesting properties coming from logic can be revisited through this correspondence. We will focus on the seminal case of translations allowing to recover classical logic, for they played an important rôle in the development of the comprehension of the Curry-Howard isomorphism.

Intuitionistic logic does not prove the classical principles exposed before, namely the double-negation elimination or the excluded middle. Yet, since Glinvenko [49] and Gödel [50], it is a well known fact that one can transform **LK** proofs into **LJ** proofs through a family of translations known as *double negation translations* to recover classical provability. Let us give an instance of such a translation in minimal logic.

Definition 21 (Double-negation translation)**.** Let R be a formula from minimal logic. We define the translation $(-)^N$ from minimal logic into itself as follows.

$$\begin{array}{lcl} \alpha^N & := & \alpha \\ (A \to B)^N & := & A^N \to (B^N \to R) \to R \end{array}$$

Sequents are then translated as follows.

$$(\Gamma \vdash A)^N := \Gamma^N \vdash (A^N \to R) \to R$$

The following theorems are close to the historical formulation given by Gödel, and would stay as-is until the Curry-Howard revolution.

Theorem 6 (Soundness)**.** *If $\Gamma \vdash A$ is derivable in minimal logic, then $(\Gamma \vdash A)^N$ is also derivable in minimal logic.*

Theorem 7 (Classical logic)**.** *The sequent $(\vdash ((A \to B) \to A) \to A)^N$ is derivable in minimal logic.*

Theorem 8 (Correction)**.** *Assuming we trivially lift the translation to* **LJ** *and take $R := \bot$, then the two sequents*

$$\Gamma \vdash A \quad \textit{and} \quad (\Gamma \vdash A)^N$$

are equiprovable in **LK**.

With our Rosetta Stone at hand, we can give a complete rereading of these theorems in terms of program translations, and shed a new light on their actual content. To this end, it is sufficient to look at the details of the soundness proof to recover a λ-term out of it.

Definition 22 (CPS translation)**.** Given a λ-term t, we define the term t^N by induction as follows.

$$\begin{array}{lcl} x^N & := & \lambda k.\, k\ x \\ (\lambda x.\, t)^N & := & \lambda k.\, k\ (\lambda x.\, t^N) \\ (t\ u)^N & := & \lambda k.\, t^N\ (\lambda f.\, u^N\ (\lambda x.\, f\ x\ k)) \end{array}$$

The soundness theorem is no more that a typing preservation of our λ-term.

Proposition 4. *Assuming* $\Gamma \vdash t : A$*, then* $\Gamma^N \vdash t^N : (A^N \to R) \to R$*.*

Proof. By induction on the typing derivation. □

The proof of Peirce's law can also be understood as a λ-term.

Proposition 5. *The term*

$$\lambda k.\, k\ (\lambda f\, k.\, f\ (\lambda x\ \_\, .\, k\ x)\ k)$$

has the type of Peirce's law through the sequent translation.

Proof. By unfolding of the definition. □

The intriguing part of this result is that the corresponding term translation is rather well-known from the programming side. Indeed, it is the *continuation-passing style* which is used pervasively in some settings, such as event-based programming as found in JavaScript for instance. The term that allows to prove Peirce's law is in particular used to provide backtrack, giving hindsights into what a classical proof is.

The explicitation of the underlying program has another consequence. What about the relative semantics of translated programs? This question is hard to formulate when the proof terms are hidden by the typing derivation, and may not even make any sense if the translation relies on types. Actually, the $(-)^N$ translation is also well-behaved w.r.t. to the equivalence of programs.

Proposition 6. *For any term* t *and any value* v*, the following equivalence holds*

$$(t[x := v])^N \equiv_\beta t^N[x := v^V]$$

where v^V *is defined by case analysis on* v *below.*

$$\begin{array}{lcl} x^V & := & x \\ (\lambda x.\, t)^V & := & \lambda x.\, t^N \end{array}$$

Proof. By induction on t. □

Theorem 9. *For any* t *and* r*, if* $t \equiv_{\beta v} r$ *then* $t^N \equiv_\beta r^N$*.*

Proof. Direct consequence of the above lemma. □

This is much richer than the mere preservation of typing. The above translation has thus both a logical and a computational content, a fact which was difficult to highlight when working with derivations only. Moreover, this theorem holds even if the terms are untyped!

2.3.3 From programs to proofs

If the Curry-Howard isomorphism were to be used only to justify *a posteriori* that a logical translation can be understood computationally, it would not be that useful. The most interesting direction is the reverse one, when applying ideas from programming languages in a logical setting.

It is probable that one of the works that spawned most long-term derivatives is the now famous article of Griffin [53] on the typing of the `call/cc` primitive from the Scheme programming language. This is maybe one of the most striking examples were the proof-as-program equivalence was taken the other way around, and it inspired many subsequent lines of work.

The `call/cc` operator, there after written `cc` for brevity, originates in the Scheme programming language, which is itself a close descendant of the untyped λ-calculus. Literally *call with current continuation*, its computational behaviour can be summarized as follows.

Definition 23 (`cc` reduction)**.** The `cc` operator has the following operational semantics

$$E_n[\texttt{cc}\ t] \to t\ (\lambda x.\, \mathcal{A}\ E_n[x])$$

where E_n stands for the call-by-name contexts from Section 2.1.2, and where the $\mathcal{A}$ operator itself has the following reduction.

$$E_n[\mathcal{A}\ t] \to t$$

Scheme being a untyped language, the typing of `cc` remained unstudied until Griffin decided to tackle the issue. The constraints were to give it a type so that the reduction rules of Definition 23 would preserve subject reduction in call-by-name. It turned out that the result was much more far-reaching as it initially seemed.

Proposition 7. *The following typing rules are sound w.r.t. the operational semantics of Definition 23.*

$$\frac{}{\Gamma \vdash \texttt{cc} : ((A \to B) \to A) \to A} \qquad \frac{}{\Gamma \vdash \mathcal{A} : A \to B}$$

This was kind of unexpected. The otherwise innocent-looking `cc` operator was therefore a program implementing Peirce's law in the λ-calculus, and thus the first step into giving a computational content to classical logic.

This discovery had another deep consequence. It was the paradigmatic shock that shifted our point of view on proof theory. Indeed, instead of trying to prove additional axioms through logic (or program) translations, one could try to get them in a *direct-style* fashion. This means that instead of relying on a heavy modification of the source terms, one could simply add an operator to the source language, together with the right operational semantics so as to *realize* the desired formula. This led to the following new entry in our Rosetta Stone.

Computation	Logic
side-effects	new reasoning principles

There are indeed many effects in the wild that the λ-calculus simply does not acknowledge, amongst which one can cite the following.

- Global state.
- Exceptions.
- Threads.
- Non-determinism.

This revamped Curry-Howard isomorphism allows us to look at those effects from a fresh logical point of view. The ongoing work of Krivine [70] in classical realizability, for instance, focuses on trying to give a computational content to the axiom of choice in a classical setting, and heavily relies on such a correspondence [81].

We place ourselves in this trend of work, and an important part of this thesis is dedicated to the understanding of the Dialectica translation as a side-effect.

2.4 Abstract machines

Abstract machines are a family of structures dedicated to the computation of the normal form of a λ-term according to some strategy. There are many kind of machines, each one implementing its own calling convention. Let us cite, amongst others, the SECD machine [38] and the ZINC [75], which are both call-by-value machines. As we will focus chiefly on the call-by-name strategy, we present here the Krivine abstract machine, which is probably the most well-known implementation of a call-by-name machine.

2.4.1 The Krivine machine

The Krivine machine, as its name hints at, due to Krivine, is a call-by-name abstract machine for the λ-calculus [68]. For brevity, we will name it by its acronym, the KAM.

The machine is described by reduction rules acting on processes, which are made of a pair of a closure and a stack. Stacks play the role of contexts in strategies, as the relationship between those two objects can be made formal. Closures are terms with delayed substitutions. They are made up of a usual λ-term, together with an environment that allows to retrieve the contents of the free variables of that term on-demand.

Definition 24 (Krivine machine processes)**.** We define below in a mutually inductive fashion the various components that constitute the machine: processes p, environments σ, closures c, stacks π and usual λ-terms t.

$$\begin{array}{rcl} p & := & \langle c \mid \pi \rangle \\ c & := & (t, \sigma) \\ \sigma & := & \cdot \mid \sigma + (x := c) \\ \pi & := & \varepsilon \mid c \cdot \pi \end{array}$$

Definition 25 (Krivine machine rules)**.** The reduction rules of the machine are written below.

$$\begin{array}{rcll} \langle (x, \sigma + (x := c)) \mid \pi \rangle & \longrightarrow & \langle c \mid \pi \rangle & (\textsc{Grab}) \\ \langle (x, \sigma + (y := c)) \mid \pi \rangle & \longrightarrow & \langle (x, \sigma) \mid \pi \rangle & (\textsc{Garbage}) \\ \langle (t\ u, \sigma) \mid \pi \rangle & \longrightarrow & \langle (t, \sigma) \mid (u, \sigma) \cdot \pi \rangle & (\textsc{Push}) \\ \langle (\lambda x.\, t, \sigma) \mid c \cdot \pi \rangle & \longrightarrow & \langle (t, \sigma + (x := c)) \mid \pi \rangle & (\textsc{Pop}) \end{array}$$

Proposition 8. *The Krivine machine implements a call-by-name reduction.*

Proof. See Krivine's article on the topic [68]. □

The fact that the KAM is call-by-name can be observed in the structure of its stacks. Indeed, if we forget about the closures, the stacks and the call-by-name contexts E_n follow the same inductive structure.

$$\begin{array}{rcccl} E_n & := & [\cdot] & \mid & E_n\ t \\ \pi & := & \varepsilon & \mid & t \cdot \pi \end{array}$$

The rule Push is transparent from the point of view of the strategy, because its only task is to find the next redex in the context by destructuring it applicationwise. This explicitation is precisely what distinguishes strategies from abstract machines.

The Krivine machine is nonetheless slightly more fine-grained that the usual call-by-name reduction, because it uses closures to delay substitutions of variables. This fact will be discussed more lengthily at Chapter 6.

2.4.2 Krivine realizability

While we are studying the KAM, we cannot help presenting at the same time Krivine realizability, also known as classical realizability. It is a model construction technique stemming from a clever use of the KAM, and is at the heart of Krivine's program aiming at realizing axioms such as the full axiom of choice [70, 71], as well as a handy tool to explain the computational content of Cohen's forcing translation [30, 81] that earned the latter the Fields medal. Even though it will not be used as such in the remaining, most of the idea we will develop are ultimately inspired by this technique, so that we believe it deserves to be at least written out once.

Krivine realizability is, as witnessed by its name, a realizability, that is, a member of the family of techniques that interpret logical formulae as a set of computational objects, called realizers. In this precise case, those programs are going to be written in the λ-calculus extended with some additional primitives. The main specificity of Krivine realizability over other types of realizability is the fact it defines not only terms, but also co-terms, which are going to be materialized by stacks in the KAM. The realization property of terms is then defined in an intertwined way, making appearing stacks

explicitly in there. One of the key points of this definition is that it is made in terms of an *orthogonality relation*, which is the set-theoretic equivalent of the double-negation construction.

Definition 26 (Orthogonality)**.** Assume two sets A and B, and R a subset of $A \times B$. For any $\alpha \subseteq A$, we define $\alpha^R \subseteq B$ the orthogonal of α as

$$\alpha^R := \{b \in B \mid \forall a \in \alpha.\, (a, b) \in R\}$$

and symmetrically for any $\beta \subseteq B$ we define β^R.

As we explained, this construction behaves in a way similar to the double-negation construction.

Proposition 9. *Assume A, B and R fixed as in the definition. Then for any $\alpha, \alpha' \subseteq A$ we have the following.*

- $\alpha \subseteq \alpha^{RR}$.
- $\alpha^{RRR} \subseteq \alpha^R$.
- *If $\alpha \subseteq \alpha'$, then $\alpha'^R \subseteq \alpha^R$.*

This construction is used a lot in models that feature some form of classical reasoning, which is the case for Krivine realizability. We can also mention constructions based on double-glueing [60].

For now, we turn to the proper definition of classical realizability. There are several variants, though. We will concentrate on the interpretation of second-order logic, for the balance between its technical simplicity and its expressive power. For us, a formula of second-order logic will be no more than a type of system F, which we already defined above.

Definition 27 (Krivine realizability)**.** Assume a set of KAM processes $\perp\!\!\!\perp$ closed by anti-reduction, i.e. for all processes p and q, if $p \to q$ and $q \in \perp\!\!\!\perp$ then $p \in \perp\!\!\!\perp$. We call such sets saturated.

For any System F type A and any assignation of variables ρ into sets of stacks, we mutually define $\|A\|_\rho$ the falsity value of A and $|A|_\rho$ the truth value of A as follows.

$$\begin{array}{lcl} \|\alpha\|_\rho & := & \rho(\alpha) \\ \|A \to B\|_\rho & := & |A|_\rho \cdot \|B\|_\rho \\ \|\forall \alpha.\, A\|_\rho & := & \bigcup_{a \subseteq \Pi} \|A\|_{\rho,(\alpha:=a)} \\ |A|_\rho & := & (\|A\|_\rho)^{\perp\!\!\!\perp} \end{array}$$

Here, Π stands for the set of stacks, and the $\cdot$ notations stands for the pointwise application of the corresponding stack constructor. Note that we abused the orthogonality notation a bit, by identifying a process with a set-theoretic pair of a closure and a stack. The $\perp\!\!\!\perp$ set is often known as a *pole* in the literature.

We say that a closure c realizes A, written $c \Vdash A$, whenever $c \in |A|_\rho$ for any assignation ρ covering the free variables of A. If in addition c realizes A for any choice of saturated $\bot\!\!\!\bot$, we say that it universally realizes A.

The main theorem is the following one. It allows to freely consider typed terms as universal realizers.

Theorem 10 (Soundness)**.** *Assume a particular choice of saturated set of processes $\bot\!\!\!\bot$. For any term t such that $\Gamma \vdash t : A$ in System F, then for any choice of closures $c_i \Vdash \Gamma_i$, we have $(t, (\vec{x}_i := \vec{c}_i)) \Vdash A$.*

Proof. The proof goes by induction on the typing derivation. We will not give the details here, but all the cases have the following form: assume a stack in the falsity value of the resulting type, apply at most one step of reduction, use the saturated property of $\bot\!\!\!\bot$ if needed and conclude by the induction hypothesis. This works because the truth value is defined by orthogonality.

The important remark to do in this proof is that one does not rely on a particular set of reduction rules. The proof goes through if there is *at least* the rules described before, but adding any supplementary reduction does not interfere with the proof.

□

Because of this openness property, one can enrich the λ-calculus with new operators to prove more than just second-order intuitionistic logic. Typically, one can enrich it with a variant of the aforementioned `call/cc` operator fitted to the KAM to recover classical logic.

Definition 28 (Call with current continuation)**.** We extend our λ-calculus with the following constructions.

$$t, u := \ldots \mid \mathtt{cc} \mid k_\pi$$

We also add the following new reductions to the KAM.

$$\begin{array}{lcl} \langle (\mathtt{cc}, \sigma) \mid c \cdot \pi \rangle & \to & \langle c \mid (k_\pi, \sigma) \cdot \pi \rangle \\ \langle (k_\pi, \sigma) \mid c \cdot \rho \rangle & \to & \langle c \mid \pi \rangle \end{array}$$

Proposition 10. *The closure $(\mathtt{cc}, \cdot)$ universally realizes Peirce's law.*

Proof. By a mere unfolding of the interpretation of Peirce's law and repeated application of the saturation property of the pole.

□

We will not continue in this direction, for it would lead us too far from the core of this thesis. Nonetheless, Krivine realizability is a fascinating object and is a renewed source of surprises. For instance, Scherer and Dagand showed that the soundness theorem itself, seen through the Curry-Howard isomorphism, was no more than an interpreter [100] written in a continuation-passing style which turns out to be the Lafont-Reus-Streicher decomposition [72]. Likewise, Girard's original proof of normalization of System F [48] by reducibility candidates can be seen as a variant of the soundness theorem [86].

3 Linear Logic

> Dis-toi qu'il est tellement plus mieux d'éradiquer les tentacules de la déréliction.
>
> *Tranxen 200 about structural rules.*

Linear logic (henceforth abbreviated as **LL**) is a logic introduced by Girard [44], from the study of coherent spaces. It is essentially a refinement of intuitionistic logic where the intuitionistic arrow $A \Rightarrow B$ is decomposed into two more atomic components: a linear arrow $\multimap$ and an exponential modality ! according to the call-by-name decomposition of the arrow:

$$A \Rightarrow B := !A \multimap B \tag{3.1}$$

Several other such decompositions exist, allowing for the encoding of various calling conventions into linear logic. We will be giving some of them later on.

This short chapter is a quick summary of the guiding principles of linear logic, allowing us to define the necessary vocabulary and conventions of this domain.

3.1 Syntax

3.1.1 Formulae

Let us start the definition of linear logic by describing the formulae that make up the language of formulae.

Definition 29 (Linear formulae)**.** The formulae of **LL** are defined by the inductive grammar given below.

$$A, B := X \mid X^{\perp} \mid \bot \mid \top \mid 1 \mid 0 \mid A \parr B \mid A \,\&\, B \mid A \otimes B \mid A \oplus B \mid !A \mid ?A$$

From a purely logical point of view, linear connectives can be seen as coming from the duplication of the connectives from usual classical propositional logic into two families, the so-called *multiplicative* and *additive* connectives. The two remaining unary connectives $!(-)$ and $?(-)$ are named exponentials.

Classical	Multiplicative	Additive
true	1	$\top$
false	$\bot$	0
and	$\otimes$	$\&$
or	$\parr$	$\oplus$

One defines a notion of duality on the formulae, in the following way.

Definition 30 (Linear duality)**.** Let A be a **LL**-formula, we define the **LL**-formula $A^\bot$ by induction on A.

$$X^\bot := X^\bot \qquad (X^\bot)^\bot := X$$
$$\bot^\bot := 1 \qquad 1^\bot := \bot$$
$$\top^\bot := 0 \qquad 0^\bot := \top$$
$$(A \parr B)^\bot := A^\bot \otimes B^\bot \qquad (A \otimes B)^\bot := A^\bot \parr B^\bot$$
$$(A \,\&\, B)^\bot := A^\bot \oplus B^\bot \qquad (A \oplus B)^\bot := A^\bot \,\&\, B^\bot$$
$$(!A)^\bot := ?A^\bot \qquad (?A)^\bot := !A^\bot$$

The linear duality is similar to the De Morgan's duality in classical logic, and it shares with it the fact that it is an involution.

Proposition 11. *For any* **LL***-formula* A*,* $(A^\bot)^\bot = A$*.*

Proof. By induction on A. □

The multiplicative (resp. additive, exponential) fragment is stable by duality, so it is usual to consider fragments made up of a given set of formula families, named according to that set (M for multiplicative, A for additive and E for exponential).

Notation 2. The linear arrow is defined as $A \multimap B := A^\bot \parr B$.

3.1.2 Proofs

Although several attempts at a term language for linear logic were made, all of them resulted partly unsatisfactory. Amongst the various families of term languages, one can mention the following ones.

- The historical syntax of proof-nets [64]. Proof-nets feature a convenient syntax with the right amount of quotients, but they are only really elegant in the MLL fragment without units, and mostly satisfactory in the MELL one without units. They tend to become fairly intricate when extended to larger fragments.
- Syntaxes based on the dual-intuitionistic presentation of linear logic (DILL [16]). While these syntaxes are usable seamlessly in a modified λ-calculus, they do not feature the fundamental duality at work in **LL**, so we do not really consider them as a syntax for **LL**.

- Modern syntaxes based on dualities of sequent calculus [35]. While these syntaxes stem from classical sequent calculus, they are a promising way to handle linearity as well. For now they only treat well polarized logic [74] and **LC** [43] rather than the whole linear fragment, so they are not an option yet.

The fact we lack a good term syntax for **LL** explains why we usually present it as a sequent calculus. We will stick to this standard approach here, for the sake of simplicity.

Definition 31 (Sequents)**.** Linear contexts are defined as lists of formulae.

$$\Gamma, \Delta := \cdot \mid \Gamma, A$$

Sequents are then simply given by a linear context, of the form $\vdash \Gamma$. For the sake of readability, we will sometimes use bilateral sequents made of two contexts, of the form $\Gamma \vdash \Delta$. This is just eye-candy for the sequent $\vdash \Gamma^\perp, \Delta$ where $(-)^\perp$ is interpreted pointwise. We will often write $\Gamma_1, \dots, \Gamma_n$ to explicit the types that make up the sequent Γ.

Definition 32 (Inference rules)**.** The rules of linear logic are given below.

$$\frac{}{\vdash A, A^\perp} \qquad \frac{\vdash \sigma(\Gamma) \qquad \sigma \text{ permutation}}{\vdash \Gamma} \qquad \frac{\vdash \Gamma, A \qquad \vdash \Delta, A^\perp}{\vdash \Gamma, \Delta}$$

$$\frac{\vdash \Gamma}{\vdash \Gamma, \bot} \qquad \frac{\vdash \Gamma, A, B}{\vdash \Gamma, A \parr B} \qquad \frac{}{\vdash 1} \qquad \frac{\vdash \Gamma, A \qquad \vdash \Delta, B}{\vdash \Gamma, \Delta, A \otimes B}$$

$$\frac{}{\vdash \Gamma, \top} \qquad \frac{\vdash \Gamma, A \qquad \vdash \Gamma, B}{\vdash \Gamma, A \,\&\, B} \qquad \frac{\vdash \Gamma, A}{\vdash \Gamma, A \oplus B} \qquad \frac{\vdash \Gamma, B}{\vdash \Gamma, A \oplus B}$$

$$\frac{\vdash \Gamma, ?A, ?A}{\vdash \Gamma, ?A} \qquad \frac{\vdash \Gamma}{\vdash \Gamma, ?A} \qquad \frac{\vdash \Gamma, A}{\vdash \Gamma, ?A} \qquad \frac{\vdash ?\Gamma, A}{\vdash ?\Gamma, !A}$$

As one can witness, the linear logic restricts the use of the structural rules (weakening and contraction) to the formulae of the form $?A$ for some A. This is actually the source of the *linear* qualificative: hypotheses are used linearly, and one needs to resort to exponentials to retrieve the usual structural rules. This also explains why linear logic is often branded as a resource-sensitive logic.

Theorem 11 (Cut-elimination)**.** *Linear logic enjoys cut-elimination.*

Proof. See for instance [44]. A semantic proof that we find particularly elegant is based on the use of phase spaces, a structure which can be roughly understood as a collapse of classical realizability, where all proof-terms have been identified [88].

Given a commutative monoid $\mathcal{M}$ and a pole $\perp\!\!\!\perp \subseteq \mathcal{M}$, we interpret a formula A as a set of elements $[\![A]\!]$ of the monoid closed by the orthogonality generated by the pole. Indeed, for any set $X \subseteq \mathcal{M}$ we can define

$$X^{\perp\!\!\!\perp} := \{y \in \mathcal{M} \mid \forall x \in X.\ xy \in \perp\!\!\!\perp\}$$

and say that a set X is closed whenever $X = X^{\perp\!\!\!\perp\perp\!\!\!\perp}$. The formulae are then interpreted inductively as given below.

$$\begin{array}{lcllcl}
[\![\top]\!] & := & \mathcal{M} & [\![0]\!] & := & \emptyset^{\perp\!\!\!\perp\perp\!\!\!\perp} \\
[\![A \,\&\, B]\!] & := & [\![A]\!] \cap [\![B]\!] & [\![A \oplus B]\!] & := & ([\![A]\!] \cup [\![B]\!])^{\perp\!\!\!\perp\perp\!\!\!\perp} \\
[\![\bot]\!] & := & \perp\!\!\!\perp & [\![1]\!] & := & \{1\}^{\perp\!\!\!\perp\perp\!\!\!\perp} \\
[\![A \parr B]\!] & := & ([\![A]\!]^{\perp\!\!\!\perp} \cdot [\![B]\!]^{\perp\!\!\!\perp})^{\perp\!\!\!\perp} & [\![A \otimes B]\!] & := & ([\![A]\!] \cdot [\![B]\!])^{\perp\!\!\!\perp\perp\!\!\!\perp} \\
[\![?A]\!] & := & ([\![A]\!]^{\perp\!\!\!\perp} \cap \{1\}^{\perp\!\!\!\perp\perp\!\!\!\perp} \cap \mathcal{I})^{\perp\!\!\!\perp} & [\![!A]\!] & := & ([\![A]\!] \cap \{1\}^{\perp\!\!\!\perp\perp\!\!\!\perp} \cap \mathcal{I})^{\perp\!\!\!\perp\perp\!\!\!\perp}
\end{array}$$

Here, $\mathcal{I}$ stands for the submonoid of idempotents elements of $\mathcal{M}$ and $X \cdot Y$ for the pointwise monoidal product of elements of X and Y. The interpretation is adapted to sequents by considering the comma to represent a $\parr$ connective.

It is then easy to show by induction the following soundness theorem: for any choice of $\mathcal{M}$ and $\perp\!\!\!\perp$, if there is a proof of the sequent $\vdash \Gamma$, then $1 \in [\![\Gamma]\!]$.

The existence of cut-free sequents is retrieved by applying this soundness theorem to the syntactic monoid $\mathcal{S}$ whose elements are sequents Γ quotiented by the equivalence relation $\cong$ generated by the two following rules.

- For any permutation σ, $\Gamma \cong \sigma(\Gamma)$.
- For any formula A, $?A \cong ?A \parr ?A$.

The monoidal product is then simply concatenation, and the unit is the empty sequent.

By taking $\perp\!\!\!\perp$ to be the set of sequents which admit a cut-free proof, the soundness theorem gives a straightforward way to recover a cut-free proof from any starting proof. □

Corollary 1. *Linear logic is consistent, i.e. there is no proof of the empty sequent.*

3.2 Polarization

Rather than the syntactic requirement of linearity, what strikes us in the very nature of **LL** is that it exhibits polarization. There are distinct ways to understand or even define polarization: more than a precise notion, it is a family of properties enjoyed by the connectives of linear logic.

First, we can discriminate two classes of connectives, that are related by duality.

Definition 33 (Polarized connectives)**.** We say that the connectives $\bot, \top, \parr$ and $\&$ are negative, and by duality, that the connectives $1, 0, \otimes$ and $\oplus$ are positive.

There is a simple syntactic criterion that distinguishes negative connectives.

Definition 34 (Invertibility)**.** A n-ary connective F is invertible if for all sequent $\vdash \Gamma, F(A_1, \dots, A_n)$, there is a proof of this sequent iff there is a proof of this sequent that starts by introducing F.

Proposition 12. *Negative connectives are invertible.*

This is a very coarse characterization, and we prefer to think of polarity as a more proof-theorical feature related to a computational behaviour. To describe this formally, we would first need to explain the linear decompositions of intuitionistic logic into **LL**, be it call-by-name or call-by-value. Yet, we give here a global picture of what we mean by polarization.

- Positive connectives are described by their values. This is not very obvious for now because we presented **LL** as a sequent calculus, but we can already perceive it somehow. All normal-form proofs of a sequent of the form $\vdash P$ where P starts with a positive connective must begin with an introduction rule for P describing its shape. In programming languages, one can observe in a similar way how positives can be observed through pattern-matching, thus effectively describing the shape of the considered term. Algebraic datatypes are the archetypal positive objects.
- Negative connectives are defined by the way they react to values. They have no definite shape and cannot be observed *per se*. In the usual programming languages, this is why functions, the negative objects by excellence, are opaque objects whose only rôle is to be applied. A nice way to think of them may be to consider them as thunks hiding some existentially-quantified closures, making them unobservable.

This gloss can be made formal, see for instance Zeilberger [107] or Munch [85]. Actually, such a duality could have already been highlighted in the proof of cut-elimination. The interpretation of positive connectives is always of the form $(A \odot B)^{\perp\!\!\perp\perp\!\!\perp}$ (for binary connectives) where $\odot$ embodies the way we build values of that type (by union for the sum, and by product for the tensor) while negative are conversely defined by orthogonality over those values. The cut-elimination theorem is a way to normalize a proof to recover the values it was hiding.

Polarization thus provides good insights into the operational behaviour of programming language, and in turn, decompositions into linear logic give a finer description of programming constructions. Rather than linearity, we will therefore be more interested in the resulting polarization.

3.3 A bit of category theory

Category theory is a widely used framework to expose the semantics of a first-order language. We will therefore briefly use it in this section to describe the semantics of effects in general and linear logic in particular.

Definition 35 (Categories)**.** A category $\mathbf{C}$ is given as a set[1] of objects $\mathrm{Obj}_{\mathbf{C}}$, and for each pair of objects A, B a set of morphisms $\mathbf{C}(A, B)$ equipped with the following structure:

- for each object A, a morphism $\mathrm{id}_A : \mathbf{C}(A, A)$
- for any three objects A, B, C, any morphism $f : \mathbf{C}(A, B)$ and $g : \mathbf{C}(B, C)$, a morphism $f \,;\, g : \mathbf{C}(A, C)$

subject to the following equalities.

- For all objects A and B, and any morphism $f : \mathbf{C}(A, B)$, $\mathrm{id}_A \,;\, f = f$.
- For all objects A and B, and any morphism $f : \mathbf{C}(A, B)$, $f \,;\, \mathrm{id}_B = f$.
- For all objects A, B, C and D, and all morphisms $f : \mathbf{C}(A, B)$, $g : \mathbf{C}(B, C)$ and $h : \mathbf{C}(C, D)$, $(f \,;\, g) \,;\, h = f \,;\, (g \,;\, h)$.

It is usual to write equality of morphisms in category theory as commutative diagrams, which we will be partially doing to explicit the types of the considered objects.

The almost immediate thing one wishes to do with categories is to put some structure on top of it. It turns out that categories themselves form a category, where morphisms between two categories are precisely the functors between them. Functors are actually the natural[2] morphisms over categories, and are defined below.

Definition 36 (Functors)**.** Let $\mathbf{C}$ and $\mathbf{D}$ be two categories. A functor F from C to D, written $F : \mathbf{C} \to \mathbf{D}$ is given by two components.

- For all object A in $\mathrm{Obj}_{\mathbf{C}}$, an object FA in $\mathrm{Obj}_{\mathbf{D}}$.
- For all objects $A, B \in \mathrm{Obj}_{\mathbf{C}}$ and any morphism $f : \mathbf{C}(A, B)$, a morphism $Ff : \mathbf{D}(FA, FB)$

This must be compatible with the underlying categorical structure, that is:

- For all object $A \in \mathrm{Obj}_{\mathbf{C}}$, $F\,\mathrm{id}_A = \mathrm{id}_{FA}$.
- For all objects $A, B, C \in \mathrm{Obj}_{\mathbf{C}}$, and all morphisms $f : \mathbf{C}(A, B)$ and $g : \mathbf{C}(B, C)$, $F(f \,;\, g) = Ff \,;\, Fg$.

Functors whose source and target categories coincide are called *endofunctors*.

Example 1. For any category $\mathbf{C}$, the identity functor $1_{\mathbf{C}}$ is trivially defined as follows.

$$\begin{array}{rcl} 1_{\mathbf{C}}\, A & := & A \\ 1_{\mathbf{C}}\, f & := & f \end{array}$$

We can push the morphization forward, by defining the proper notion morphisms between functors, which are named *natural transformations*.

[1] We will not be discussing fundation problems in this definition, so think of any loosely-defined set theory as the ambient metatheory.

[2] Still no categorical meaning intended.

Definition 37 (Natural transformations)**.** Let F and G be two functors from $\mathbf{C}$ to $\mathbf{D}$. A natural transformation α from F to G is given by an $\mathrm{Obj}_{\mathbf{C}}$-indexed family of $\mathbf{D}$-morphisms

$$\alpha_A : FA \longrightarrow GA$$

such that, for all $f : \mathbf{C}(A, B)$,

$$\begin{array}{ccc} FA & \xrightarrow{Ff} & FB \\ {\scriptstyle \alpha_A}\downarrow & & \downarrow{\scriptstyle \alpha_B} \\ GA & \xrightarrow{Gf} & GB \end{array}$$

We will now recall here the notion of monad. Altough monads originated in category theory, since the seminal work of Moggi [83], they are used pervasively in purely functional programming languages (most notably Haskell) to encode impure side-effects.

Definition 38. Let $\mathbf{C}$ be a category. A monad over $\mathbf{C}$ is a triple (T, η, μ) where $T : \mathbf{C} \to \mathbf{C}$ is an endofunctor on $\mathbf{C}$, and $\eta : 1 \to T$ and $\mu : T^2 \to T$ are natural transformations, subject to the following laws:

$$\begin{array}{ccc} TA & \xrightarrow{\eta_{TA}} & T^2A \\ {\scriptstyle T\eta_A}\downarrow & \searrow & \downarrow{\scriptstyle \mu_A} \\ T^2A & \xrightarrow{\mu_A} & TA \end{array} \qquad \begin{array}{ccc} T^3A & \xrightarrow{\mu_{TA}} & T^2A \\ {\scriptstyle T\mu_A}\downarrow & & \downarrow{\scriptstyle \mu_A} \\ T^2A & \xrightarrow{\mu_A} & TA \end{array}$$

The right intuition about the nature of a monad is to think of TA as a computation yielding some A. The monadic type contains additional data describing how the computation is evaluated. The η morphism allows to inject pure computations in the monad (with no additional data) while the μ morphism collapses this additional data in one chunk, thus performing both effects at once.

Example 2. Here are some standard instances of the monad structure.

- The identity monad $TA := A$.
- The double negation monad $TA := (A \to \bot) \to \bot$.
- The list monad $TA := 1 + A \times TA$.

Definition 39. A monad (T, η, μ) in a cartesian category $\mathbf{C}$ is strong whenever there exists a natural morphism

$$\sigma : \mathbf{C}(A \times TB, T(A \times B))$$

called a monadic strength, that respects the underlying identity and associativity induced by the cartesian product.

In practice, we will be working in cartesian closed categories where the monad we will be working with is internalizable. In those cases, the monad is automatically strong.

Definition 40. A strong monad (T, η, μ, σ) is commutative when the two canonical morphisms

$$TA \times TB \rightrightarrows T(A \times B)$$

coincide.

From the programming point of view, a commutative monad is a monad where the actual order of effects does not matter, only their presence (or absence) do.

Example 3. There are quite a few commutative monads in the wild. We give below the main representatives of this tribe.

- Given an object R, the reader monad $R \to -$ is a commutative monad. The order of reading the value does not matter indeed.
- Given an object W equipped with a commutative monoidal structure, the writer monad $- \times W$ is a commutative monad. Because the object written to does not keep track of the order of operations, this monad is obviously commutative.
- The option monad $1 + -$ is a commutative monad. The one thing that matters is the fact that the computation is defined or not.
- The multiset monad $\mathfrak{M}(-)$ is a commutative monad. This is a generalized version of the option monad where only the number of occurrences matters (and not anymore the mere presence or absence).

We will not dwell on the models of linear logic, for it would take us too far from the point we wanted to insist on. A comprehensive description can be found in the survey of Melliès [80] It suffices to state that one of the standard model is the linear-non-linear (LNL) adjunction, due to Benton [19]. To explain it shortly, this class of models is made up of a pair of categories, one which handles the linear part of the calculus, while the other one embeds the necessary structure for the exponentials. They are furthermore related by an adjunction allowing to go from one to the other.

The important remark we wanted to highlight is the following one. Linear logic is the somehow the syntax for writing commutative effects in direct-style, as stated by the theorem below.

Proposition 13. *Any LNL model of multiplicative linear logic gives rise to a commutative monad. Conversely, any commutative monad in a category with enough structure (essentially equalizers) can be decomposed in a linear-non-linear adjunction. Note that this decomposition does not need to be unique.*

Proof. See [19] for instance.

□

Combined with the polarization properties we mentioned before, this is the reason that makes us think that the relevant core of linear logic is not so much linearity, but rather the fact it is the perfect direct-style polarized commutative effect-handling language.

The linear side of the model, featuring a comonad, can be thought of as the direct-style part, where effects are marked in the type with the exponential modalities. As it is seen through the looking-glass of the adjunction, effects are actually implicit in this world.

3.4 Intuitionistic and classical decompositions

Linear logic is famous for its ability to be seen as a refinement of both intuitionistic and classical logic, through decompositions. In this section, we recall such well-known decompositions into linear logic. As we will be using them afterwards, it is indeed useful to gather them here.

3.4.1 The call-by-name decomposition

This decomposition is the historical one, and was actually at the source of the creation of linear logic. The main idea at work is that intuitionistic types are translated as negatives, and we need to sandwich enough bangs at each polarity shift.

Definition 41 (Call-by-name translation)**.** The call-by-name translation $[\![-]\!]_{\mathrm{n}}$ from intuitionistic logic to linear logic is inductively defined on types as follows.

- $[\![0]\!]_{\mathrm{n}} := 0$
- $[\![1]\!]_{\mathrm{n}} := 1$
- $[\![A + B]\!]_{\mathrm{n}} := ![\![A]\!]_{\mathrm{n}} \oplus ![\![B]\!]_{\mathrm{n}}$
- $[\![A \times B]\!]_{\mathrm{n}} := ![\![A]\!]_{\mathrm{n}} \otimes ![\![B]\!]_{\mathrm{n}}$
- $[\![A \to B]\!]_{\mathrm{n}} := ![\![A]\!]_{\mathrm{n}} \multimap [\![B]\!]_{\mathrm{n}}$

Sequents are then translated as given below.

$$[\![\Gamma_1, \dots, \Gamma_n \vdash A]\!]_{\mathrm{n}} := ![\![\Gamma_1]\!]_{\mathrm{n}}, \dots, ![\![\Gamma_n]\!]_{\mathrm{n}} \vdash [\![A]\!]_{\mathrm{n}}$$

Proposition 14. *If $\Gamma \vdash A$ is derivable in* **LJ***, then so is $[\![\Gamma \vdash A]\!]_{\mathrm{n}}$ in* **LL***.*

Proof. Rather than translating the sequents upfront, we prefer to only give the precise uses of exponential rules for each rule.

- Axiom: weakening and dereliction.
- Arrow introduction: none.
- Arrow elimination: contraction and promotion.

- Positive introductions: contraction and weakening (according to the rule) and promotion.
- Positive eliminations: contraction and weakening (according to the rule).

□

We can consider that only promotion and dereliction are the rules performing effects, if we think of them as the monadic operations in the dual intuitionistic world. This allows us to give a precise intuition of this translation: axiom rules performs the effect hidden in a variable by forcing its content (we are in call-by-name, after all) while arrow elimination and positive introduction box their arguments into an exponential. It is noteworthy that arrow introduction is neutral (this explains the validity of η-expansion in call-by-name).

3.4.2 The call-by-value decomposition

This decomposition is a little more involved than the previous one, because we need to somehow define two translations at once, even though the second one is simply derived from the second.

Definition 42 (Call-by-value translation)**.** The call-by-value translation $[\![-]\!]_{\mathrm{v}}$ is inductively defined on types as follows.

- $[\![0]\!]_{\mathrm{v}} := 0$
- $[\![1]\!]_{\mathrm{v}} := 1$
- $[\![A + B]\!]_{\mathrm{v}} := [\![A]\!]_{\mathrm{v}} \oplus [\![B]\!]_{\mathrm{v}}$
- $[\![A \times B]\!]_{\mathrm{v}} := [\![A]\!]_{\mathrm{v}} \otimes [\![B]\!]_{\mathrm{v}}$
- $[\![A \to B]\!]_{\mathrm{v}} := !([\![A]\!]_{\mathrm{v}} \multimap [\![B]\!]_{\mathrm{v}})$

Sequents are then translated as

$$[\![\Gamma_1, \ldots, \Gamma_n \vdash A]\!]_{\mathrm{v}} := [\![\Gamma_1]\!]_{\mathrm{v}}, \ldots, [\![\Gamma_n]\!]_{\mathrm{v}} \vdash [\![A]\!]_{\mathrm{v}}$$

In call-by-value, there is a sharp difference between *values* and *computations*. This is somehow reflected in the translation. The ! connective boxes the computations hidden in an arrow, rendering it inert, thus, a value.

Remark 2. This is not the unique presentation of this decomposition. Indeed, we could also interpret sequents as usually, that is, seeing $\Gamma \vdash A$ as $\Gamma_1 \to \ldots \to \Gamma_n \to A$.

Definition 43 (Alternative decomposition)**.** The following alternative decomposition can be used. The $[\![-]\!]_{\mathrm{v}}$ decomposition on types is defined as before, but now, sequents are translated as

$$[\![\Gamma \vdash A]\!]_{\mathrm{v}} := \vdash !(\Gamma_1 \multimap !(\ldots \multimap !(\Gamma_n \multimap A)))$$

The right way to think about it is to think of the ! connective as a *thunking* primitive: it encapsulates terms of negative type, potentially effectful, into a pure positive type, somehow turning them into values.

We can discriminate a special class of types, hereafter named hereditarily positive type, on this particular behaviour.

Definition 44 (Hereditarily positive types)**.** Hereditarily positive types P^+ are the types generated by the following inductive grammar.

$$P^+, Q^+ := 0 \mid 1 \mid P^+ \oplus Q^+ \mid P^+ \otimes Q^+ \mid !A$$

In call-by-value, appending a bang modality to hereditarily positive types (in particular, all value types in our translation) is useless, because we know that they are pure. In particular, the soundness of the encoding relies on the following lemma.

Proposition 15 (Monadic run)**.** *For all hereditarily positive type P^+, the following sequent is derivable.*

$$\vdash P^+ \multimap !P^+$$

Proof. By induction on P^+ and case analysis on the considered value. □

From the computational point of view, the above term allows one to purge an inhabitant of a hereditarily positive type from its effects, by recursively forcing its subcomponents. This lemma is the linear equivalent of Krivine's storage operators [69].

Proposition 16. *If $\Gamma \vdash A$ is derivable in* **LJ***, then so is $[\![\Gamma \vdash A]\!]_{\mathrm{v}}$ in* **LL***.*

Proof. As in the call-by-name case, we list the required exponential rules at each intuitionistic rule. When we write run + X, we mean that we apply the monadic run, the X rule and then dereliction on all run types. The global result of this operation is that we can transparently use exponential rules in the types *as if* there were bang modalities on the whole context.

- Axiom: run + weakening.
- Arrow introduction: run + promotion.
- Arrow elimination: run + contraction, and dereliction.
- Positive introductions: run + contraction and run + weakening (according to the rule).
- Positive eliminations: run + contraction and run + weakening (according to the rule).

□

Once again, as for the call-by-name translation, this interpretation gives intuitions about the effects occurring in the source calculus. We should underline the fact that the run $+\ X$ operation by itself can be seen as a mere typing artefact, because in the categorical models, run followed by dereliction amounts to identity. Let us comment a bit on these rules. The one rule performing effects is the arrow elimination, which is what the programming intuition gives us. Contrarily to call-by-name, arrow introduction is not free: it adds a packing layer to its argument through promotion. And once again, this is reflected by the fact that in call-by-value, η-expansion is *not* valid, because it transforms a computation into a (functional) value.

3.4.3 Classical-by-name

This is actually a variant of call-by-name rather than a proper translation. This is also the first translation that features a $?(-)$ modality without duality. Classical-by-name logic is essentially call-by-name where the computational power of terms have been extended to continuation-handling. Indeed, the equational theory is similar to the one call-by-name. From the linear point of view, this corresponds to adding $?(-)$ connectives under each $!(-)$ connective, as well as at outside the return type of arrows.

Definition 45 (Classical-by-name translation)**.** The classical call-by-name translation $[\![-]\!]_{\mathcal{C}\mathrm{n}}$ is inductively defined on types below.

- $[\![0]\!]_{\mathcal{C}\mathrm{n}} := 0$
- $[\![1]\!]_{\mathcal{C}\mathrm{n}} := 1$
- $[\![A + B]\!]_{\mathcal{C}\mathrm{n}} := !?[\![A]\!]_{\mathcal{C}\mathrm{n}} \oplus !?[\![B]\!]_{\mathcal{C}\mathrm{n}}$
- $[\![A \times B]\!]_{\mathcal{C}\mathrm{n}} := !?[\![A]\!]_{\mathcal{C}\mathrm{n}} \otimes !?[\![B]\!]_{\mathcal{C}\mathrm{n}}$
- $[\![A \to B]\!]_{\mathcal{C}\mathrm{n}} := !?[\![A]\!]_{\mathcal{C}\mathrm{n}} \multimap ?[\![B]\!]_{\mathcal{C}\mathrm{n}}$

Sequents are then translated as

$$[\![\Gamma_1, \ldots, \Gamma_n \vdash A]\!]_{\mathcal{C}\mathrm{n}} := !?[\![\Gamma_1]\!]_{\mathcal{C}\mathrm{n}}, \ldots, !?[\![\Gamma_n]\!]_{\mathcal{C}\mathrm{n}} \vdash ?[\![A]\!]_{\mathcal{C}\mathrm{n}}$$

Proposition 17. *If $\Gamma \vdash A$ is derivable in* **LK***, then so is $[\![\Gamma \vdash A]\!]_{\mathcal{C}\mathrm{n}}$ in* **LL***.*

Proof. Very similar to the one of call-by-name, save for a lot of dull administrative exponential steps to accommodate the pervasive presence of $?(-)$.

Let us rather look at a classical principle such as double-negation elimination to taste the particular flavour of this translation. We have

$$[\![\neg\neg A \to A]\!]_{\mathcal{C}\mathrm{n}} \equiv !?(!?(!?[\![A]\!]_{\mathcal{C}\mathrm{n}} \multimap ?0) \multimap ?0) \multimap ?[\![A]\!]_{\mathcal{C}\mathrm{n}}$$

We should put forward the fact that ?0 is linearly equivalent to $\bot$, so that the expression $A \multimap ?0$ is actually linearly equivalent to $A^{\bot}$. Applying this simplification twice, we only have to show that

$$\vdash ?!?!!?[\![A]\!]_{\mathcal{C}_{\mathrm{n}}} \multimap ?[\![A]\!]_{\mathcal{C}_{\mathrm{n}}}$$

which is in turn easily proved.

□

4 Dependent type theory

> Parce ce que je vais vous dire, vous êtes
> un type dans mon genre.
>
> *Louis Jouvet about stratification.*

In this chapter, we give a high-level introduction to the dependent type theory, the benefits it provides, as well as the specificities that raise interesting issues absent from the non-dependent systems. We would like to insist in particular on the dependent elimination in presence of inductive types, which are in our opinion an essential expressive gap that dependency alone does not really bring.

4.1 Term in types: a bird's-eye notion of dependency

In a nutshell, dependency can be described as the property that types can contain terms. This is a serious departure from the simple types, and even from second-order types, because this blurs the boundaries between terms and types. Indeed, usual terms do compute, so dependent types lift this computation inside types, resulting in the requirement of term reduction while type checking.

A significant fragment of potentially dependently-typed systems can be represented by the formalism of so-called pure type systems [18] (often abridged PTS). The main construction allowing terms to flow into types is the Π-type $\Pi x : A.\, B$, which is a generalization of the usual arrow. The bound variable x can indeed appear in B. This construction is naturally introduced by the λ-abstraction, thus generalizing the simply-typed arrow, in a rule of the shape

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x : A.\, t : \Pi x : A.\, B}$$

The fact that x may appear in B in the type $\Pi x : A.\, B$ has important consequences that one may not think about at first sight. First, that imposes a linear structure on the hypotheses that was not present in absence of dependency. Indeed, while $x : A, y : B[x]$ is a well-formed context, where $B[x]$ insists on the fact that B contains x, one cannot reorder it as they would do if B was not depending on x, as $y : B[x], x : A$ does not make any sense. Nonetheless, some reorderings are possible as long as they do not break the variable dependencies. This actually gives a rich structure to contexts. The well-formedness of contexts must be ensured by its own statement in the typing rules, usually written $\vdash_{\mathrm{wf}} \Gamma$.

This has logical consequences on the expressiveness as well. While the two types

$$A \to B \to C \quad B \to A \to C$$

are isomorphic in a pure calculus, and thus essentially the same, the presence of dependency makes the two following types

$$\Pi x : A.\, \Pi y : B[x].\, C[x, y] \quad \Pi y : B.\, \Pi x : A[y].\, C[x, y]$$

incomparable. Similarly, many reorderings do not even make sense. This gives additional constraints, potentially unsatisfiable, when trying to lift program transformations in a dependent case.

Note that the non-dependent arrow can nevertheless be retrieved from the dependent one, by simply taking $A \to B := \Pi x : A.\, B$ where x does not appear in B.

Second, eliminating an arrow results in the introduction of a term in a type, as the standard dependent rule for application is of the form

$$\frac{\Gamma \vdash t : \Pi x : A.\, B \qquad \Gamma \vdash u : A}{\Gamma \vdash t\ u : B[x := u]}$$

As mentioned before, this has the consequence that we need to handle reduction of terms in types to perform type-checking. Such a requirement is embodied in the inference rules as a rule usually called *conversion*, of the form

$$\frac{\Gamma \vdash t : B \qquad A \equiv_\beta B}{\Gamma \vdash t : A}$$

While one could stop here, and end up with a calculus only handling dependency *per se* (usually called $\lambda\Pi$ [18]), it is natural to go all the way up to PTS and the Barendregt cube, by also adding second and higher-order types, thus making effectively collapsing the dyke between term and types. To this end, we need to represent the type of types as a term. Usually, it is denoted by $\Box$. We then type types with this particular type.

We conclude this introductory section by giving an archetypal dependently-typed system $\mathbf{CC}_0$, to settle the ideas.

Definition 46 ($\mathbf{CC}_0$)**.** The terms of $\mathbf{CC}_0$ t, u, A, B are inductively defined by the grammar below.

$$t, u, A, B := x \mid t\ u \mid \lambda x : A.\, t \mid \Pi x : A.\, B \mid \Box$$

Context are defined as a list of named formulae.

$$\Gamma := \cdot \mid \Gamma, x : A$$

Reduction rules are taken from the usual λ-calculus and extended with the expected congruence closure rules.

$$\frac{}{\vdash_{\text{wf}} \cdot} \qquad \frac{\Gamma \vdash A : \square}{\vdash_{\text{wf}} \Gamma, x : A} \qquad \frac{\Gamma \vdash t : B \qquad \Gamma \vdash A : \square}{\Gamma, x : A \vdash t : B}$$

$$\frac{\Gamma \vdash A : \square}{\Gamma, x : A \vdash x : A} \qquad \frac{\vdash_{\text{wf}} \Gamma}{\Gamma \vdash \square : \square} \qquad \frac{\Gamma \vdash A : \square \qquad \Gamma, x : A \vdash B : \square}{\Gamma \vdash \Pi x : A.\, B : \square}$$

$$\frac{\Gamma, x : A \vdash t : B \qquad \Gamma \vdash \Pi x : A.\, B : \square}{\Gamma \vdash \lambda x.\, t : \Pi x : A.\, B} \qquad \frac{\Gamma \vdash t : \Pi x : A.\, B \qquad \Gamma \vdash u : A}{\Gamma \vdash t\ u : B[x := u]}$$

$$\frac{\Gamma \vdash t : B \qquad \Gamma \vdash A : \square \qquad A \equiv_\beta B}{\Gamma \vdash t : A}$$

Note that we added some side-conditions w.r.t. the naive rules described in the body of the text. They are mainly there to ensure well-formation of types.

As for the expressiveness, $\mathbf{CC}_0$ is a superset of such a rich system as F^ω, thus allowing to write quite a lot of functions.

4.2 The issue of universes

All would be well and good in our system, except for one slight defect.

Theorem 12. $\mathbf{CC}_0$ *is inconsistent, i.e. there exists a closed term* t *such that*

$$\vdash t : \Pi A : \square.\, A$$

Proof. See for instance [32]. □

The issue stems from the rule typing types, because $\vdash \square : \square$ is inconsistent in general. It allows to encode type-theoretic variants of the Burali-Forti or Russel paradoxes. A rough set-theoretic equivalent of this typing rule would state that the set of all sets is a set, which is known to lead to trouble.

Usually, this is worked around by imposing a hierarchy discipline on the universes. For instance, the historical Calculus of Constructions [33] (**CC**) distinguishes between a universe of types $*$ and the type of $*$, $\square$. The faulty typing rule is turned into $\vdash * : \square$ and the type-forming rules are adapted accordingly. In particular, $\square$ has no type.

Quite often, for expressiveness purposes, universes are assigned an ordinal (ranging on the natural integers in practical implementations) i.e. type $\square_\alpha$ is now indexed by an ordinal α, and the aforementioned typing rule is bound to respect the induced ordinal order, along the lines of $\vdash \square_\alpha : \square_\beta$ only whenever $\alpha < \beta$. We will be using such an integer-based hierarchy in chapter 11.

As we are not claiming to give a comprehensive description of the phenomena at work in dependent type theory, but rather a quick overview, we will not be describing anymore the complex consequences of the various design choices described above, and simply carry

on. We will also forget about universe issues most of the time, because we are not taking profit of complex impredicative encodings to do what we are describing, and we will thus hide this unneeded complexity under the carpet in high-level descriptions.

4.3 Dependent elimination

Dependent elimination is the adaptation of the usual elimination of positive types in a dependent setting, and it is, according to us, the real *killer feature* of dependent types. We will therefore give a quick overview of dependent elimination in this section.

We recall that in the simply-typed λ-calculus, inductive types are manipulated through introduction and elimination rules which take the following form

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash \mathrm{inl}\ t : A + B} \qquad \frac{\Gamma \vdash t : B}{\Gamma \vdash \mathrm{inr}\ t : A + B}$$

$$\frac{\Gamma \vdash t : A + B \qquad \Gamma, x : A \vdash u_1 : C \qquad \Gamma, y : B \vdash u_2 : C}{\Gamma \vdash \mathtt{match}\ t\ \mathtt{with}\ [x \mapsto u_1 \mid y \mapsto u_2] : C}$$

for the sum type, for instance. As usual, introduction rules are materialized as constructors application and elimination rules by pattern-matching.

Dependent elimination is based on the observation that head normal forms of terms inhabiting inductive types can be merely described by their head constructor. For instance, any term $t : A + B$ is convertible either to $\mathrm{inl}\ u_1$ or to $\mathrm{inl}\ u_2$ for some $u_1 : A$ and $u_2 : B$. Note that this is only true in purely intuitionistic settings; in presence of side-effects such as continuations or non-termination, it is well-known that the above observation is false. As our systems are pure, it is therefore legit to add this principle built-in, in the logic. A simple way to do this is to add an induction principle whose type would be

$$\Pi P : A + B \to \square.\, (\Pi x : A.\, P\ (\mathrm{inl}\ x)) \to (\Pi y : B.\, P\ (\mathrm{inr}\ y)) \to \Pi p : A + B.\, P\ p$$

together with the expected reduction rules.

There is actually a much more elegant way to deal with it, though. As in the simply-typed case, the (dependent) induction principle can be seen as an instance of a (dependent) pattern-matching. To achieve this, one simply need to make the C return type of the previous rule depend on the term being pattern-matched, resulting in the following rule.

$$\frac{\Gamma \vdash t : A + B \qquad \Gamma, x : A \vdash u_1 : C[z := \mathrm{inl}\ x] \qquad \Gamma, y : B \vdash u_2 : C[z := \mathrm{inr}\ y]}{\Gamma \vdash \mathtt{match}\ t\ \mathtt{with}\ [x \mapsto u_1 \mid y \mapsto u_2] : C[z := t]}$$

where z is fresh and x, y are not free in C. Indeed, we statically *know* in each branch the shape of the term t, so that if it was present in C, we can refine its value in each of the branches.

5 Effects and dependency

Ça dépend, ça dépasse.

Katia about dependent effects.

In this short chapter, we give a quick presentation of the issues that arise when considering effectful programming in a dependently-typed programming language, in particular in presence of dependent elimination. Conversely, we also sketch some nice properties recovered thanks to the additional expressiveness of the considered systems.

We suppose from now on that we have fixed an expressive enough dependent type theory that fits our needs. We will not dwell on details though, because we do not aim at applicability, but rather at a high-level description. We will thus forget about any hierarchy of universes, and just write $\square$ for any universe, actually embracing the typical ambiguity. Luckily, the constructions given in the remaining of this chapter can be expressed in almost any sufficiently enough expressive system, such as the Coq proof assistant for instance. Likewise, we will omit some easily inferable arguments and mark them with the _ wildcard.

5.1 Dependent Monads: a naive generalization

We fist have a look at the notion of monads in a dependent setting. As they represent the usual way to encode effects in otherwise pure languages, it is natural to see how well they perform with a little bit of dependency.

The main issue that appears when mixing a monad (T, η, μ) with dependent types is precise interaction between the type constructor $T(-)$ and the universal quantification $\Pi x : A.\, B$. To further explain this interaction, let us state more formally what we mean by a monad in a type-theoretical rather than categorical realm.

Definition 47 (Type-theoretical monad)**.** A monad is given by the four following closed terms:

- $\mathrm{T} : \square \to \square$
- $\mathrm{map} : \Pi A : \square.\, \Pi B : \square.\, (A \to B) \to \mathrm{T}\ A \to \mathrm{T}\ B$
- $\mathrm{return} : \Pi A : \square.\, A \to \mathrm{T}\ A$
- $\mathrm{join} : \Pi A : \square.\, \mathrm{T}\ (\mathrm{T}\ A) \to \mathrm{T}\ A$

subject to the equivalences of Definitions 37 and 38.

The first two terms correspond to the functor structure of T, while the two latter ones are exactly the η and μ natural transformations. Note that we do not want to insist on the actual equivalences those terms need to comply with. Indeed, we willingly leave totally undisclosed their nature, be it definitional (i.e. mere β-equivalence) or propositional equalities, or even more complex relations described in the theory. If that were the case, we would fall in an awing pitfall of technicalities that would lead us far from the high-level description we want to stick to here.

Even though it does not matter that much, we will pretend that they are implemented as definitional equalities, because the monads considered here will be totally opaque objects.

Remark 3. In programming languages that feature monads, it is usual to consider another presentation where the map and join operators have been merged into another one, the bind operator of the following type.

$$\mathrm{bind} : \Pi A : \square.\, \Pi B : \square.\, \mathrm{T}\ A \to (A \to \mathrm{T}\ B) \to \mathrm{T}\ B$$

The two presentations are equivalent as soon as we have higher-order functions, because we can interdefine them.

$$\begin{aligned}
\mathrm{bind} &:= \lambda(A\,B : \square)\,(x : \mathrm{T}\ A)\,(f : A \to \mathrm{T}\ B).\, \mathrm{join}\ A\ (\mathrm{map}\ A\ (\mathrm{T}\ B)\ f\ x) \\
\mathrm{map} &:= \lambda(A\,B : \square)\,(f : A \to B)\,(x : \mathrm{T}\ A).\, \mathrm{bind}\ A\ B\ x\ (\lambda x.\, \mathrm{return}\ (f\ x)) \\
\mathrm{join} &:= \lambda(A : \square)\,(x : \mathrm{T}\ (\mathrm{T}\ A)).\, \mathrm{bind}\ A\ A\ x\ \mathrm{return}
\end{aligned}$$

We will not use this combinator in this section, but we recall it for the sake of comprehensiveness.

Assume some monad given by the quadruple described above. Now, if one wants to make the dependent arrow interact with the monad, she needs to generalize the only term that features a higher-order construction, that is:

$$\mathrm{map} : \Pi A : \square.\, \Pi B : \square.\, (A \to B) \to \mathrm{T}\ A \to \mathrm{T}\ B$$

We would like to make the function argument dependent, so that we get a dependent function acting over monads in the end. This dependent map would therefore have the following form

$$\mathrm{map} : \Pi A : \square.\, \Pi B : A \to \square.\, (\Pi x : A.\, B\ x) \to \Pi \hat{x} : \mathrm{T}\ A.\, ?$$

where ? stands for some well-chosen type depending on $\hat{x}$. Alas, we face a somehow expected problem: in general, there is no way to escape T and thus we cannot state anything about $\hat{x}$, let alone plugging it into B which is expecting some $x : A$.

There are two ways to work around this issue. The probably most obvious one is to give a way to lift indexed families $B : A \to \square$ into $B^* : \mathrm{T}\ A \to \square$ in a similar fashion to

the bind operator that lifts term-level arrows. The problem is that there is no automatic way to do so, given a particular monad. Intuitively, if we see $\mathrm{T}\ A$ as a box containing A plus additional information, B^* should be the type defined on the content of that box. Yet, it is not possible in general to create a type out of a boxed element.

Rather than restricting ourselves to the case of monads that feature such a $(-)^*$ operator, we propose here a generic way to handle dependency when dealing with monads. It only requires the language to feature a dependent sum type, which is quite standard.

Definition 48 (Dependent sum)**.** We extend our type theory with the following terms and typing rules.

$$t, u, A, B := \ldots \mid \Sigma x : A.\, B \mid (t, u) \mid \texttt{match}\ t\ \texttt{with}\ (x, y) \mapsto u$$

$$\frac{\Gamma \vdash A : \square \qquad \Gamma, x : A \vdash B : \square}{\Gamma \vdash \Sigma x : A.\, B : \square}$$

$$\frac{\Gamma \vdash \Sigma x : A.\, B : \square \qquad \Gamma \vdash t : A \qquad \Gamma \vdash u : B[x := t]}{\Gamma \vdash (t, u) : \Sigma x : A.\, B}$$

$$\frac{\Gamma \vdash t : \Sigma x : A.\, B \qquad \Gamma, x : A, y : B \vdash u : C[z := (x, y)] \qquad z \text{ fresh}}{\Gamma \vdash \texttt{match}\ t\ \texttt{with}\ (x, y) \mapsto u : C[z := t]}$$

We also consider the usual reduction rule for pairs.

$$(\texttt{match}\ (t, u)\ \texttt{with}\ (x, y) \mapsto r) \to r[x := t, y := u]$$

The dependent sum features an adjunction with the dependent arrow akin to the one relating the usual product and arrow, namely:

$$(\Pi p : (\Sigma x : A.\, B).\, C) \cong (\Pi x : A.\, \Pi y : B.\, C[p := (x, y)])$$

We take advantage of this adjunction to work around the dependency issue, by embedding the term on which the return type depends in a sum. That is, we want a term

$$\mathrm{dmap} : \Pi A : \square.\, \Pi B : A \to \square.\, (\Pi x : A.\, B\ x) \to \mathrm{T}\ A \to \mathrm{T}\ (\Sigma x : A.\, B\ x)$$

which acts as a dependent version of the usual map operator. It is actually easily derived from the non-dependent version, which also gives its semantics for free. We can pose indeed

$$\mathrm{dmap} := \lambda A\, B\, (f : \Pi x : A.\, B)\, (m : \mathrm{T}\ A).\, \mathrm{map}\ A\ (\Sigma x : A.\, B)\ (\lambda x.\, (x, f\ x))\ m$$

Likewise, we can generalize the notion of monadic strength of Definition 39 to the dependent setting as follows.

$$\begin{array}{lcl} \mathrm{dstr} & : & \Pi A : \square.\, \Pi B : A \to \square.\, (\Sigma x : A.\, \mathrm{T}\ (B\ x)) \to \mathrm{T}\ (\Sigma x : A.\, B\ x) \\ & := & \lambda A\, B\, p.\, \mathtt{match}\ p\ \mathtt{with}\ (x, y) \mapsto \mathrm{join}\ \_\ (\mathrm{map}\ \_\ \_\ (\lambda y.\, \mathrm{return}\ \_\ (x, y))\ y) \end{array}$$

By using this simple scheme, we can use a bit of effects in our dependent type theory. Any first-order program potentially dependent can be lifted seamlessly. There are some important limitations though.

The main limitation of this straightforward use of dependent monads lies in the fact that there is no way to make effects flow at the level of types. Indeed, if we wish to create a type depending on effects, we will recover a term of type $\mathrm{T}\ \square$. Without further structure on our monad, there is no way to recover anything like a type from it. As we will see, in some cases this is actually possible, most notably when T is some form of a reader monad. Even with a term $\mathrm{run} : \mathrm{T}\ \square \to \square$, it is not obvious to come up with the conditions over it that would make it useful.

A direct consequence of this lack of effectful types is that one cannot state anything about the contents of a monadic type. Indeed, given a proposition of type $A \to \square$ for some A, there is no way to construct a proposition of type $\mathrm{T}\ A \to \square$ that would make sense without further knowledge of the innards of the monad. Therefore, this generic construction is not really useful in practice.

Although this construction is automatic and does not depend on the actual monad used, it is not satisfactory from the point of view of program proving. We exposed it here it to give a rough idea of the consequences of monadic encodings in dependent type theory.

We give in Chapter 11 a translation adding side-effects in a dependent setting that does not arise from a monadic encoding, and that performs well for the negative fragment. The problem of monadic escaping is washed away there thanks to the fact that the resulting term is essentially intuitionist and does not require any additional running operation. In Chapter 12, we present some translations that do arise from monadic decompositions, while still retaining a way to be adapted into a dependent system. This comes from the fact that the monad they are based on is a very special one, namely the reader monad. Those two cases may give further ideas on how easily one can monadify a dependent calculus.

5.2 Indexed CPS

We study in this section an interesting property featured by the dependent elimination in a specific case of side-effects, namely delimited continuations. Delimited continuations are provided by a generalization of the usual continuation-passing (or double-negation) monad, but contrarily to it, they allow the computation to escape the monad and return an effective value. This contrasts with the first-class continuations resulting from the use of the `callcc` operator which never return.

We recall that the usual double-negation monad is defined as

$$\mathrm{T}\ A := (A \to R) \to R$$

for some fixed R. We generalize it by parameterizing the monad by two types as follows.

Definition 49 (Indexed CPS)**.** For any types I and O, we define the type $\mathrm{T}_I^O\ A$ as:

$$\mathrm{T}_I^O\ A := (A \to I) \to O$$

We will call I the input type of the monad, and O its output type.

This finer-grained presentation allows to refine the types given to the usual monadic combinators.

Proposition 18. *The monadic combinators can be given the following types.*

$$\begin{array}{lcl} \mathrm{return} & := & \lambda x\, k.\, k\ x \\ & : & A \to \mathrm{T}_R^R\ A \end{array}$$

$$\begin{array}{lcl} \mathrm{bind} & := & \lambda m\, f\, k.\, m\ (\lambda x.\, f\ x\ k) \\ & : & \mathrm{T}_M^O\ A \to (A \to \mathrm{T}_I^M\ B) \to \mathrm{T}_I^O\ B \end{array}$$

The finer type of those combinators gives us a little insight into their computational behaviour. Indeed, the return operator constructs a computation that has the same input and output types, hinting at the fact it is pure. Dually, the type of the bind operator highlights the fact that it composes two effects, as their is an intermediate type appearing in the arguments that get erased in the resulting computation. The control flow is thus more explicit.

The interest of this additional typing information can be found in the new combinators it allows us to write and type.

Definition 50 (Delimited combinators)**.** We can define the following combinators.

$$\begin{array}{lcl} \mathrm{run} & := & \lambda m.\, m\ (\lambda x.\, x) \\ & : & \mathrm{T}_A^R\ A \to R \end{array}$$

$$\begin{array}{lcl} \mathrm{abort} & := & \lambda x\, k.\, x \\ & : & O \to \mathrm{T}_I^O\ A \end{array}$$

$$\begin{array}{lcl} \mathrm{catch} & := & \lambda m\, k.\, m\ k\ k \\ & : & \mathrm{T}_I^{\mathrm{T}_I^O\ A}\ A \to \mathrm{T}_I^O\ A \end{array}$$

The *run* operator allows us in particular to escape the monad, provided the input index agrees with the parameter type of the monad. Such a property is heavily used to build *delimited* continuations, where *run* actually plays the rôle of the delimiter.

We will not digress too much on the nice properties of delimited continuations, but there is a rich literature on this topic. In particular, it is known since Filinski [41] that direct-style delimited continuations allow to give a direct style encoding of any monad in call-by-value. The underlying idea is to hide the monadic types in the input and output

types of the monad, resulting in the illusion that there is no monadic encoding involved. The monadic operations do appear, but at the time of the delimitation only. We give in Chapter 12 a practical instance of this phenomenon, but in a call-by-name presentation, where continuations have more structure than just functions with a parameterized return type.

Instead, we will have a look at the interaction of indexed CPS with dependency, and in particular positive datatypes. Those subtle interactions should advocate for a look at effect handling in the light of dependency.

Definition 51 (Commutation morphism)**.** Given a n-ary connective F, we call a commutation morphism for F any term of type

$$F(\neg\neg A_1, \ldots, \neg\neg A_n) \to \neg\neg F(A_1, \ldots, A_n)$$

for any A_1, …, A_n where $\neg A$ stands for $A \to R$ for some fixed R.

Note that $\neg\neg A$ is a degenerate instance of the indexed CPS, because $\neg\neg A \equiv \mathrm{T}_R^R\ A$.

When considering the usual double-negation translation, there are a few commutations morphisms for the usual positive connectives.

Definition 52. We have the following commutation morphisms.

$$\begin{array}{lcl}
\theta_0 & := & \lambda m.\, \texttt{match}\ m\ \texttt{with}\ [\cdot] \\
 & : & 0 \to \neg\neg 0 \\
 & & \\
\theta_1 & := & \lambda m.\, \texttt{match}\ m\ \texttt{with}\ () \mapsto \lambda k.\, k\ () \\
 & : & 1 \to \neg\neg 1 \\
 & & \\
\theta_{A+B} & := & \lambda m.\, \texttt{match}\ m\ \texttt{with}\ [x \mapsto \lambda k.\, x\ (\lambda x.\, k\ (\mathbf{inl}\ x)) \mid y \mapsto \lambda k.\, y\ (\lambda y.\, k\ (\mathbf{inr}\ y))] \\
 & : & \neg\neg A + \neg\neg B \to \neg\neg(A + B) \\
 & & \\
\theta^l_{A\times B} & := & \lambda m.\, \texttt{match}\ m\ \texttt{with}\ (x, y) \mapsto \lambda k.\, x\ (\lambda x.\, y\ (\lambda y.\, k\ (x, y))) \\
 & : & \neg\neg A \times \neg\neg B \to \neg\neg(A \times B) \\
 & & \\
\theta^r_{A\times B} & := & \lambda m.\, \texttt{match}\ m\ \texttt{with}\ (x, y) \mapsto \lambda k.\, y\ (\lambda y.\, x\ (\lambda x.\, k\ (x, y))) \\
 & : & \neg\neg A \times \neg\neg B \to \neg\neg(A \times B)
\end{array}$$

As one can witness, there are two distinct natural way to make the product commute with the double negation. It corresponds to the two evaluation choices for the strict pair, that is, either forcing the left element first ($\theta^l_{A\times B}$) or the right one first ($\theta^r_{A\times B}$). Because the return type R is unspecified, there is no other way to write these commutations, and we are forced to commit ourselves to an evaluation strategy for the commutation of products. We will see in Chapter 12 a clever way to get around this restriction, by delimiting each term under a constructor.

For now, let us look at these terms when enriching the double-negation with type indexes, i.e. we turn $\neg\neg A$ into $\mathrm{T}_I^O\ A$ for some I and O. We trivially generalize the

notion of commutations to the indexed case, and we immediately manage to recover the following.

Definition 53. We have the following indexed commutation morphisms.

$$\begin{array}{lcl}
\hat{\theta}_0 & := & \lambda m.\, \texttt{match}\ m\ \texttt{with}\ [\cdot] \\
& : & 0 \to \mathrm{T}_I^O\ A \\
\\
\hat{\theta}_1 & := & \lambda m.\, \texttt{match}\ m\ \texttt{with}\ () \mapsto \lambda k.\, k\ () \\
& : & 1 \to \mathrm{T}_R^R\ 1 \\
\\
\hat{\theta}_{A+B} & := & \lambda m.\, \texttt{match}\ m\ \texttt{with}\ [x \mapsto \lambda k.\, \mathbf{inl}\ (x\ (\lambda x.\, \mathrm{fst}\ (k\ (\mathbf{inl}\ x)))) \mid y \mapsto \lambda k.\, \mathbf{inr}\ (y\ (\lambda y.\, \mathrm{snd}\ (k\ (\mathbf{inr}\ y))))] \\
& : & \mathrm{T}_I^O\ A + \mathrm{T}_J^P\ B \to \mathrm{T}_{I \times J}^{O+P}\ (A + B) \\
\\
\hat{\theta}^l_{A \times B} & := & \lambda m.\, \texttt{match}\ m\ \texttt{with}\ (x, y) \mapsto \lambda k.\, x\ (\lambda x.\, y\ (\lambda y.\, k\ (x, y))) \\
& : & \mathrm{T}_M^O\ A \times \mathrm{T}_I^M\ B \to \mathrm{T}_I^O\ (A \times B) \\
\\
\hat{\theta}^r_{A \times B} & := & \lambda m.\, \texttt{match}\ m\ \texttt{with}\ (x, y) \mapsto \lambda k.\, y\ (\lambda y.\, x\ (\lambda x.\, k\ (x, y))) \\
& : & \mathrm{T}_I^M\ A \times \mathrm{T}_M^O\ B \to \mathrm{T}_I^O\ (A \times B)
\end{array}$$

There are quite a few things to comment about these new combinators, even before trying to get them dependent.

First, the combinators for the unit and product types remain computationally identical, only their type is modified. Indeed, for the unit type, one must choose a arbitrary return type for the continuation. Meanwhile, for the product types, the order of evaluation is hard-written in the type of the combinators. The input type of one side matches the output type of the other side, thus indicating which is to be evaluated first.

For the empty and sum types, the situation is even harsher. The empty commutation morphism needs to answer an indexed CPS type totally arbitrarily, and the combinator for the sum type is distinct from its unindexed counterpart. There is no way indeed to statically discriminate in which case we are, so that the input type must be the product of both arguments' input type to be able to recover each projection in both cases. Likewise, the output type can be one of the output types of the two arguments, so that we need to return a sum.

In presence of dependent elimination, we can actually type the two unindexed commutation morphisms for the empty and sum types so that they also behave well in an indexed setting. They can be given the following fairly dependent types:

$$\begin{array}{lcl}
\theta_0 & := & \lambda m.\, \texttt{match}\ m\ \texttt{with}\ [\cdot] \\
& : & \Pi p : 0.\, \texttt{match}\ p\ \texttt{with}\ [\cdot] \\
\\
\theta_{A+B} & := & \lambda m.\, \texttt{match}\ m\ \texttt{with}\ [x \mapsto \lambda k.\, x\ (\lambda x.\, k\ (\mathbf{inl}\ x)) \mid y \mapsto \lambda k.\, y\ (\lambda y.\, k\ (\mathbf{inr}\ y))] \\
& : & \Pi p : (\mathrm{T}_I^O\ A + \mathrm{T}_J^P\ B).\ (\Pi p : A + B.\, \texttt{match}\ p\ \texttt{with}\ [x \mapsto I \mid y \mapsto J]) \to \\
& & \qquad \texttt{match}\ p\ \texttt{with}\ [x \mapsto O \mid y \mapsto P]
\end{array}$$

The idea behind those types is that, because they make commute a positive type with a monadic modality, they can observe the value they are being fed with. In particular, they can discriminate statically which branch is going to be taken (if any). Likewise, because the inner computation will be provided with a value, it can discriminate which type it will return.

In order to make these morphisms fit into our framework of indexed CPS, we need to generalize them a little to cope with the supplementary dependency. It is a simple matter of adding the possibility that the input type may depend on the value taken by the continuation. That is, instead of taking $A \to I$ in $\mathrm{T}_A^I\ O$ as a simple arrow, we rather consider it to be a telescope, i.e.

$$\Pi x_1 : A_1 . \Pi : \ldots . \Pi x_n : A_n . I$$

where I depends on $x_1, \ldots, x_n$ and each A_i depends on the preceding x_j. Our commutation morphisms naturally extend to this generalization, and in particular the term θ_{A+B} can be made dependent by simply considering that I and J depend respectively on x and y, and by making explicit those variables in the arrows $A \to I$ and $B \to J$.

This hints at the fact that there exists a generalization of CPS translations that behave better in presence of dependency. In particular, as witnessed by the commutation of the sum types, dependent elimination allows to type terms that lost typability in an indexed setting, at a low cost. Indeed, we do not require any fancy higher order system, but only look at a extension of the simply-typed calculus that features a very weak form of dependency. Delimited control arising from such monadic encoding may therefore require a form of dependent elimination to be fully satisfactory.

6 Logical by need

> From each according to his ability, to each according to his need.
>
> *Marx about computation sharing.*

We propose in this chapter a presentation of the so-called *call-by-need* calling convention based on considerations stemming from linear logic. Contrarily to historical presentations of the family of call-by-need calculi, the one we describe here is more uniform and shares strong bounds with other computation systems, most notably the KAM.

We will note Δ for $\lambda x.\, x\ x$, I for $\lambda y.\, y$ and we will write $\to_{\text{cbn}}$ (resp. $\to_{\text{cbv}}$) the strategies associated with call-by-name (resp. by-value). The redex which is involved in a reduction will be emphasized by showing it in a grey box.

6.1 An implicit tension

Executing computations which may not be used to produce a value may obviously lead to unnecessary work being done, potentially resulting in non-termination even when a value exists. An alternative is to fire a redex only when it happens to be necessary to pursue the evaluation towards a value.

For instance, it is well-known that call-by-value may trigger computations that could be completely avoided, resulting in potential non-termination, while call-by-name evaluates programs on demand. This is examplified in:

$$\begin{array}{rcl} t & \equiv & \boxed{(\lambda x.\, I)\ (\Delta\ \Delta)} \to_{\text{cbn}} I \\ t & \equiv & (\lambda x.\, I)\ \boxed{(\Delta\ \Delta)} \to_{\text{cbv}} t \to_{\text{cbv}} \ldots \to_{\text{cbv}} \ldots \end{array}$$

In this example, call-by-value reduction will reduce $\Delta\ \Delta$ again and again when the redex is of no use for reaching a value while call-by-name simply discards the argument.

Call-by-name, and more precisely weak head reduction thus realizes a form of demand-driven computation: a redex is fired only if it contributes to the weak head normal form, usually abbreviated as whnf.

On the other hand, while call-by-name is more parcimonious than call-by-value in terms of which parts of a program will be evaluated, call-by-value will happen to be more parcimonious when it comes to arguments which are actually used in the computation: they are evaluated only once, before substituting the value, while call-by-name discipline will redo the same computation several times, as in:

$$\begin{array}{rcl} u & \equiv & \Delta\ (I\ I) \to_{\mathrm{cbn}} I\ I\ (I\ I) \to_{\mathrm{cbn}} I\ (I\ I) \to_{\mathrm{cbn}} I\ I \to_{\mathrm{cbn}} I \\ u & \equiv & \Delta\ (I\ I) \to_{\mathrm{cbv}} \Delta\ I \to_{\mathrm{cbv}} I\ I \to_{\mathrm{cbv}} I \end{array}$$

In the above example, call-by-name reduction duplicates the computation of $I\ I$ while call-by-value only duplicates value I, resulting in a shorter reduction path to value.

Interestingly, demand-driven computation resulted in two lines of works, one motivated by theoretical purposes and rooted in logic, Danos and Regnier's linear head reduction, the other being motivated by more practical concerns and resulting in the study of lazy evaluation strategies for functional languages.

6.2 Linear head reduction

6.2.1 A brief history of the unloved linear head reduction

As far as we can trace it back, linear head reduction was first described by Regnier [96] in his 1992 PhD thesis, albeit under the name of *spinal reduction*. It arose from the study of the computational correspondance between proof-nets and the call-by-name λ-calculus. In this work, Regnier showed that linear head reduction was essentially an extension of the usual head reduction up to a relation called σ-equivalence that allowed to permute morally irrelevant but blocking redexes for the head reduction.

The proper name *linear head reduction* appears in print two years later in an article due to Mascari and Pedicini [79] that sums up and extends the results of Regnier on the natural relation between linear head reduction and proof-nets. They proved in particular that this reduction was the equivalent of the proof-net reduction through the call-by-name decomposition of linear logic, which is one of the reasons for the use of the linear adjective, the other one being that the substitutions are non-destructive, contrarily to what happens in the usual β-reduction. Likewise, the head qualificative can be justified by its closeness to head reduction, as exposed by Regnier.

This peculiar notion of reduction then made very scarce appearances in the literature for fifteen years, seemingly falling into oblivion except for the original authors. Indeed, it is the object of study in an article by Danos, Regnier and Herbelin [36], as well as in an unpublished note by the two former authors [37]. They build in them an abstract machine implementing this reduction, the Pointer Abstract Machine (PAM) that relies on a notion of pointers into subterms of a λ-term to cope with the linear substitution feature. As for the proof-net reduction, those documents show a strong relationship between the PAM (and thus linear head reduction) and game semantics [59].

Although at first sight it looked like linear head reduction had slowly faded away, it made a surprise comeback by the beginning of the 2010's through a research line initiated by Accattoli and Kesner [6]. Their seminal article describes the so-called structural λ-calculus, featuring explicit substitutions and at-distance reduction, taking once again inspiration from the computational behaviour of proof-nets and revamping the σ-equivalence relation in this framework.

The at-distance reduction in particular contrasts sharply with the usual treatment of explicit substitutions which generally require a set of rules allowing the substitutions to

commute with the various contexts. In their system, blocks of explicits substitutions are instead stuck to the place where they were created and are considered transparent for all purposes but the rule of substitution of variables.

This is practically done by splitting the usual λ-calculus reduction in two phases, paralleling what happens in the proof-net reduction: multiplicative steps, corresponding to the creation of explicit substitutions, and exponential steps, corresponding to the effective substitution of a variable by some term. Linear head reduction then naturally arises from the call-by-name flavour of those two rules as described, and indeed the connection with the historical linear head reduction is made explicit in many articles from the subsequent trend [8, 6, 4] and is furthemore used to obtain results ranging from computational complexity to factorization of rewriting theories of the λ-calculus [9, 5, 10, 7].

Such a vast array of results suggests that linear head reduction deserves more attention and ought to be studied in more details. In this chapter, we will give a new alternative presentation of it, based on the novel use of a unusual class of term contexts.

6.2.2 The old-fashioned linear head reduction

From an abstract point of view, linear head reduction allows to synthesize similar observations made amongst different computational paradigms, namely the Krivine abstract machine [68], proof-nets [44, 96], and game semantics [59].

The basic claim is that the core of these systems does not implement the usual head reduction as thought commonly, but rather use some more parcimonious reduction, which they define under the name of linear head reduction, which realizes a stronger form of computation-on-demand than call-by-name: the argument of a function cannot be said to truly contribute to the result if it never reaches head position; in such a case, the corresponding redex may only contribute to the (w)hnf in a non-essential way; for instance by blocking other redexes as in $(\lambda x\, y.\, y)\ t\ u$. Linear head reduction makes this observation formal.

Linear head reduction has two main features:

- first it reduces only the β-redex binding to the leftmost variable (therefore the *head* from its name);
- secondly it substitutes for the argument only the head occurrence of the variable (therefore the *linear* from its name) without destroying the fired redex.

A third noticeable point is that linear head reduction is not truly a reduction in that it does not reduce only redexes (at least not only β-redexes), but also sorts of hidden β-redexes that are true β-redexes only up to an equivalence on λ-terms induced by their encoding in proof nets, namely σ-equivalence. This point shall be made clear later on.

We now recall Danos and Regnier's definition of linear head reduction:

Definition 54. The *spine* of a λ-term t is the set $\restriction t$ of λ-terms inductively defined as:

$$\frac{}{t \in \restriction t} \qquad \frac{r \in \restriction t}{r \in \restriction t\ u} \qquad \frac{r \in \restriction t}{r \in \restriction \lambda x.\, t}$$

$\uparrow t$ always contains exactly one variable by construction, written $\mathrm{hoc}(t)$, for *head occurrence*.

Stated another way, the spine of t is nothing more than the set of left subterms of t, $\mathrm{hoc}(t)$ being its leftmost variable.

Definition 55 (Head lambdas, Prime redexes)**.** Let t be a λ-term. The head lambdas $\lambda_h(t)$ and the prime redexes $p(t)$ of t are mutually defined by induction on t as follows.

$$\begin{array}{rclcrcl}
\lambda_h(x) & := & \varepsilon & & p(x) & := & \emptyset \\
\lambda_h(\lambda x.\, t) & := & x :: \lambda_h(t) & & p(\lambda x.\, t) & := & p(t) \\
\lambda_h(t\ u) & := & \begin{cases} \varepsilon \\ \ell \end{cases} & & p(t\ u) & := & \begin{cases} p(t) & \text{if } \lambda_h(t) = \varepsilon \\ p(t) \cup \{x \leftarrow u\} & \text{if } \lambda_h(t) = x :: \ell \end{cases}
\end{array}$$

Remark 4. To understand head lambdas and prime redexes, it is convenient to consider blocks of applications. We have indeed the following equalities.

$$\begin{array}{rcl}
\lambda_h((\lambda x.\, t)\ u\ \vec{r}) & \equiv & \lambda_h(t\ \vec{r}) \\
p((\lambda x.\, t)\ u\ \vec{r}) & \equiv & \{x \leftarrow u\} \cup p(t\ \vec{r})
\end{array}$$

Head lambdas are precisely lambdas from the spine which will not be fed with arguments during head reduction. Now that we are equipped with the above notions, we can formally define the linear head reduction:

Definition 56 (Linear head reduction)**.** Let u a λ-term, let $x := \mathrm{hoc}(u)$. We say that u linear-head reduces to r, written $u \to_{\mathtt{lh}} r$, when:

- there exists some term t s.t. $\{x \leftarrow t\} \in p(u)$;
- r is u where the variable occurrence $\mathrm{hoc}(u)$ has been substituted by t.

Remark 5. Linear head reduction only substitutes one occurrence of a variable at a time and never reduces an application node. Likewise, it does not decrease the number of prime redexes. Thus terms keep growing, hence the name *linear* taken for linear *substitution*. An example of linear head reduction is given in Figure 6.1.

6.3 Lazy evaluation

Wadsworth introduced lazy evaluation [106] as a mean to overcome defects of both call-by-name and call-by-value evaluation recalled in the above paragraphs. Lazy evaluation, or Call-by-need, can be viewed as a strategy reconciling the best of the by-value and by-name worlds in terms of reductions: a computation is triggered only when it is needed for the evaluation to progress and, in this case, it avoids redoing computations.

The price to pay is that the by-need strategy is tricky to formulate and reason about. For instance, Wadsworth had to introduce a graph reduction in order to allow sharing of sub-terms, and the following developments on lazy evaluation essentially dealt with machines. The essence of call-by-need is summarized by Danvy et al. [39]:

Demand-driven computation & memoization of intermediate results

Designing a proper calculus for call-by-need remained open for about two decades, until the mid-nineties when, in 1994, two very close solutions to this problem were simultaneously presented by Ariola and Felleisen on the one hand, and Maraist, Odersky and Wadler on the other [14, 11, 78]. Ariola and Felleisen's calculus can be presented as follows.

Definition 57. AF-calculus is defined by the following syntax:

Syntax

Term	t, u	$:=$	$x \mid \lambda x.\, t \mid t\ u$
Value	v	$:=$	$\lambda x.\, t$
Answer	A	$:=$	$v \mid (\lambda x.\, A)\ t$
Evaluation context	E	$:=$	$[\cdot] \mid E\ t \mid (\lambda x.\, E)\ t \mid (\lambda x.\, E[x])\ E$

Reductions

(Deref)	$(\lambda x.\, E[x])\ v$	$\to$	$(\lambda x.\, E[v])\ v$
(Lift)	$(\lambda x.\, A)\ t\ u$	$\to$	$(\lambda x.\, A\ u)\ t$
(Assoc)	$(\lambda x.\, E[x])\ ((\lambda y.\, A)\ t)$	$\to$	$(\lambda y.\, (\lambda x.\, E[x])\ A)\ t$

Intuitively, the above calculus shall be understood as follows:

- The lazy behaviour of the calculus is coded in the structure of contexts: term $E[x]$ evidences that variable x is in needed position in term $E[x]$.
- Rule Deref then gets the argument, in case it has already been computed and it has been detected as needed. In that case, the argument is substituted for one copy of the variable x, the one in needed position. As a consequence, the application is not erased and a single occurrence of the variable has been substituted. (E is a single-hole context.)
- Rules Lift and Assoc allow for the commutation of evaluation contexts in order for deref redexes to appear despite the persisting binders.

We give an example of a reduction sequence in Ariola-Felleisen call-by-need λ-calculus in figure 6.1. In the last line we highlight the term that would remain after having applied the garbage-collection rule considered by Maraist et al [78]. Even though this is not part of the calculus, this convention of garbage-collecting weakening redexes will be used in the rest of this chapter in order to ease the reading of values.

6.4 Linear head reduction versus call-by-need

Linear head reduction and call-by-need have striking common features:

- call-by-need can be seen as an optimization of both call-by-name and call-by-value while linear head reduction can be seen as an optimization of head reduction;

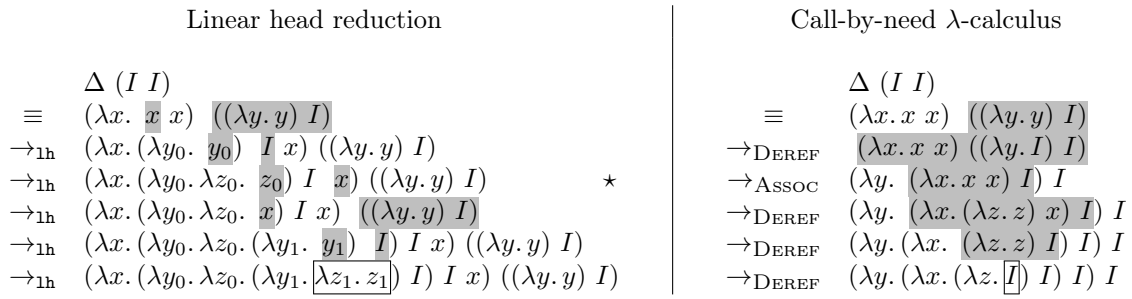

Figure 6.1: Linear head reduction vs. Ariola-Felleisen calculus.

- both rely on a linear, rather than destructive, substitution (at least in Ariola-Felleisen calculus presented above).
- more importantly, both share with call-by-name the same notion of convergence and the induced observational equivalences. Being observationally indistinguishable in the pure λ-calculus, they require instead side-effects to be told apart from call-by-name.

While it took two decades for call-by-need to be equipped with a proper calculus, the way linear head reduction is usually defined is intricate and inconvenient to work with. We actually view this fact, together with the observational indistinguishability, as one of the reasons for the almost complete inexistence of linear head reduction in literature for about two decades.

In the present chapter, we aim at remedying this unfairness and making formal the deep connections between linear head reduction and call-by-need which are not the least contingent. To this aim, we shall thus redefining here the required notions.

The connections between the two formalisms are striking and actually not the least contingent. Understanding the precise relationships between the two may be useful to build call-by-need on firm, logical grounds. While comparing the merits of various call-by-need calculi in order to evidence one such calculus as being more canonical than the other may be quite dubious [1], when extending call-by-need with control not only do we make call-by-name and call-by-need observational equivalences differ, but also can we distinguish between several call-by-need calculi [12] as evidenced by the work of Saurin and Herbelin about defining call-by-need extensions of $\overline{\lambda}\mu\tilde{\mu}$ which are sequent-style $\lambda\mu$-calculus [35]. In this context, it does make sense to wonder which calculus to pick and what observational impact these choice may have. Somehow, we can summarize the aim of the current presentation as integrating logically call-by-need and control operators in a different way from previous work [12]. While the framework of [35] call-by-need is added

[1] For instance Maraist, Odersky and Wadler calculus differ in their 1998 journal version from the calculus introduced by Ariola Felleisen, but in no essential way since both calculi share the same standard reductions.

to control, we suggest to work the other way around: we want to integrate as cleanly as possible call-by-need in an intuitionistic setting before lifting it to the classical setting.

We contribute to the theory of linear head reduction and show that it can be made into calculus. Doing so will allow us to formally connect linear hear reduction to call-by-need, showing how to systematically derive well-known call-by-need calculi from linear head reduction. More precisely, we will we will justify the following motto:

Lazy	$\equiv$	Demand-driven comp. *(weak linear head reduction)*	+	Memoization *(by value)*	+	Sharing *(closure sharing)*

We will show in Section 6.6 that three steps lead us from linear head reduction to call-by-need:

1. restriction to a weak linear head reduction by specializing closure contexts,
2. then enforcing memoization of intermediate results by restricting to value passing;
3. and finally implementing some sharing, thanks to closure contexts.

Our results are validated in two ways: first, the resulting calculi correspond to well-known call-by-need calculi, providing a validation to Chang-Felleisen recent single-axiom call-by-need calculus [27] with one axiom. Second, we extend our results to the classical case, defining linear head reduction for $\lambda\mu$-calculus and deriving from it a call-by-need $\lambda\mu$-calculus.

6.5 A modern reformulation of linear head reduction

In the previous section, we recalled the historical presentation [37] of linear head reduction. In the present section we give a new formulation of the reduction based on *closure contexts* from which we progressively build a call-by-need calculus, but first we recall some facts about σ-equivalence.

6.5.1 Reduction up to σ-equivalence

It is noteworthy that head linear reduction reduces terms which are not yet redexes for β, i.e. `lh` may get the argument of a binder even if it is not directly applied to it. The third reduction ($\star$) of the example from Figure 6.1 features such a cross-redex reduction. In this reduction, the λz_1 binder steps across the prime redex $\{y_0 \leftarrow \lambda z_0.\, z_0\}$ in order to recover its argument x. This kind of reduction would not have been allowed by the usual head reduction. This peculiar behaviour can be made more formal thanks to a rewriting up to equivalence, also introduced by Regnier [97, 96].

Definition 58 (σ-equivalence)**.** σ-equivalence is the reflexive, symmetric and transitive closure of to binary relation on λ-terms generated by:

$$\begin{array}{lcll} (\lambda x.\, t)\ u\ v & \cong_\sigma & (\lambda x.\, t\ v)\ u & \text{with } x \text{ fresh for } v \\ (\lambda x\, y.\, t)\ u & \cong_\sigma & \lambda y.\, (\lambda x.\, t)\ u & \text{with } y \text{ fresh for } u \end{array}$$

with the expected freshness conditions on bound variables.

Intuitively, σ-equivalence allows reduction in a term where it would have been forbidden by other essentially transparent redexes.

Proposition 19. *If $t \cong_\sigma u$, then $p(t) = p(u)$.*

The following proposition highlights the strong kinship relating linear head reduction and σ-equivalence. Let us recall that a left context L is inductively defined by the following grammar:

$$L := [\cdot] \mid L\ t \mid \lambda x.\, L$$

Proposition 20. *If $t \to_{\mathtt{lh}} r$ then there exist two left contexts L_1 and L_2 such that*

$$t \cong_\sigma L_1[(\lambda x.\, L_2[x])\ u] \quad \textit{and} \quad r \cong_\sigma L_1[(\lambda x.\, L_2[u])\ u]$$

The previous result can be slightly refined. The $\cong_\sigma$ relation is reversible, so that we can rebuild r by applying to $L_1[(\lambda x.\, L_2[u])\ u]$ the reverse σ-equivalence steps from the rewriting from t to $L_1[(\lambda x.\, L_2[x])\ u]$. We will not detail this operation here but rather move to the definition of closure contexts.

6.5.2 Closure contexts

With the aim to give a first-class status to the reduction up to σ-equivalence of Proposition 20, we introduce a new kind of reduction contexts, *closure contexts*.

Definition 59 (Closure contexts)**.** Closure contexts are inductively defined as:

$$\mathcal{C} := [\cdot] \mid \mathcal{C}_1[\lambda x.\, \mathcal{C}_2]\ t$$

While closure contexts may seem odd at first, they feature all the required properties that provide them with a nice algebraic behaviour, that is, composability and factorization.

Proposition 21 (Composition)**.** *Let $\mathcal{C}_1$ and $\mathcal{C}_2$ be closure contexts, then $\mathcal{C}_1[\mathcal{C}_2]$ is also one.*

Proposition 22 (Factorization)**.** *Any term t can be uniquely decomposed as a maximal closure context, in the usual meaning of composition, and a subterm t_0.*

Actually, we get even more: closure contexts precisely capture the notion of prime redex.

Proposition 23. *Let t be a term. Then $\{x \leftarrow u\} \in p(t)$ if and only if there exist a left context L, a closure context $\mathcal{C}$ and a term t_0 such that $t = L[\mathcal{C}[\lambda x.\, t_0]\ u]$.*

Closure contexts are morally transparent for some well-behaved head reduction: one can consider that $\mathcal{C}[\lambda x.\,[\cdot]]\ t$ is a context that only adds a binding $(x := t)$ to the environment, as well as the bindings contained in $\mathcal{C}$.

This intuition can be made formal thanks to the Krivine abstract machine (KAM). As stated by the following result, transitions Push and Pop of the KAM implement the computation of closure contexts.

Proposition 24. *Let t be a term, σ an environment, π a stack and $\mathcal{C}$ a closure context. We have the following reduction*

$$\langle (\mathcal{C}[t], \sigma) \mid \pi \rangle \longrightarrow^*_{\textsc{Push}, \textsc{Pop}} \langle (t, \sigma + [\mathcal{C}]_\sigma) \mid \pi \rangle$$

where $[\mathcal{C}]_\sigma$ is defined by induction over $\mathcal{C}$ as follows:

$$[[\cdot]]_\sigma \equiv \emptyset \qquad [\mathcal{C}_1[\lambda x.\,\mathcal{C}_2]\ t]_\sigma \equiv [\mathcal{C}_1]_\sigma + (x := (t, \sigma)) + [\mathcal{C}_2]_{\sigma + [\mathcal{C}_1]_\sigma + (x := (t,\sigma))}$$

Conversely, for all t_0 and σ_0 such that

$$\langle (t, \sigma) \mid \pi \rangle \longrightarrow^*_{\textsc{Push}, \textsc{Pop}} \langle (t_0, \sigma_0) \mid \pi \rangle$$

there exists $\mathcal{C}_0$ such that $t = \mathcal{C}_0[t_0]$, where $\mathcal{C}_0$ is inductively defined over σ_0.

6.5.3 The $\lambda_{\mathtt{lh}}$-calculus

Owing to the fact that closure contexts capture prime redexes, we will provide an alternative and more conventional definition for the linear head reduction. It will result in the $\lambda_{\mathtt{lh}}$-calculus, based on contexts rather than ad-hoc variable manipulations.

Definition 60 ($\lambda_{\mathtt{lh}}$-calculus)**.** The $\lambda_{\mathtt{lh}}$-calculus is defined by the reduction rule:

$$L_1[\mathcal{C}[\lambda x.\, L_2[x]]\ u] \to_{\lambda_{\mathtt{lh}}} L_1[\mathcal{C}[\lambda x.\, L_2[u]]\ u]$$

where L_1, L_2 are left contexts, $\mathcal{C}$ is a closure context, t and u are λ-terms, with the usual freshness conditions to prevent variable capture in u.

Proposition 25 (Stability of $\lambda_{\mathtt{lh}}$ by σ)**.** *Let t, u and v be terms such that $t \cong_\sigma u \to_{\mathtt{lh}} v$, then there is w such that $t \to_{\mathtt{lh}} w \cong_\sigma v$.*

Theorem 13. *The $\lambda_{\mathtt{lh}}$-calculus captures the linear head reduction.*

$$t \to_{\lambda_{\mathtt{lh}}} r \quad \textit{iff} \quad t \to_{\mathtt{lh}} r$$

Indeed, the x from the rule is precisely the hoc of the term, and because we are reducing up to closure contexts, Proposition 23 ensures that $\{x \leftarrow u\}$ is a prime radical. Hence the expected result.

6.5.4 LHR with microscopic reduction

Instead of providing our calculus with macroscopic rules that work on a whole contextified term, we can also describe it using microscopic reduction based on atomic, small-step reductions. To this end, we need to switch to a `let`-based calculus, whose syntax is defined below.

$$\begin{array}{lcl} t, u & := & x \mid \lambda x.\, t \mid t\ u \mid \texttt{let}\ x := t\ \texttt{in}\ u \\ E & := & [\cdot] \mid E\ t \mid \texttt{let}\ x := t\ \texttt{in}\ E \end{array}$$

$$\begin{array}{lcll} (\lambda x.\, t)\ u & \to & \texttt{let}\ x := u\ \texttt{in}\ t & (\textsc{Let}) \\ (\texttt{let}\ x := u\ \texttt{in}\ \mathcal{C}[\lambda y.\, t])\ r & \to & \texttt{let}\ x := u\ \texttt{in}\ \mathcal{C}[\lambda y.\, t]\ r & (\textsc{Lift}) \\ \mathcal{C}[\texttt{let}\ x := u\ \texttt{in}\ \lambda y.\, t]\ r & \to & \mathcal{C}[\lambda y.\, \texttt{let}\ x := u\ \texttt{in}\ t]\ r & (\textsc{Dig}) \\ \texttt{let}\ x := t\ \texttt{in}\ E[x] & \to & \texttt{let}\ x := t\ \texttt{in}\ E[t] & (\textsc{Subst}) \end{array}$$

As it is the norm in such calculi, the `let` binder allows to track β-redexes by pairing them whenever they appear as such. This is precisely the rôle of the following rule:

$$(\lambda x.\, t)\ u \to \texttt{let}\ x := u\ \texttt{in}\ t$$

In the macroscopic calculus, this rôle was devoted to the closure contexts instead. Now that we have first-class `let`-s, we need to adapt those contexts. This is easily done by replacing pairs of prime redexes by a corresponding `let`, as follows:

$$\mathcal{C} := [\cdot] \mid \texttt{let}\ x := t\ \texttt{in}\ \mathcal{C}$$

There is a small loss of information though, as the previous closure contexts had a slightly more complex data structure than the current ones, which are isomorphic to a plain list of binders.

The σ-equivalence rules can be rewritten to fit into this presentation as follows:

$$\begin{array}{lcl} (\texttt{let}\ x := u\ \texttt{in}\ t)\ v & \cong_\sigma & \texttt{let}\ x := u\ \texttt{in}\ t\ v \\ \texttt{let}\ x := u\ \texttt{in}\ \lambda y.\, t & \cong_\sigma & \lambda y.\, \texttt{let}\ x := u\ \texttt{in}\ t \end{array}$$

with the usual freshness conditions.

Because of the slight mismatch between usual closure contexts and `let`-based context closures, there is a choice to be made in the microscopic reduction rule. While the original closure contexts made prime redexes appear naturally, in the `let`-calculus, we need to explicitly create β-redexes by making the potential redexes commute with the surrounding context. There are two ways to do so, each one corresponding to a generating rule of the σ-equivalence:

- Either by pushing the application node inside closure contexts:

$$(\texttt{let}\ x := u\ \texttt{in}\ \mathcal{C}[\lambda y.\, t])\ v \to \texttt{let}\ x := u\ \texttt{in}\ \mathcal{C}[\lambda y.\, t]\ v \qquad (\textsc{Lift}).$$

- Or by extruding applied λ-abstractions from the surrounding context:

$$\mathcal{C}[\texttt{let}\ x := u\ \texttt{in}\ \lambda y.\, t]\ v \to \mathcal{C}[\lambda y.\, \texttt{let}\ x := u\ \texttt{in}\ t]\ v \qquad (\textsc{Dig}).$$

Reduction Lift is the most common in literature, probably because it is easier to formulate without a clear notion of closure contexts. It corresponds to the call-by-name calculus described in [39], where answers A are no more than terms of the form $\mathcal{C}[\lambda x.\, t]$. Reduction Dig is an alternative choice. The two reductions are not only different, but also incompatible, in the sense that their left-hand sides are the same while their right-hand sides are not convertible, thus breaking confluence. Yet, the reduced terms still agree up to σ-equivalence by construction.

Finally, the last rule performs the linear substitution:

$$\texttt{let}\ x := t\ \texttt{in}\ E[x] \to \texttt{let}\ x := t\ \texttt{in}\ E[t]$$

Here, $E[x]$ represents an evaluation context, so that the substitution only replaces exactly one variable.

6.6 Towards call-by-need

Our journey from linear head reduction to call-by-need will now follow three steps: first restricting linear head reduction to a weak reduction, then imposing a value-restriction and finally introducing an amount of sharing.

6.6.1 Weak linear head reduction

The linear head reduction as given at paragraph 6.5.3 is a *strong* reduction: it reduces under abstractions. We now adapt $\lambda_{\texttt{lh}}$-calculus to the weak case. It is easy to give a weak version of the reduction in the historical setting of Danos-Regnier, which inherits the same defects as its strong counterpart.

Definition 61 (Historical `wlh`-reduction)**.** We say that t weak-linear-head reduces to r, which we will write $t \to_{\texttt{wlh}} r$, iff $t \to_{\texttt{lh}} r$ and t does not have any head λ.

On the other hand, the $\lambda_{\texttt{lh}}$ reduction can be denied the possibility to reduce under abstractions by restricting the evaluation contexts inside which it can be triggered. This requires some care though. Indeed, the contexts may contain λ-abstractions, assuming they have been compensated by as many previous applications. That is, those binders must pertain to a prime redex as in $(\lambda z_1 \dots\ z_n\, x.\, E^w[x])\ r_1\ \dots\ r_n\ u$.

Plain closure contexts are not enough to capture this kind of situation. We need to split the enclosing context in two parts, applicative and binding contexts inductively defined as:

$$\mathcal{C}^{@} := \mathcal{C} \mid \mathcal{C}[\mathcal{C}^{@}\ t] \quad \mathcal{C}_\lambda := \mathcal{C} \mid \mathcal{C}[\lambda x.\, \mathcal{C}_\lambda]$$

While applicative contexts introduce supernumerary applications, binding contexts consume them. It would be only possible to consider pairs of context such that their composition would be balanced.

Somehow, those contexts are the insensibilization of binding (resp. applicative) contexts $\mathcal{C}_\lambda$ (resp. $\mathcal{C}^{@}$) to closure contexts by sandwiching each abstraction (resp. application) node with as many closures as required.

We can formally define the filling relation allowing us to define the weak linear head reduction.

Definition 62 (Filling)**.** We define inductively the size function $|\cdot|_\lambda$ (resp. $|\cdot|_@$) on binding (resp. applicative) contexts below.

$$\begin{array}{ll} |\mathcal{C}|_\lambda := 0 & |\mathcal{C}|_@ := 0 \\ |\mathcal{C}[\lambda x.\, \mathcal{C}_\lambda]|_\lambda := 1 + |\mathcal{C}_\lambda|_\lambda & \left|\mathcal{C}[\mathcal{C}^@\ u]\right|_@ := 1 + \left|\mathcal{C}^@\right|_@ \end{array}$$

We finally pose $\mathcal{C}^@ \ni \mathcal{C}_\lambda$ when $\left|\mathcal{C}^@\right|_@ \geq |\mathcal{C}_\lambda|_\lambda$.

For the sake of readability, we will write as $\mathcal{C}^w$ an up-to-closure applicative context that is not expected to be composed with a binding context.

Definition 63 ($\lambda_{\texttt{wlh}}$-calculus)**.** The weak linear head calculus $\lambda_{\texttt{wlh}}$ is defined by the rule:

$$\mathcal{C}^@[\mathcal{C}[\lambda x.\, \mathcal{C}_\lambda[\mathcal{C}_w[x]]]\ u] \quad \to_{\lambda_{\texttt{wlv}}} \quad \mathcal{C}^@[\mathcal{C}[\lambda x.\, \mathcal{C}_\lambda[\mathcal{C}_w[u]]]\ u] \qquad \text{if } \mathcal{C}^@ \ni \mathcal{C}_\lambda$$

Proposition 26 (Stability of $\lambda_{\texttt{wlh}}$ by σ)**.** *Let t, u and v be terms such that $t \cong_\sigma u \to_{\texttt{wlh}} v$, then there is w such that $t \to_{\texttt{wlh}} w \cong_\sigma v$.*

Remark 6. If $\mathcal{C}^@ \ni \mathcal{C}^\lambda$, then $\mathcal{C}^@[\mathcal{C}^\lambda]$ is an up-to-closure applicative context.

For completeness purpose, we can also give an alternative definition of this reduction in the historical setting of Danos-Regnier. This presentation inherits the same defects as its strong counterpart.

Definition 64 (Historical `wlh`-reduction)**.** We say that t weak-linear-head reduces to r, which we will write $t \to_{\texttt{wlh}} r$, iff t linear-head reduces to r and t does not have any head lambda.

We can now compare the historical weak linear head reduction with $\lambda_{\texttt{wlh}}$. Intuitively, they correspond since not having head lambda is exactly equivalent to having its binding subcontexts filled. Somehow, the historical presentation of the weak reduction implicitly deals with binding contexts. Actually, the two notions of weak reduction are the same:

Theorem 14. *The $\lambda_{\texttt{wlh}}$-calculus and the historical* `wlh`*-reduction coincide, that is $t \to_{\lambda_{\texttt{wlh}}} r$ iff $t \to_{\texttt{wlh}} r$.*

Proof. They correspond since not having head lambda is exactly equivalent to having its binding subcontexts filled.

□

6.6.2 Call-by-value linear head reduction

In order to obtain a call-by-value linear head reduction, we will restrict contexts that trigger substitutions to react only in front of a value. In addition, the up-to-closure paradigm used so far will also incite us to consider values up to closures defined as

$$w := \mathcal{C}[v]$$

when v stands for values.

Going from the usual call-by-name to the usual call-by-value is then simply a matter of adding a context forcing values. Likewise, we just add a context forcing up-to values. This construction is made in a systematic way according to the standard call-by-value encoding.

$$E^v := [\cdot] \mid E^v\ u \mid \mathcal{C}^@[\mathcal{C}[\lambda x.\, \mathcal{C}_\lambda[E_1^v[x]]]\ E_2^v] \mid \mathcal{C}[E^v] \qquad \text{where } \mathcal{C}^@ \ni \mathcal{C}_\lambda$$

Let us insist that although we added a dedicated rule $\mathcal{C}[E^v]$ to reason up to context, this is actually only for readability purposes. We could also proceed to the inlined sandwiching exactly like in the previous section. The call-by-value weak linear head reduction is then obtained straightforwardly:

Definition 65. The $\lambda_{\texttt{wlv}}$-calculus is defined by the unique reduction rule:

$$\mathcal{C}^@[\mathcal{C}[\lambda x.\, \mathcal{C}_\lambda[E^v[x]]]\ w] \quad \to_{\lambda_{\texttt{wlv}}} \quad \mathcal{C}^@[\mathcal{C}[\lambda x.\, \mathcal{C}_\lambda[E^v[w]]]\ w] \qquad \text{if } \mathcal{C}^@ \ni \mathcal{C}_\lambda$$

It is easy to check that the reduction rule itself was not deeply modified. The essential difference lies in the clever choice for the contexts.

Proposition 27 (Stability of $\lambda_{\texttt{wlv}}$ by σ)**.** *Let t, u and v be terms such that $t \cong_\sigma u \to_{\texttt{wlv}} v$, then there is w such that $t \to_{\texttt{wlv}} w \cong_\sigma v$.*

Although we branded this calculus as a call-by-value one, it already implements a call-by-need strategy since it triggers the reduction of an argument if and only if it was made necessary by the encounter of a corresponding variable in (call-by-value) hoc position. We give the reduction on our running example:

$$\begin{array}{lll}
\Delta\ (I\ I) & \equiv & (\lambda x.\, x\ x)\ ((\lambda y.\ y)\ \ I) \\
& \to_{\texttt{wlv}} & (\lambda x.\ \ x\ x)\ \ ((\lambda y.\, I)\ I) \\
& \to_{\texttt{wlv}} & (\lambda x.\, (\lambda y_1\, z_1.\, z_1)\ I\ \ x)\ ((\lambda y.\, I)\ I) \\
& \to_{\texttt{wlv}} & (\lambda x.\, (\lambda y_1\, z_1.\ \ z_1)\ I\ ((\lambda y.\, I)\ I))\ ((\lambda y.\, I)\ I) \\
& \to_{\texttt{wlv}} & (\lambda x.\, (\lambda y_1\, z_1.\, \boxed{I})\ I\ ((\lambda y.\, I)\ I))\ ((\lambda y.\, I)\ I)
\end{array}$$

In the first transition, the reduction occurs in the argument required by x, returning a value (up to closure) that will then be substituted.

6.6.3 Closure sharing

More complex call-by-need reduction schemes from the literature [11] cannot be captured by the $\lambda_{\texttt{wlv}}$-calculus. Indeed, the scrutiny of the example from the previous section reveals a duplication of computation:

$$\mathcal{C}^@[\mathcal{C}'[\lambda x.\, \mathcal{C}_\lambda[E^v[x]]]\ \mathcal{C}[v]] \to_{\lambda_{\texttt{wlv}}} \mathcal{C}^@[\mathcal{C}'[\lambda x.\, \mathcal{C}_\lambda[E^v[\mathcal{C}[v]]]]\ \mathcal{C}[v]]$$

In that case, we copied the closure context $\mathcal{C}$, which will end up in recomputing its bound terms if ever they are going to be used throughout the reduction. While our running example $\Delta\ (I\ I)$ does not feature such a behaviour, this can instead be seen on

$(\lambda x.\, x\ I\ x)\ (\lambda y\, z.\, y)\ (I\ I)$ because while the term to the right of the application node is already an up-to value, it also uses the argument $I\ I$ from its closure. During the substitution, this subterm is copied as-is, resulting in its recomputation at each call to x in the body of the abstraction.

It is possible to solve this issue in an elegant way akin to the Assoc rule of Ariola-Felleisen calculus. This is achieved by the extrusion of the closure of the value at the instant it is substituted. There is no need to refine contexts further, because everything is already in order. We obtain the calculus below.

Definition 66. The call-by-value linear head calculus with sharing is defined by the unique reduction rule:

$$\mathcal{C}^{@}[\mathcal{C}'[\lambda x.\, \mathcal{C}_\lambda[E^v[x]]]\ \mathcal{C}[v]] \quad \to_{\lambda_{\texttt{wls}}} \quad \mathcal{C}^{@}[\mathcal{C}[\mathcal{C}'[\lambda x.\, \mathcal{C}_\lambda[E^v[v]]]\ v]] \qquad \text{if } \mathcal{C}^{@} \ni \mathcal{C}_\lambda$$

with the usual freshness conditions to prevent variable capture in $\mathcal{C}'$.

6.6.4 $\lambda_{\texttt{wls}}$ is a call-by-need calculus

Remarkably enough, the resulting calculus is almost exactly Felleisen and Chang's call-by-need calculus [27] (CF-calculus). The main difference lies in the fact that the latter features the usual destructive substitution, while ours is linear.

It is easy to convince oneself that their answer contexts correspond to our closure contexts, while the inner and outer variants stand respectively for our binding and applicative contexts. There are tiny mismatches, relatively to filling in particular. The filling condition is stated as an equality in CF-calculus, even though the two formalisms are equivalent. This is essentially a matter of rearranging the context splitting. Moreover, CF-calculus plugs closures in the reverse order compared to $\lambda_{\texttt{wls}}$.

The reduction $\beta_{\texttt{cfr}}$ can be described in our formalism in a straightforward manner.

Theorem 15 (CF-calculus revisited)**.** *CF-calculus is given by the reduction rule below.*

$$\mathcal{C}^{@}[\mathcal{C}'[\lambda x.\, \mathcal{C}_\lambda[E^v[x]]]\ \mathcal{C}[v]] \quad \to_{\texttt{cfr}} \quad \mathcal{C}^{@}[\mathcal{C}'[\mathcal{C}[\mathcal{C}_\lambda[E^v[x]][x := v]]]] \qquad \textit{if } \mathcal{C}^{@} \ni \mathcal{C}_\lambda$$

The difference in the order of closure plugging may seem irrelevant in Chang and Felleisen's framework because they use non-linear destructive substitutions and both orders are possible: an ad-hoc choice was made there. On the contrary, our design strongly guided by logic directly led us to a plugging order compatible with linear substitution.

6.6.5 From miscroscopic LHR to Ariola-Felleisen calculus

Notice that we chose to keep a macro, single-axiom, reduction close to the historical linear head reduction. It is possible though to describe linear head reduction with a more atomic reduction and the same development as we did can be achieved using the microscopic presentation from Section 6.5.4.

Any choice between rules Lift or Dig will lead to call-by-need calculi. Still we consider it is more interesting to opt for Lift since it allows us to recover known calculi. From the microscopic linear head calculus with Lift, we can apply the same three transformations as in the macroscopic case.

1. weak reduction constrain evaluation contexts to be applicative contexts up to closures:
$$E := [\cdot] \mid E\ t \mid (\lambda x.\, E)\ t$$

2. restriction to value (up to closure) substitutions, which creates new call-by-value, evaluation contexts:
$$E := \ldots \mid (\lambda x.\, E[x])\ E$$

3. sharing of closures, introducing the rule for commutation of closure contexts and which happens to be, with the simplified contexts, the usual Assoc rule:
$$(\lambda x.\, E[x])\ ((\lambda y.\, A)\ t) \to (\lambda y.\, (\lambda x.\, E[x])\ A)\ t$$

Proposition 28. *The resulting calculus is precisely AF-calculus.*

6.7 Classical Linear Head Reduction

Thanks to the intuition provided by the linear substitution, we propose in this section a classical weak linear head calculus. We hope it to be the inspiration for a more canonical classical call-by-need calculus.

We present our calculus as a variation of the $\lambda\mu$-calculus [94]. We recall the syntax and reduction of its call-by-name variant below.

$$t, u := x \mid \lambda x.\, t \mid t\ u \mid \mu\alpha.\, c$$

$$c := [\alpha]\, t$$

$$\begin{array}{lcr} (\lambda x.\, t)\ u & \to & t[x := u] \\ (\mu\alpha.\, c)\ u & \to & \mu\alpha.\, c[[\alpha]\, r := [\alpha]\, r\ u] \\ [\alpha]\, \mu\beta.\, c & \to & c[\beta := \alpha] \end{array}$$

We will only be interested in the reduction of commands in the remainder of this section. Our calculus is a direct elaboration of the aforementioned linear weak head calculus. We dedicate this section to its thorough description.

Definition 67 (Classical LHR)**.** We first define left stack contexts K by induction.

$$K := [\cdot] \mid [\alpha]\, L[\mu\beta.\, K]$$

Then we define a classical extension of left contexts and closure contexts as follows.

$$\begin{array}{rcl} \overline{\mathcal{C}} & := & [\cdot] \mid \overline{\mathcal{C}}_1[\lambda x.\, \overline{\mathcal{C}}_2]\ t \mid \overline{\mathcal{C}}_1[\mu\alpha.\, K[[\alpha]\, \overline{\mathcal{C}}_2]] \\ \overline{L} & := & [\cdot] \mid \lambda x.\, \overline{L} \mid \overline{L}\ t \mid \mu\beta.\, [\alpha]\, \overline{L} \end{array}$$

The classical linear head calculus is then defined by the following reduction.

$$[\alpha]\, \overline{L}_1[\overline{\mathcal{C}}[\lambda x.\, \overline{L}_2[x]]\ t] \quad \to_{\mathtt{clh}} \quad [\alpha]\, \overline{L}_1[\overline{\mathcal{C}}[\lambda x.\, \overline{L}_2[t]]\ t]$$

Definition 68 (Classical σ-equivalence)**.** The σ-equivalence is extended to the $\lambda\mu$-calculus with the following generators [74].

$$\begin{array}{rcll}
(\lambda x.\, \mu\alpha.\, [\beta]\, t)\ u & \cong_\sigma & \mu\alpha.\, [\beta]\, (\lambda x.\, t)\ u & \text{with } \alpha \notin u \\
[\alpha]\, (\mu\beta.\, [\gamma]\, (\mu\delta.\, c)\ u)\ t & \cong_\sigma & [\gamma]\, (\mu\delta.\, [\alpha]\, (\mu\beta.\, c)\ t)\ u & \text{with } \beta \notin u,\ \delta \notin t \\
[\alpha]\, \lambda x.\, \mu\beta.\, [\gamma]\, \lambda y.\, \mu\delta.\, c & \cong_\sigma & [\gamma]\, \lambda y.\, \mu\delta.\, [\alpha]\, \lambda x.\, \mu\beta.\, c & \\
[\alpha]\, (\mu\beta.\, [\gamma]\, \lambda x.\, \mu\delta.\, c)\ t & \cong_\sigma & [\gamma]\, \lambda x.\, \mu\delta.\, [\alpha]\, (\mu\beta.\, c)\ t & \text{with } x \notin t,\ \beta \notin t
\end{array}$$

Proposition 29 (Stability by σ)**.** *Let t, u and v be terms such that $t \cong_\sigma u \to v$, then there is w such that $t \to w \cong_\sigma v$.*

Lifting the notion of substitution sequences given in [37] from λ-calculus to $\lambda\mu$-calculus, there is a simulation theorem relating the μ-KAM with the classical LHR. To fully state it, one must start with a technical remark.

Remark 7. The `clh` reduction rule is actually abusive. Indeed, in the above reduction rule, t can be any term. This means that to ensure later capture-free substitution by preserving the Barendregt condition on terms, one has to rename the variables of t on the fly, so that the legitimate rule would rather be

$$[\alpha]\, \overline{L}_1[\overline{\mathcal{C}}[\lambda x.\, \overline{L}_2[x]]\ t] \quad \to_{\texttt{clh}} \quad [\alpha]\, \overline{L}_1[\overline{\mathcal{C}}[\lambda x.\, \overline{L}_2[\#\, t]]\ t]$$

where $\#\, t$ stands for t where bound variables have been replaced by fresh variable instances.

We need to define properly the relation between the original and the substituted terms in the above rule.

Definition 69 (One-step residual)**.** In the above rule, we say that t is the residual of $\#\, t$ in the source term.

It turns out that this definition can be extended to a reduction of arbitrary length thanks to the following lemma.

Proposition 30. *For any reduction of the form*

$$[\alpha]\, t \quad \to_{\texttt{clh}}{}^* \quad [\alpha]\, \overline{L}_1[\overline{\mathcal{C}}[\lambda x.\, \overline{L}_2[x]]\ r_0] \quad \to_{\texttt{clh}} \quad [\alpha]\, \overline{L}_1[\overline{\mathcal{C}}[\lambda x.\, \overline{L}_2[\#\, r_0]]\ r_0]$$

there exists a subterm r of t such that $r_0 \equiv \#\, r$. We call it the residual *of r_0 in t.*

Proof. By induction on the reduction. The key point is that all along the reduction, all terms on the right of an application node are subterms of the original term, up to some variable renaming. □

Definition 70 (Substitution sequence)**.** Given two terms t and t_0 s.t. $t_0 \to_{\texttt{clh}}^* t$, we define the substitution sequence of t w.r.t. t_0 as the (possibly infinite) sequence $\mathfrak{S}_{t_0}(t)$ of subterms of t_0 defined as follows, where α is a fresh stack variable.

- If $[\alpha]\, t \not\to_{\texttt{clh}}$ then $\mathfrak{S}_{t_0}(t) := \emptyset$.

- If $[\alpha]\, t \equiv [\alpha]\, \overline{L}_1[\overline{\mathcal{C}}[\lambda x.\, \overline{L}_2[x]]\ r] \to_{\mathtt{clh}} [\alpha]\, t'$ then $\mathfrak{S}_{t_0}(t) := r_0 :: \mathfrak{S}_{t_0}(t')$ where r_0 is the residual of r in t_0.

We finally pose $\mathfrak{S}(t) := \mathfrak{S}_t(t)$.

The μ-KAM naturally features a similar behaviour w.r.t. residuals.

Proposition 31. *If $\langle (t, \cdot) \mid \varepsilon \rangle \to^* \langle (t_0, \sigma) \mid \pi \rangle$ then t_0 is a subterm of t.*

Proof. By a straightforward induction over the reduction path. □

This proposition can (and actually needs to) be generalized to any source process whose stacks and closures only contain subterms of t. This leads to the definition of a similar notion of substitution sequence for the KAM.

Definition 71 (KAM substitution sequence)**.** For any term t, we define the KAM substitution sequence of a process p as the possibly infinite sequence of terms $\mathfrak{K}(p)$ defined as:

- If $p \not\to$ then $\mathfrak{K}(p) := \emptyset$.
- If $p \equiv \langle (x, \sigma) \mid \pi \rangle \to \langle (t, \tau) \mid \pi \rangle$ then $\mathfrak{K}(p) := t :: \mathfrak{K}(\langle (t, \tau) \mid \pi \rangle)$.
- Otherwise if $p \to q$ then $\mathfrak{K}(p) := \mathfrak{K}(q)$.

Finally, the KAM substitution sequence of any term t is defined as $\mathfrak{K}(t) := \mathfrak{K}(\langle (t, \cdot) \mid \varepsilon \rangle)$.

By the previous lemma, $\mathfrak{K}(t)$ is a sequence of subterms of t. We can therefore formally relate it to $\mathfrak{S}(t)$.

Proposition 32. *Let t be a term. Then $\mathfrak{K}(t)$ is a prefix of $\mathfrak{S}(t)$.*

Proof. By coinduction, for each step of $\mathfrak{S}(t)$, it is sufficient either to construct a matching step in $\mathfrak{K}(t)$ or to stop. Let us assume that

$$[\alpha]\, t \quad \equiv \quad [\alpha]\, \overline{L}_1[\overline{\mathcal{C}}[\lambda x.\, \overline{L}_2[x]]\ r_0] \quad \to_{\mathtt{clh}} \quad [\alpha]\, \overline{L}_1[\overline{\mathcal{C}}[\lambda x.\, \overline{L}_2[\#\, r_0]]\ r_0] \quad \equiv \quad [\alpha]\, t_r$$

There are now two cases, depending on the KAM reduction of $\mathfrak{K}(\langle (t, \cdot) \mid \varepsilon \rangle)$. By a simple generalization of Lemma 24, the normal form of this process in the Grab-free fragment of the KAM rules can be one of the two following form:

- either $\langle (x, \sigma + (x := (r_0, \tau)) \mid \pi \rangle$ for some σ, τ and π
- or a blocked state of the KAM for all rules

The second case can occur if there are too many λ-abstractions in the left contexts of the above reduction rule or if there is an free stack variable appearing in a command part of the left contexts. In this case $\mathfrak{K}(t) = \emptyset$, which is indeed a prefix of $\mathfrak{S}(t)$.

Otherwise, one has $\mathfrak{K}(t) = r_0 :: \mathfrak{K}(\langle (r_0, \tau) \mid \pi \rangle)$ and $\mathfrak{S}(t) = r_0 :: \mathfrak{S}_t(t_r)$. It it therefore sufficient to show that the property holds for the tail of those two sequences.

It is a noteworthy fact that, when we put t_r in the KAM, we obtain a reduction of the form

$$\langle (t_r, \cdot) \mid \varepsilon \rangle \quad \to^* \quad \langle (\#\, r_0, \sigma + (x := (r_0, \tau)) + \sigma_0) \mid \pi \rangle$$

for some σ_0, where the GRAB rule does not appear. This reduction follows indeed the very same transitions as the process made of the source term. Moreover, a careful inductive analysis of the possible transitions shows that $\sigma = \tau + \sigma_1$ for some σ_1, where the variables bound by σ_1 are not free in $\#\, r_0$. Therefore,

$$\mathfrak{K}(t_r) = \mathfrak{K}(\langle (\#\, r_0, \sigma + (x := (r_0, \tau)) + \sigma_0) \mid \pi \rangle) = \mathfrak{K}(\langle (\#\, r_0, \tau) \mid \pi \rangle)$$

because the KAM reduction is not affected by extension of closure environments with variables absent from the closure term. By applying the coinduction hypothesis, we immediately obtain than $\mathfrak{K}(t_r)$ is a prefix of $\mathfrak{S}(t_r)$.

But now, we can conclude, because $\mathfrak{K}(\langle (\#\, r_0, \tau) \mid \pi \rangle)$ and $\mathfrak{K}(\langle (r_0, \tau) \mid \pi \rangle)$ (resp. $\mathfrak{S}(t_r)$ and $\mathfrak{S}_t(t_r)$) are the same sequence up to a renaming of the bound variables coming from $\#\, r_0$ which is common to both kinds of reduction. Thus $\mathfrak{K}(\langle (r_0, \tau) \mid \pi \rangle)$ is a prefix of $\mathfrak{S}_t(t_r)$ and we are done.

□

From this lemma, one can immediately derive the following theorem.

Theorem 16. *Let $c_1 \to_{\mathtt{clh}} c_2$ where $c_1 := [\alpha]\, \overline{L}_1[\overline{\mathcal{C}}[\lambda x.\, \overline{L}_2[x]]\ t]$, then the substitution sequence of process c_1 is either empty or of the form $t :: \ell$ where ℓ is the substitution sequence of process c_2.*

6.8 Classical by Need

To extend our classical LHR calculus to a fully-fledged call-by-need calculus, we follow the same three-step path that lead us from LHR to call-by-need.

6.8.1 Weak classical LHR

The most delicate point is actually the introduction of weak reduction. Indeed we need to be able to tell when we did not go through too many λ-abstractions.

In a classical setting, the actual applicative context of a variable may be strictly larger than it seems, because in commands of the form $[\alpha]\, t$, the α variable may be bound to a stack featuring supplementary applications. That is, we need to keep track of the supernumerary abstractions applied to a given stack variable.

To this end, we introduce dual stack contexts K_w which correspond to fragments of stacks which have been fed with enough applications to reduce in a weak setting.

$$K_w := [\cdot] \mid [\alpha]\, \mathcal{C}_\lambda \circ \mathcal{C}^{@}[\mu\beta.\, K_w]$$

All K_w contexts are not legal, though. We use an environment Γ to record the number of supplementary applications bound to each stack variable, which is generated by the inductive grammar below.

$$\Gamma := \cdot \mid \Gamma, \alpha : n$$

The filling relation from the intuitionistic case is adapted as follows.

Definition 72. We define the classical filling $\Gamma \vdash K_w$ relation as follows.

$$\frac{}{\Gamma \vdash [\cdot]}$$

$$\frac{(\alpha, n) \in \Gamma \qquad |\mathcal{C}_\lambda|_\lambda \leq n \qquad \Gamma, \beta : n - |\mathcal{C}_\lambda|_\lambda + \left|\mathcal{C}^{@}\right|_{@} \vdash K_w}{\Gamma \vdash [\alpha]\, \mathcal{C}_\lambda \circ \mathcal{C}^{@}[\mu\beta.\, K_w]}$$

Contrarily to the classical-by-need case, this relation is uniquely defined when it exists.

There is a design choice here. One could indeed extend closure contexts so as to integrate K_w contexts, i.e.

$$\mathcal{C} := \ldots \mid \mathcal{C}_1[\mu\alpha.\, K_w[[\alpha]\, \mathcal{C}_2]]$$

as we did directly in the classical LHR. This works, but leads to a presentation which is a bit cluttered, because one still needs to perform μ reductions in a call-by-value setting, which we will be doing later on. We rather choose to stick to intuitionistic closure contexts, and integrate this property in the μ reductions.

Definition 73 (Weak classical LHR)**.** The weak classical LHR calculus is defined by the two following reduction rules, where $C := \mathcal{C}_\lambda \circ \mathcal{C}^{@'}$ and $K_0 := K_w[[\alpha]\, C]$.

$$\frac{\mathcal{C}^{@} \ni \mathcal{C}_\lambda \qquad \vdash K_0[\mu\_.\, [\cdot]]}{K_0[\mathcal{C}^{@}[\mathcal{C}_1[\lambda x.\, \mathcal{C}_\lambda \circ \mathcal{C}^{w}[x]]\ t]] \to K_0[\mathcal{C}^{@}[\mathcal{C}_1[\lambda x.\, \mathcal{C}_\lambda \circ \mathcal{C}^{w}[t]]\ t]]}$$

$$\frac{\vdash K_0[\mu\beta.\, K'_w]}{K_0[\mu\beta.\, K'_w[[\beta]\, t]] \to K_0[\mu\beta.\, K'_w[[\alpha]\, C[t]]]}$$

6.8.2 Call-by-value weak classical LHR

We simply adapt the notions of the previous section to a call-by-value-like setting. This leads to the definitions of stack fragments K_v of the following form.

$$K_v := [\cdot] \mid [\alpha]\, \mathcal{C}_\lambda \circ E_v[\mu\beta.\, K_v]$$

A context K_v is essentially an E_v context interspersed with μ-binders and commands which may start with a certain number of supplementary λ-abstractions. Indeed, the α variable in the above definition may be bound to n applications, so that the leading $\mathcal{C}_\lambda$ can be made of at most n λ-abstractions. The filling relation needs to be upgraded to acknowledge this fact. Instead of relating an applicative context with a binding context, it uses an environment Γ to record the number of supplementary applications bound to each stack variable, which is generated by the inductive grammar below.

$$\Gamma := \cdot \mid \Gamma, \alpha : n$$

Definition 74. The call-by-value weak classical LHR calculus is defined by the two following reduction rules, where $C := \mathcal{C}_\lambda \circ E'_v$ and $K_0 := K_v[[\alpha]\, C]$.

$$\frac{\vdash K_0[\mu\_.\,[\cdot]]}{K_0[\mathcal{C}_1[\lambda x.\, E_v[x]]\ \mathcal{C}_2[v]] \to K_0[\mathcal{C}_1[\lambda x.\, E_v[\mathcal{C}_2[v]]]\ \mathcal{C}_2[v]]}$$

$$\frac{\vdash K_0[\mu\beta.\, K'_v]}{K_0[\mu\beta.\, K'_v[[\beta]\, t]] \to K_0[\mu\beta.\, K'_v[[\alpha]\, C[t]]]}$$

6.8.3 Call-by-Need in a Classical Calculus

The classical-by-need calculus is essentially the same as the call-by-value weak classical LHR. It is indeed sufficient to apply the sharing in the β-reduction rule. All the other definitions are unchanged. This results in the following rules.

Definition 75. We define the classical filling $\Gamma \vdash K_v$ relation as follows.

$$\frac{}{\Gamma \vdash [\cdot]} \qquad \frac{(\alpha, n) \in \Gamma \qquad |\mathcal{C}_\lambda|_\lambda \leq n \qquad \Gamma, \beta : \left|\mathcal{C}^{@}\right|_{@} \vdash K_v}{\Gamma \vdash [\alpha]\, \mathcal{C}_\lambda \circ E_v \circ \mathcal{C}^{@}[\mu\beta.\, K_v]}$$

$$\frac{(\alpha, n) \in \Gamma \qquad |\mathcal{C}_\lambda|_\lambda \leq n \qquad \Gamma, \beta : n - |\mathcal{C}_\lambda|_\lambda + \left|\mathcal{C}^{@}\right|_{@} \vdash K_v}{\Gamma \vdash [\alpha]\, \mathcal{C}_\lambda \circ \mathcal{C}^{@}[\mu\beta.\, K_v]}$$

In the second and third rule, we take advantage of the fact that $\mathcal{C}^{@}$ contexts are actually a particular subcase of the E_v grammar. Note that this relation, when derivable, is not uniquely defined: there may be various ways to decompose the E_v context in the second rule.

Definition 76 (Classical-by-need)**.** The classical-by-need `cwls`-calculus is defined by the two following reduction rules, where $C := \mathcal{C}_\lambda \circ E'_v$ and $K_0 := K_v[[\alpha]\, C]$.

$$\frac{\vdash K_0[\mu\_.\,[\cdot]]}{K_0[\mathcal{C}_1[\lambda x.\, E_v[x]]\ \mathcal{C}_2[v]] \to K_0[\mathcal{C}_2[\mathcal{C}_1[\lambda x.\, E_v[v]]\ v]]}$$

$$\frac{\vdash K_0[\mu\beta.\, K'_v]}{K_0[\mu\beta.\, K'_v[[\beta]\, t]] \to K_0[\mu\beta.\, K'_v[[\alpha]\, C[t]]]}$$

6.8.4 Comparison with existing works

We now turn to the comparison of our calculus with another classical-by-need calculus [12]. We recall here the description of this calculus.

Definition 77 (Ariola-Herbelin-Saurin calculus)**.** AHS reduction for the $\lambda\mu$-calculus is defined below.

$$
\begin{array}{lcl}
 & & (\lambda x.\, t)\ u\ r \ \to\ (\lambda x.\, t\ r)\ u \\
v & := & x \mid \lambda x.\, t \qquad (\lambda x.\, t)\ v \ \to\ t[x := v] \\
n & := & t\ u \qquad (\lambda z.\, C[z])\ ((\lambda x.\, t)\ u) \ \to\ (\lambda x.\, (\lambda z.\, C[z])\ t)\ u \\
E & := & [\cdot] \mid E\ t \qquad (\mu\alpha.\, c)\ t \ \to\ \mu\alpha.\, c[[\alpha]\, r := [\alpha]\, r\ t] \\
C & := & E \mid (\lambda z.\, C)\ n \mid \qquad (\lambda x.\, C[x])\ (\mu\alpha.\, c) \ \to\ \mu\alpha.\, c[[\alpha]\, r := [\alpha]\, (\lambda x.\, C[x])\ r] \\
 & & (\lambda x.\, C[x])\ (E\ t) \qquad (\lambda x.\, \mu\alpha.\, [\beta]\, t)\ n \ \to\ \mu\alpha.\, [\beta]\, (\lambda x.\, t)\ n \\
 & & [\alpha]\, \mu\beta.\, c \ \to\ c[\beta := \alpha]
\end{array}
$$

Because of the extensional proximity of this calculus with our classical-by-need calculus, we make the following conjecture.

Conjecture 1. *For any command c, there exists an infinite standard reduction in AHS-calculus starting from c iff there exists an infinite reduction starting from c in* `cwls`*-calculus.*

Proving formally the above conjecture reveals to be tricky because AHS is built on destructive substitution which additionally is plain β_v. The reason for such a presentation of AHS is to be found in its sequent calculus origin [12]. As we did for the comparison with Chang-Felleisen calculus, we will consider a variant of AHS with a deref rule à la Ariola-Felleisen and prove that the conjecture holds in this case.

We will actually see that this result is easy to show for the considered variant of the above AHS-calculus where the derefencing rule has been restricted to needed context, instead of direct substitution of values bound by let bindings.

We now consider a slightly modified version of the previous calculus from [12]. AHS'-calculus consists in AHS-calculus where the beta reduction has been replaced by a deref rule à la Ariola Felleisen (where variables are not values) and the notion of evaluation context has been adapted accordingly. This calculus has been first described, in sequent style, in [13].

Definition 78 (Ariola-Herbelin-Saurin modified calculus)**.** AHS' reduction for the $\lambda\mu$-calculus is defined below.

$$
\begin{array}{lcl}
 & & (\lambda x.\, t)\ u\ r \ \to\ (\lambda x.\, t\ r)\ u \\
v & := & \lambda x.\, t \qquad (\lambda x.\, C[x])\ v \ \to\ (\lambda x.\, C[v])\ v \\
n & := & x \mid t\ u \mid \mu\alpha.\, c \qquad (\lambda z.\, C[z])\ ((\lambda x.\, t)\ u) \ \to\ (\lambda x.\, (\lambda z.\, C[z])\ t)\ u \\
E & := & [\cdot] \mid E\ t \qquad (\mu\alpha.\, c)\ t \ \to\ \mu\alpha.\, c[[\alpha]\, r := [\alpha]\, r\ t] \\
C & := & E \mid (\lambda z.\, C)\ t \mid \qquad (\lambda x.\, C[x])\ (\mu\alpha.\, c) \ \to\ \mu\alpha.\, c[[\alpha]\, r := [\alpha]\, (\lambda x.\, C[x])\ r] \\
 & & (\lambda x.\, C[x])\ E \qquad (\lambda x.\, \mu\alpha.\, [\beta]\, t)\ n \ \to\ \mu\alpha.\, [\beta]\, (\lambda x.\, t)\ n \\
 & & [\alpha]\, \mu\beta.\, c \ \to\ c[\beta := \alpha]
\end{array}
$$

Theorem 17. *For any command c, there exists an infinite standard reduction in AHS'-calculus starting from c iff there exists an infinite reduction starting from c in* `cwls`*-calculus.*

Proof. We show this by giving the sketch of a pair of simulation theorems. We will separate AHS' reduction rules in three groups:

- The structural rules (S) which are made of the Lift and Assoc rules, together with the rule $(\lambda x.\, \mu\alpha.\, [\beta]\, t)\ n \to \mu\alpha.\, [\beta]\, (\lambda x.\, t)\ n$.
- The performing rules (P) which is only the derefencing rule.
- The classical rules (C) which are the three remaining rules.

Transforming reductions in `cwls`-calculus into AHS' is straightforward. First, assuming a closure stack fragment K_v, one can see that AHS' will normalize it into a delimited C context in the following way. For each splice of K_v of the form $[\alpha]\, \mathcal{C}_\lambda \circ E_v[\mu\beta.\, [\cdot]]$, the $\mathcal{C}_\lambda$ part is actually fed with the corresponding arguments when this splice comes in standard position, so that the rules (S) will flatten it to an answer context. The E_v context will be simplified likewise, until the (S) rules cannot be applied anymore. According to the form of K_v, either the reduction stops (if there is no remaining splice) or it performs a certain number of (C) rules, until which the normalization procedure recursively applies. The filling condition on K_v ensures us that the (C) rule will provide the missing applications to the next $\mathcal{C}_\lambda$ contexts from each remaining splice. A dereferencing cannot occur at this point because while there are remaining splices, the current needed context cannot contain variables, as the splices all have a μ binder in needed position. Note that the resulting normalized context is still a closure stack fragment. By a simple size argument, this normalization procedure must terminate, so that we will consider it transparent for the simulation.

Now, assume that

$$K_0[\mathcal{C}_1[\lambda x.\, E_v[x]]\ \mathcal{C}_2[v]] \to K_0[\mathcal{C}_2[\mathcal{C}_1[\lambda x.\, E_v[v]]\ v]]$$

with the associated conditions for this rule. The normalization procedure of the K_0 transforms it into an needed context $[\alpha]\, C$, and then the (S) rules apply to $\mathcal{C}_1$ and $\mathcal{C}_2$. This effectively transforms them into answer contexts, so that a derefencing rule can occur after the E_v has been flattened as well. The important thing to observe is that the same normalization steps apply to the reduct $K_0[\mathcal{C}_2[\mathcal{C}_1[\lambda x.\, E_v[v]]\ v]]$ so that each step from `cwls` is going to be matched by a growing but finite quantity of normalization steps followed by a derefencing.

Likewise, assume

$$K_0[\mu\beta.\, K'_v[[\beta]\, t]] \to K_0[\mu\beta.\, K'_v[[\alpha]\, C[t]]]$$

with the associated conditions. Such a rule is actually directly handled by the normalization procedure for the K_0 prefix.

We turn to the simulation of the AHS' reduction by the `cwls`-calculus. First, the standard reduction contexts are a degenerated case of K_v contexts with one splice and flattened closure contexts, which allows to easily transfer rules from the source to the target. We actually match each class of reduction (S), (P) and (C) to a given behaviour in the target calculus.

- The (S) rules are transparent for the `clh`-calculus, because they are natively handled by closure contexts. So a (S)-reduction does not give rise to a `clh`-reduction.

- A group of (C) rules can be matched by an arbitrary number of reductions, including none. This depends on the way the corresponding stack variable is used.
- The (P) rule is conversely matched by exactly one rule in the `clh` reduction.

The trick is to use the fact that $(C + S)$ is normalizing, as we already did in the previous case. Moreover, such reductions do not change the possibility to perform a derefencing in the corresponding `clh` term. So we actually consider groups of reductions $(C + S)^*, P$ in the source calculus. This is always possible to decompose a sequence of AHS' reductions as such thanks to the normalization of $(C + S)$. It it then easy to witness that the S part will have no effect, each C reduction will be matched by a finite number of context reductions in `clh`, and that the final (P) will correspond to exactly one derefencing reduction in `clh`. □

7 Dialectica: a historical presentation

> — Hé, les copains, un peu de dialectique.
> — C'est la seule façon de casser des briques, camarades.
>
> *René Viénet about the functional interpretation.*

The Dialectica transformation, also known as the functional interpretation, was introduced by Gödel in the eponymous journal in 1958 [51], although he had been designing it since the 30's [15]. It turns out it was a tentative workaround to the incompleteness theorem, then perceived as great catastrophe. As classical logic could not be considered a trustful tool to justify itself anymore, one had to solve the problem of foundations by using *constructive* means.

This paradigmatic shift can be considered as one of the foundational stones of modern proof-theory.

Gödel's historical presentation of the Dialectica translation uses intuitionistic arithmetic as the source system, and targets higher-order intuitionistic arithmetic. For the sake of completeness, we recall its slightly modernized presentation in the following section. It is similar to the presentation given by Avigad and Feferman [15], although we already take from the start a more proof-theoretical standpoint.

7.1 Intuitionistic arithmetic

As the translation is defined in intuitionistic arithmetic, we will recall here how we define it. This is a first-order logic whose domain covers the integers, and whose unique predicate is the equality. The complete set of data describing the first-order part was defined at Section 2.2.3, so we only recall here the relevant extensions.

Definition 79 (Terms and formulae of intuitionistic arithmetic)**.** Terms of arithmetic are terms built upon the usual signature:

$$t, u ::= x \mid \mathbf{0} \mid \mathbf{S}\, t \mid t + u \mid t \times u$$

The formulae of intuitionistic arithmetic are the one from first-order logic where the only predicate is a binary symbol for equality. Expanding the definition gives the following inductive grammar.

$$A, B ::= t = u \mid A \to B \mid \top \mid A \wedge B \mid \bot \mid A \vee B \mid \forall x.\, A \mid \exists x.\, A$$

The deduction rules of **HA** are the rules of first-order logic, with supplementary axioms allowing to express the properties of integers.

Definition 80 (Rules of **HA**)**.** The rules of **HA** are the usual rules for intuitionistic predicate logic from Section 2.2, enriched with the following rules allowing to deal with properties of equality and integers. We assume the usual freshness conditions when dealing with introduced variables.

$$\frac{}{\Gamma \vdash t = t} \qquad \frac{}{\Gamma \vdash 0 \neq \mathbf{S}\, t} \qquad \frac{\Gamma \vdash \mathbf{S}\, t = \mathbf{S}\, u}{\Gamma \vdash t = u}$$

$$\frac{\Gamma \vdash t = u \qquad \Gamma \vdash A[x := u]}{\Gamma \vdash A[x := t]} \qquad \frac{\Gamma \vdash A[x := \mathbf{0}] \qquad \Gamma, A \vdash A[x := \mathbf{S}\, x]}{\Gamma \vdash A[x := t]}$$

$$\frac{}{\Gamma \vdash 0 + u = u} \qquad \frac{}{\Gamma \vdash \mathbf{S}\, t + u = \mathbf{S}\,(t + u)}$$

$$\frac{}{\Gamma \vdash 0 \times u = 0} \qquad \frac{}{\Gamma \vdash \mathbf{S}\, t \times u = u + (t \times u)}$$

7.2 System T

Traditionally, the Dialectica interpretation is defined through the use of the so-called System T. In modern words, System T is a higher-order language, based on the simply-typed λ-calculus, but extended with inductively defined natural numbers.

Definition 81 (System T)**.** The types of system T are all simple types with the natural numbers as sole base type.

$$\tau, \sigma ::= \mathbb{N} \mid \tau \to \sigma$$

Note that in this chapter, we will use Greek letters σ, τ to represent System T types rather than the letters A, B we were used to, to emphasize the difference between System T types and **HA** formulae.

The terms are the simply-typed λ-calculus together with constructors and recursor for the integers.

$$t, u ::= x \mid t\ u \mid \lambda x.\, t \mid \mathbf{0} \mid \mathbf{S}\, t \mid \mathbf{rec}\, t\, f\, g$$

The typing rules are taken from the simply-typed λ-calculus, with the following supplementary rules for the integer-related constants.

$$\frac{}{\Gamma \vdash \mathbf{0} : \mathbb{N}} \qquad \frac{\Gamma \vdash t : \mathbb{N}}{\Gamma \vdash \mathbf{S}\, t : \mathbb{N}} \qquad \frac{\Gamma \vdash t : \mathbb{N} \qquad \Gamma \vdash f : \tau \qquad \Gamma \vdash g : \mathbb{N} \to \tau \to \tau}{\Gamma \vdash \mathbf{rec}\, t\, f\, g : \tau}$$

Likewise, we extend the usual β-reduction with the following reductions.

$$\mathbf{rec}\, \mathbf{0}\, f\, g \to_\beta f \qquad \mathbf{rec}\,(\mathbf{S}\, t)\, f\, g \to_\beta g\, t\,(\mathbf{rec}\, t\, f\, g)$$

7.3 HA $+ T$

We now need to allow our meta-logical system to reason about System T terms. To this end we define the **HA** $+ T$ system. This is a variant of **HA** where higher-order functions are allowed in the term syntax. Rather than plain old first-order logic, it is actually closer to a dependently typed logic, using conversion instead of axiomatic equalities.

Definition 82 (**HA** $+ T$)**.** We define the system **HA** $+ T$ as **HA** where terms can now range over the terms of System T, and extended with the rewriting rules generated by β-equivalence. A formal definition of this system is presented below.

Because free term variables may now have types other than $\mathbb{N}$, we need to keep track of their type in a dedicated environment. The sequents are therefore of the following form

$$\Sigma \mid \Gamma \vdash A$$

where Γ is a list of formulae and Σ a System T typing environment.

$$\begin{aligned} \Gamma &:= \cdot \mid \Gamma, A \\ \Sigma &:= \cdot \mid \Sigma, x : \sigma \end{aligned}$$

The derivation rules are extended accordingly. Note that we must ensure that there are no ill-formed System T terms in the formulae, so we enforce typing wherever needed. This forces us to define a well-foundedness property on environments, akin to the one from dependent types.

$$\frac{}{\Sigma \vdash_{\text{wf}} \cdot} \qquad \frac{\Sigma \vdash_{\text{wf}} \Gamma \qquad \Sigma \vdash_{\text{wf}} A}{\Sigma \vdash_{\text{wf}} \Gamma, A} \qquad \frac{\Sigma \vdash_{\text{wf}} A \qquad \Sigma \vdash_{\text{wf}} B}{\Sigma \vdash_{\text{wf}} A \to B}$$

$$\frac{\Sigma \vdash t : \mathbb{N} \qquad \Sigma \vdash u : \mathbb{N}}{\Sigma \vdash_{\text{wf}} t = u} \qquad \frac{\Sigma, x : \sigma \vdash_{\text{wf}} A}{\Sigma \vdash_{\text{wf}} \forall x : \sigma.\, A} \qquad \frac{\Sigma, x : \sigma \vdash_{\text{wf}} A}{\Sigma \vdash_{\text{wf}} \exists x : \sigma.\, A}$$

$$\frac{\Sigma \vdash_{\text{wf}} A \qquad \Sigma \vdash_{\text{wf}} B}{\Sigma \vdash_{\text{wf}} A \wedge B} \qquad \frac{}{\Sigma \vdash_{\text{wf}} \top} \qquad \frac{\Sigma \vdash_{\text{wf}} A \qquad \Sigma \vdash_{\text{wf}} B}{\Sigma \vdash_{\text{wf}} A \vee B} \qquad \frac{}{\Sigma \vdash_{\text{wf}} \bot}$$

The whole system of derivation rules is defined below.

Propositional logic

$$\frac{\Sigma \mid \Gamma \vdash B \qquad A \equiv_\beta B \qquad \Sigma \vdash_{\text{wf}} A}{\Sigma \mid \Gamma \vdash A}$$

$$\frac{\Sigma \vdash_{\text{wf}} \Gamma, A}{\Sigma \mid \Gamma, A \vdash A} \qquad \frac{\Sigma \mid \Gamma, A \vdash B}{\Sigma \mid \Gamma \vdash A \to B} \qquad \frac{\Sigma \mid \Gamma \vdash A \to B \qquad \Sigma \mid \Gamma \vdash A}{\Sigma \mid \Gamma \vdash B}$$

$$\frac{\Sigma \vdash_{\text{wf}} \Gamma}{\Sigma \mid \Gamma \vdash \top} \qquad \frac{\Sigma \mid \Gamma \vdash A \wedge B}{\Sigma \mid \Gamma \vdash A} \qquad \frac{\Sigma \mid \Gamma \vdash A \wedge B}{\Sigma \mid \Gamma \vdash B} \qquad \frac{\Sigma \mid \Gamma \vdash A \qquad \Sigma \mid \Gamma \vdash B}{\Sigma \mid \Gamma \vdash A \wedge B}$$

$$\frac{\Sigma \mid \Gamma \vdash \bot \qquad \Sigma \vdash_{\text{wf}} A}{\Sigma \mid \Gamma \vdash A} \qquad \frac{\Sigma \mid \Gamma \vdash A \qquad \Sigma \vdash_{\text{wf}} B}{\Sigma \mid \Gamma \vdash A \vee B} \qquad \frac{\Sigma \mid \Gamma \vdash B \qquad \Sigma \vdash_{\text{wf}} A}{\Sigma \mid \Gamma \vdash A \vee B}$$

$$\frac{\Sigma \mid \Gamma \vdash A \vee B \qquad \Sigma \mid \Gamma, A \vdash C \qquad \Sigma \mid \Gamma, B \vdash C}{\Sigma \mid \Gamma \vdash C}$$

First order

$$\frac{\Sigma, x : \sigma \mid \Gamma \vdash A}{\Sigma \mid \Gamma \vdash \forall x : \sigma.\, A} \qquad \frac{\Sigma \vdash t : \sigma \qquad \Sigma \mid \Gamma \vdash \forall x : \sigma.\, A}{\Sigma \mid \Gamma \vdash A[x := t]}$$

$$\frac{\Sigma \vdash t : \sigma \qquad \Sigma \mid \Gamma \vdash A[x := t]}{\Sigma \mid \Gamma \vdash \exists x : \sigma.\, A} \qquad \frac{\Sigma \mid \Gamma \vdash \exists x : \sigma.\, A \qquad \Sigma, x : \sigma \mid \Gamma, A \vdash C}{\Sigma \mid \Gamma \vdash C}$$

Arithmetic

$$\frac{\Sigma \vdash_{\text{wf}} \Gamma \qquad \Sigma \vdash t : \mathbb{N}}{\Sigma \mid \Gamma \vdash t = t} \qquad \frac{\Sigma \mid \Gamma \vdash t = u \qquad \Sigma \mid \Gamma \vdash A[x := u]}{\Sigma \mid \Gamma \vdash A[x := t]}$$

$$\frac{t \equiv_\beta u \qquad \Sigma \vdash_{\text{wf}} A[x := t] \qquad \Sigma \mid \Gamma \vdash A[x := u]}{\Sigma \mid \Gamma \vdash A[x := t]}$$

$$\frac{\Sigma \vdash_{\text{wf}} \Gamma, 0 \neq \mathbf{S}\, t}{\Sigma \mid \Gamma \vdash 0 \neq \mathbf{S}\, t} \qquad \frac{\Sigma \mid \Gamma \vdash t = u}{\Sigma \mid \Gamma \vdash \mathbf{S}\, t = \mathbf{S}\, u}$$

$$\frac{\Sigma \vdash t : \mathbb{N} \qquad \Sigma \mid \Gamma \vdash A[x := \mathbf{0}] \qquad \Sigma, x : \mathbb{N} \mid \Gamma, A \vdash A[x := \mathbf{S}\, x]}{\Sigma \mid \Gamma \vdash A[x := t]}$$

We can see **HA** as a subtheory of $\mathbf{HA} + T$, because System T already encodes natural numbers. We can define the previously axiomatized arithmetical operations by proper λ-terms based on the integer recursor. Typically, we define a translation $[\![\cdot]\!]$ from **HA** terms to System T terms by induction as:

$$\begin{aligned}
[\![x]\!] &:= x \\
[\![\mathbf{0}]\!] &:= \mathbf{0} \\
[\![\mathbf{S}\, t]\!] &:= \mathbf{S}\, [\![t]\!] \\
[\![t + u]\!] &:= \mathbf{rec}\ [\![t]\!]\ [\![u]\!]\ (\lambda p\, r.\, \mathbf{S}\, r) \\
[\![t \times u]\!] &:= \mathbf{rec}\ [\![t]\!]\ \mathbf{0}\ (\lambda p\, r.\, [\![u]\!] + r)
\end{aligned}$$

All **HA** terms can then be casted as natural numbers in $\mathbf{HA} + T$. We will often use the interpretation function $[\![\cdot]\!]$ implicitly.

Proposition 33. *For any* **HA** *term* t *with free variables* $\vec{n}$, $\vec{n} : \mathbb{N} \vdash [\![t]\!] : \mathbb{N}$.

This allows to lift propositions from **HA** to $\mathbf{HA} + T$. To any formula A from **HA**, we associate a formula $[\![A]\!]$ in $\mathbf{HA} + T$ defined by induction, which is A with all terms t replaced by $[\![t]\!]$ and all quantifications $\forall x.\, A$ (resp. $\exists x.\, A$) relativized to $\forall x : \mathbb{N}.\, A$ (resp. $\exists x : \mathbb{N}.\, A$). Once again, we will omit the $[\![\cdot]\!]$ translation.

Proposition 34. *For any closed formula* A, *if* **HA** *proves* A *then* $\mathbf{HA} + T$ *proves* A.

Proof. By induction on the proof of A. We have to generalize the statement a bit, to handle free variables and assumptions. The resulting lemma is given as: for all **HA**-formula A and hypotheses Γ with free variables $\vec{x}$, if $\Gamma \vdash A$ then $\vec{x} : \mathbb{N} \mid \Gamma \vdash A$.

- Derivations of the propositional and first-order fragments are transported as-is, by taking care to extend the term environment when introducing fresh variables.
- We only have to check that the arithmetic fragment is preserved. Most rules are straightforward, as they only require the additional typing of **HA** terms which comes for free, as any **HA** term t admits a typing derivation $\vec{x} : \mathbb{N} \vdash t : \mathbb{N}$ when $\vec{x}$ ranges over the free variables of t.

  The only rules that are somehow new are the axioms for the rewriting of $\times$ and $+$. They are proved by a simple conversion followed by reflexivity of equality in $\mathbf{HA} + T$. We actually chose these axioms to agree on the primitive recursive definition of the arithmetical operations.

□

From now on, we will be implicitly casting terms from **HA** into $\mathbf{HA} + T$ according to the translation scheme defined above.

Remark 8. For the sake of readability, we will not be using this system as such to reason about terms, but rather formulate our proofs in a more human-friendly metatheory whose proofs can still be formalized in $\mathbf{HA} + T$. The main goal of $\mathbf{HA} + T$ is to delimit the expressive power we need, even though it will not be explicit in the demonstrations.

7.4 Gödel's motivations

The goal of Gödel's Dialectica translation was to realize strictly more than purely intuitionistic arithmetic, even though all of its definitions and proofs are made intuitionistically. There would not be much logical interest otherwise, as **HA** is the perfect candidate for a computational intuitionistic arithmetic.

In addition to **HA**, Dialectica realizes indeed two semi-classical principles, respectively known as *Markov's principle* and *the independence of premise*. The exact statements of those formulae are given below.

$$\frac{}{\neg\forall x.\, \neg A \to \exists x.\, A} \qquad \frac{}{(A \to \exists x.\, B) \to \exists x.\, (A \to B)}$$

Such axioms are independent of pure intuitionistic logic, and of **HA** in particular. Their constructivist acceptance depends on the formula A considered. In general, one requires A to be decidable in the case of Markov's principle, while A should be somehow computationally irrelevant in the independence of premise. Let us justify them under these assumptions informally.

- If A is decidable, Markov's principle can be algorithmically implemented as follows: start an unbounded loop from $n = 0$ and incrementally check whether $A[x := n]$ holds. This is possible because A is decidable. If ever some index satisfies the formula, stop and return it as the result for the existential. Thanks to the assumption $\neg\forall x.\, \neg A$, this loop must eventually terminate even though there is no definite way to know when. The termination argument is classical but it does not interfere with the operational behaviour of the algorithm.

- Assume we have a good notion of computationally irrelevant. One can imagine it as purely logical content[1], or in a weaker setting, a subset of types which are not observable, like purely negative types. Then if A is such a type, the content of the argument fed to the $A \to \exists x.\, B$ function does not matter to build the witness. One may pass it a default value. To ensure consistency, one should assume that the A type is inhabited. In our arithmetical system, we can safely produce a default integer when A is not inhabited, and ask the programmer for a subsequent proof of A, which would be impossible in this case. This actually justifies the above presentation of this axiom.

Some systems precisely implement the behaviours we just described. See for instance Herbelin's work [55], where the *return* part of Markov's principle is implemented thanks to an exception.

It is known that the expressive power of the Dialectica translation is essentially equivalent to Heyting's arithmetic enriched with these two principles [15]. At the end of this chapter, we will be review how it interprets these principles, and in particular what computational content they hide.

[1] The knowledgeable reader may think of Coq's `Prop` values, or HoTT truncations.

7.5 Gödel's Dialectica

The original presentation of the Dialectica interpretation is a bit abstruse to our modern eyes. Indeed, System T is lacking some of the structure necessary to an elegant presentation, and Gödel had to resort to various tricks to make the translation work. The global picture is the following: take a proof $\vdash A$ of **HA**, and translate it into a statement of the form:

$$\exists \vec{u} : \vec{\tau}. \forall \vec{x} : \vec{\sigma}. A^D$$

where $\vec{u}$ and $\vec{x}$ range over the terms of System T, and A^D is defined by induction over A.

A careful scrutiny of this formulation reveals that, in our modern phraseology, we would call this a realizability interpretation: from a proof, extract a computational interpretation in a given programming language, here System T. In order not to disrupt the mental workflow of a reader already acquainted with the various flavours of realizability, we will try to stick to this intuition and present the historical transformation according to our modern understanding.

The translated formula can be seen as a sort of a game, as in game semantics: the existentially quantified terms $\vec{u}$ are going to be seen as the proponents (or witnesses) of A, the universally quantified terms $\vec{x}$ as opponents (or counters) of A, and the translated formula A^D as the rules of the game on A. The goal of the witnesses is to defeat any possible opponent according to the rules of A^D.

7.5.1 Sequences

First, as Dialectica works with sequences a lot, we will define some useful constructions to manipulate them.

Notation 3. Assuming some objects $x_1, \ldots, x_n$, we will write $\vec{x}$ to represent the sequence of such objects when the indexing is clear. We will freely write $\vec{x}\ ;\ \vec{y}$ for the concatenation of two such sequences, x for the singleton sequence and $\emptyset$ for the empty sequence. We will also write $|\vec{x}|$ for the length of the sequence $\vec{x}$.

Suppose a sequence of System T types $\vec{\sigma} := \sigma_1\ ;\ \ldots\ ;\ \sigma_n$, and a type τ, we will use the following notations:

$$\begin{array}{lcll} \vec{\sigma} \to \tau & := & \sigma_1 \to \ldots \to \sigma_n \to \tau & \text{is a type} \\ \tau \to \vec{\sigma} & := & \tau \to \sigma_1\ ;\ \ldots\ ;\ \tau \to \sigma_n & \text{is a sequence of types} \end{array}$$

This notation may lead to confusion at first sight. Indeed, one may think that the notation $\vec{\sigma} \to \vec{\tau}$ is ambiguous. Actually, it does not matter how one unfolds the notation. Either way we recover the following sequence of type, where $m = |\vec{\sigma}|$ and $n = |\vec{\tau}|$:

$$\vec{\sigma} \to \vec{\tau} := \sigma_1 \to \ldots \to \sigma_m \to \tau_1\ ;\ \ldots\ ;\ \sigma_1 \to \ldots \to \sigma_m \to \tau_n$$

We will also use the same notation for the $\mathbf{HA} + T$ implication arrow.

Likewise, we extend term application to the case of sequences of terms, with the interpretation defined as the unique way that makes sense w.r.t. the typing defined above, i.e. if t is a term and $\vec{u}$ is a sequence of terms, then:

$$\begin{array}{lcll} t\ \vec{u} & := & t\ u_1\ \dots\ u_n & \text{is a term} \\ \vec{u}\ t & := & u_1\ t\ ;\ \dots\ ;\ u_n\ t & \text{is a sequence of terms} \end{array}$$

As in the case of sequences of types, the notation $\vec{t}\ \vec{u}$ is not ambiguous, and represents the sequence of terms

$$\vec{t}\ \vec{u} := t_1\ u_1\ \dots\ u_m\ ;\ \dots\ ;\ t_n\ u_1\ \dots\ u_m$$

By duality, the very same mechanism goes on for λ-abstraction.

$$\begin{array}{lcll} \lambda\vec{x}.\, t & := & \lambda x_1 \dots x_n.\, t & \text{is a term} \\ \lambda x.\, \vec{t} & := & \lambda x.\, t_1\ ;\ \dots\ ;\ \lambda x.\, t_n & \text{is a sequence of terms} \end{array}$$

If in addition $\vec{u}$ has the same length as $\vec{\sigma}$, we will write $\Gamma \vdash \vec{u} : \vec{\sigma}$ if each term is typable with the corresponding type componentwise, and similarly $\vec{x} : \vec{\sigma}$ will represent the sequence $x_1 : \sigma_1\ ;\ \dots\ ;\ x_n : \sigma_n$ when the length of $\vec{x}$ and $\vec{\sigma}$ coincide.

Remark 9. If we look at the particular case of empty sequences, we have the following equalities.

$$\begin{array}{lcllcl} \emptyset \to \tau & \equiv & \tau & \sigma \to \emptyset & \equiv & \emptyset \\ t\ \emptyset & \equiv & t & \emptyset\ u & \equiv & \emptyset \\ \lambda\emptyset.\, t & \equiv & t & \lambda x.\, \emptyset & \equiv & \emptyset \end{array}$$

Remarkably enough, typing can be naturally lifted to sequences without any kind of problem, as attested by the following lemma.

Proposition 35. *The following typing rules are admissible when the notations make sense, i.e. when* $|\vec{x}| = |\vec{u}| = |\vec{\sigma}|$ *and* $\left|\vec{t}\right| = |\vec{\tau}|$.

$$\frac{\Gamma, \vec{x} : \vec{\sigma} \vdash \vec{t} : \vec{\tau}}{\Gamma \vdash \lambda\vec{x}.\, \vec{t} : \vec{\sigma} \to \vec{\tau}} \qquad \frac{\Gamma \vdash \vec{t} : \vec{\sigma} \to \vec{\tau} \qquad \Gamma \vdash \vec{u} : \vec{\sigma}}{\Gamma \vdash \vec{t}\ \vec{u} : \vec{\tau}}$$

Sequences also preserve β-reduction, i.e.

$$(\lambda\vec{x}.\, \vec{t})\ \vec{u} \equiv_\beta \vec{t}[\vec{x} := \vec{u}]$$

with the same restrictions as above and where $\equiv_\beta$ is interpreted pointwise.

Proof. By induction on the length of the sequences considered. □

This allows us to reason on sequences almost as if they were plain terms and types. This works particularly well because we only have one logical connective at the level of proof-terms, the arrow, and probably explains how Gödel recovered a tractable system. This becomes sharply more delicate to handle if we wish to add other constructions to the language. Indeed, we did not define such sequence notations for the integer structures of System T.

In the following, we will often use sequences implicitly when the context is clear.

7.5.2 Witnesses and counters

We now turn to define the witnesses and counters of a formula A of **HA**.

Definition 83. Given a formula A of **HA**, the witnesses $\mathbb{W}(A)$ and counters $\mathbb{C}(A)$ of A are defined as a sequence of System T types, by induction on A.

- $\mathbb{W}(t = u) = \mathbb{C}(t = u) = \emptyset$
- $\mathbb{W}(\bot) = \mathbb{C}(\bot) = \emptyset$
- $\mathbb{W}(\top) = \mathbb{C}(\top) = \emptyset$
- $\mathbb{W}(A \wedge B) = \mathbb{W}(A)\,;\,\mathbb{W}(B)$ and $\mathbb{C}(A \wedge B) = \mathbb{C}(A)\,;\,\mathbb{C}(B)$
- $\mathbb{W}(A \vee B) = \mathbb{N}\,;\,\mathbb{W}(A)\,;\,\mathbb{W}(B)$ and $\mathbb{C}(A \vee B) = \mathbb{C}(A)\,;\,\mathbb{C}(B)$
- $\mathbb{W}(A \to B) = \begin{cases} \mathbb{W}(A) \to \mathbb{W}(B)\,; \\ \mathbb{W}(A) \to \mathbb{C}(B) \to \mathbb{C}(A) \end{cases}$ and $\mathbb{C}(A \to B) = \mathbb{W}(A)\,;\,\mathbb{C}(B)$
- $\mathbb{W}(\forall x.\, A) = \mathbb{N} \to \mathbb{W}(A)$ and $\mathbb{C}(\forall x.\, A) = \mathbb{N}\,;\,\mathbb{C}(A)$
- $\mathbb{W}(\exists x.\, A) = \mathbb{N}\,;\,\mathbb{W}(A)$ and $\mathbb{C}(\exists x.\, A) = \mathbb{N} \to \mathbb{C}(A)$

Notation 4. Witnesses and counters are readily lifted to sequences of types pointwise.

$$\begin{array}{rcl} \mathbb{W}(A_1\,;\ldots;\,A_n) & := & \mathbb{W}(A_1)\,;\ldots;\,\mathbb{W}(A_n) \\ \mathbb{C}(A_1\,;\ldots;\,A_n) & := & \mathbb{C}(A_1)\,;\ldots;\,\mathbb{C}(A_n) \end{array}$$

Interestingly enough, the $\mathbb{W}(-)$ and $\mathbb{C}(-)$ translations are not sensible to the terms contained in the formula being translated. This is formally stated below.

Proposition 36. *Let A be a* **HA***-formula and x a free variable of A. Then for any* **HA***-term t, the following equalities hold.*

$$\begin{array}{rcl} \mathbb{W}(A[x := t]) & = & \mathbb{W}(A) \\ \mathbb{C}(A[x := t]) & = & \mathbb{C}(A) \end{array}$$

Proof. By induction on A. □

This will allow us to freely forget about substitution in type translation, a property we will be using quite often.

7.5.3 Interpretation

The final ingredient we need to define the Dialectica interpretation is the interpretation matrix, which is a formula in $\mathbf{HA} + T$ defined by induction on a given formula in **HA**, parameterized by System T term whose types match the witness and counter sequences.

Definition 84 (Interpretation matrix)**.** Assume a formula A of **HA**. We define its interpretation matrix $A_D[\vec{u}, \vec{x}]$, a quantifier-free formula of $\mathbf{HA} + T$ by induction, where $\vec{u}$ and $\vec{x}$ a are sequences of fresh variables parameterizing the formula respectively of the same length as $\mathbb{W}(A)$ and $\mathbb{C}(A)$. We just write A_D when both sequences are empty. We ensure by construction that the free variables of $A_D[\vec{u}, \vec{x}]$ are exactly the free variables of A together with $\vec{u}, \vec{x}$.

The inductive definition is given below.

- $(t = u)_D := t = u$
- $\bot_D := \bot$
- $\top_D := \top$
- $(A \wedge B)_D[\vec{u}\,;\,\vec{v}, \vec{x}\,;\,\vec{y}] := A_D[\vec{u}, \vec{x}] \wedge B_D[\vec{v}, \vec{y}]$
- $(A \vee B)_D[b\,;\,\vec{u}\,;\,\vec{v}, \vec{x}\,;\,\vec{y}] := (b = 0 \wedge A_D[\vec{u}, \vec{x}]) \vee (b = 1 \wedge B_D[\vec{v}, \vec{y}])$
- $(A \to B)_D[\vec{\varphi}\,;\,\vec{\psi}, \vec{u}\,;\,\vec{y}] := A_D[\vec{u}, \vec{\psi}\,\vec{u}\,\vec{y}] \to B_D[\vec{\varphi}\,\vec{u}, \vec{y}]$
- $(\forall z.\, A)_D[\vec{u}, z\,;\,\vec{x}] := A_D[\vec{u}\ z, \vec{x}]$
- $(\exists z.\, A)_D[z\,;\,\vec{u}, \vec{x}] := A_D[\vec{u}, \vec{x}\ z]$

Note that the binders used in quantifications should match, so that the property that A and A_D share the same free variables up to $\vec{u}$ and $\vec{x}$ is preserved.

Let us step back a bit and look at the system in its entirety before going any further. There are two interesting cases.

- First, if we consider the translation of the $\vee$ connective, we can see that it is no more than an encoding of sum types using pairs. Indeed, its interpretation amounts to the usual coding of sum types as tagged records in modern programming languages: here, the tag is an integer used as a boolean discriminating which field of the record is actually containing something relevant. The other field is, as we are going to see in the translation of proofs, filled with a dummy argument, amounting to a garbage placeholder. Gödel had to use such encoding tricks because System T is indeed lacking true sum types.
- The arrow case is the most complicated one. We will alleviate the perplexity of the reader by giving a bit of typing of this statement. Recall that

$$\begin{array}{rcl} \mathbb{W}(A \to B) & := & \mathbb{W}(A) \to \mathbb{W}(B)\,;\,\mathbb{W}(A) \to \mathbb{C}(B) \to \mathbb{C}(A) \\ \mathbb{C}(A \to B) & := & \mathbb{W}(A)\,;\,\mathbb{C}(B) \end{array}$$

so that, with the name of the bound variables above, we have:

$$\begin{array}{rcl} \vec{\varphi} & : & \mathbb{W}(A) \to \mathbb{W}(B) \\ \vec{\psi} & : & \mathbb{W}(A) \to \mathbb{C}(B) \to \mathbb{C}(A) \\ \vec{u} & : & \mathbb{W}(A) \\ \vec{y} & : & \mathbb{C}(B) \end{array}$$

From this we need to construct a pair of witness and counter for both A and B. And indeed, the terms used to feed A_D and B_D have the right type, because:

$$\begin{array}{rcl} \vec{\psi}\ \vec{u}\ \vec{y} & : & \mathbb{C}(A) \\ \vec{\varphi}\ \vec{u} & : & \mathbb{W}(B) \end{array}$$

From a realizability standpoint, the other cases are defined as a plain extraction of the considered connective.

We will implicitly use the A_D translation applied to any System T terms. There is nothing complicated, but we give the formal definition below nonetheless. Note that we actually already used this notation in the definition of the interpretation matrix without insisting on it.

Definition 85. We extend the definition of A_D to System T terms by substitution, that is, for any sequence of terms $\vec{v}$ and $\vec{w}$ of the right length,

$$A_D[\vec{v}, \vec{w}] := (A_D[\vec{u}, \vec{x}])[\vec{u} := \vec{v}, \vec{x} := \vec{w}]$$

where $\vec{u}$ and $\vec{x}$ are fresh variables.

Notation 5. As usual, we define the interpretation matrix of a sequence of **HA** formulae as a sequence of $\mathbf{HA} + T$ formulae componentwise, that is, if $\vec{u} : \mathbb{W}(\vec{A})$ and $\vec{x} : \mathbb{C}(\vec{A})$, we pose:

$$(A_1 \ ; \ \dots \ ; \ A_n)[\vec{u}, \vec{x}] := A_{1D}[u_1, x_1] \ ; \ \dots \ ; \ A_{nD}[u_n, x_n]$$

Luckily, the interpretation of a **HA**-formula is a well-formed formula of $\mathbf{HA} + T$.

Proposition 37. *Let A be a* **HA***-formula whose free variables are $\vec{n}$. Then*

$$\vec{n} : \vec{\mathbb{N}}, \vec{u} : \mathbb{W}(A), \vec{x} : \mathbb{C}(A) \vdash_{\text{wf}} A_D[\vec{u}, \vec{x}]$$

for any fresh variables $\vec{u}$, $\vec{x}$.

Proof. By induction on A. □

Proposition 38. *Let A be a* **HA***-formula and z a free variable of A. Assume $\vec{u}$ and $\vec{x}$ be two sequences of terms such that z is not free in them. Then for all* **HA***-term t, we have*

$$(A[z := t])_D[\vec{u}, \vec{x}] = (A_D[\vec{u}, \vec{x}])[z := [\![t]\!]]$$

when $\vec{u}$ and $\vec{x}$ have the correct length.

Proof. By induction on A. □

7.5.4 Soundness theorem

For now, we are going to state and prove the logical soundness of this realizability interpretation. We first need to define formally the realizability relation.

Definition 86 (Realizability)**.** Let A be a formula of **HA** whose free variables are $\vec{n}$. We say that a sequence of System T terms $\vec{u}$ realizes A, written $\vec{u} \Vdash A$, if:

- $\vec{n} : \mathbb{N} \vdash \vec{u} : \mathbb{W}(A)$
- $\vec{x} : \mathbb{C}(A), \vec{n} : \mathbb{N} \mid \cdot \vdash A_D[\vec{u}, \vec{x}]$

where the $\vec{x}$ variables are fresh. If A is realized by some sequence, we will just write $\Vdash A$.

We then get to the following soundness result, which states that proofs of **HA** are transported to System T realizers.

Theorem 18. *Let A be a* **HA***-formula. If $\vdash A$, then $\Vdash A$.*

The proof of this theorem is not that complicated, but we need to base ourselves on some ancillary lemmas first, because the proof terms involved may be rather complex to analyse at first sight.

Remark 10. Because we are still sticking to the historical presentation, we will manipulate whole derivations instead of λ-terms. This has some drawbacks, one of them being the high verbosity of derivations, but there is nothing to do about it for now. We will need in particular to give names to derivations. To discriminate them from other objects, they will be named using the fraktur typeface $\mathfrak{p}$, $\mathfrak{q}$, etc.

With those conventions, the statement of the theorem we want to prove takes a clearly more proof-as-program-ish flavour, as it becomes: for all derivation $\mathfrak{p}$ of $\vdash A$, one may construct a sequence of terms $[\mathfrak{p}] \Vdash A$. We will therefore start to willingly merge the notions of proofs, sequents and derivations when there is no risk of confusion.

For now, we return to the basic structures we need to define the translation. As mentioned before, Gödel used encoding tricks that are available in our arithmetical setting. One of them relies on the fact that any System T type is inhabited.

Proposition 39. *Let σ be a System T type. Then there exists a term $✠_\sigma$, hereafter called the* dummy term *of σ, such that $\vdash ✠_\sigma : \sigma$.*

Proof. We build it by induction over σ:

- $✠_{\mathbb{N}} := \mathbf{0}$
- $✠_{\sigma \to \tau} := \lambda\_.✠_\tau$

□

Remark 11. The ✠ notation for the dummy term is inspired by ludics, where there exists what would be the *daimon*, a canonically aborting term, if ludics were to be written using terms, following Terui's presentation [103].

Here, dummy terms serve a similar but distinct purpose, that is, to be some placeholder *a priori* unrelated to any orthogonality. Yet, as we will see later, the dummy term can also be seen as a call-by-name exception, which is precisely the rôle devoted to the *daimon*, hence the notation.

Notation 6. As for all the other constructs, we can extend dummy terms to sequences of types in the obvious way, that is, componentwise.

$$✠_{\sigma_1 \,;\, \dots \,;\, \sigma_n} := ✠_{\sigma_1} \,;\, \dots \,;\, ✠_{\sigma_n}$$

Definition 87 (Booleans)**.** We can loosely encode booleans in System T using integers. Any non-zero integer is going to be considered as true, while **0** will be encoding false. Using the recursor we can write out the usual boolean connectives. For the sake of readability, we define the following constants, and write the type $\mathbb{B}$ as an alias for $\mathbb{N}$ used as booleans.

$$\begin{array}{rcll} \mathrm{true} & : & \mathbb{B} & := \mathbf{S}\,\mathbf{0} \\ \mathrm{false} & : & \mathbb{B} & := \mathbf{0} \\ \mathrm{ifz} & : & \mathbb{N} \to A \to A \to A & := \lambda n\, t\, u.\, \mathbf{rec}\ n\ t\ (\lambda p\, r.\, u) \\ \mathrm{and} & : & \mathbb{B} \to \mathbb{B} \to \mathbb{B} & := \lambda t\, u.\, \mathbf{rec}\ t\ \mathbf{0}\ (\lambda p\, r.\, u) \end{array}$$

As their name suggests, true and false will encode the two boolean values, while and is the logical and of two booleans, and ifz tests whether an integer is zero or not. Likewise, we can define in System T a term that tests equality of integers, and return a pseudo-boolean accordingly.

$$\mathrm{eqb} : \mathbb{N} \to \mathbb{N} \to \mathbb{B} := \lambda m.\, \mathbf{rec}\ m\ (\lambda n.\, \mathrm{ifz}\ n\ \mathrm{true}\ \mathrm{false})\ (\lambda p\, r\, n.\, \mathbf{rec}\ n\ \mathrm{false}\ (\lambda q\, s.\, r\ q))$$

Proposition 40. *Let A be a* **HA***-formula with free variables $\vec{n}$. Then there exists a System T term*

$$\vec{n} : \mathbb{N} \vdash \mathrm{decide}_A : \mathbb{W}(A) \to \mathbb{C}(A) \to \mathbb{B}$$

such that the following formula is provable in **HA** $+ T$*:*

$$\vec{n} : \mathbb{N}, \vec{u} : \mathbb{W}(A), \vec{x} : \mathbb{C}(A) \mid \cdot \vdash (\mathrm{decide}_A\ \vec{u}\ \vec{x} = 1 \wedge A_D[\vec{u}, \vec{x}]) \vee (\mathrm{decide}_A\ \vec{u}\ \vec{x} = 0 \wedge \neg A_D[\vec{u}, \vec{x}])$$

Proof. The decide_A term is built by induction on the formula A.

- $\mathrm{decide}_{t=u} := \mathrm{eqb}\ t\ u$
- $\mathrm{decide}_{\top} := \mathrm{true}$
- $\mathrm{decide}_{\bot} := \mathrm{false}$

- $\mathrm{decide}_{A\wedge B} := \lambda \vec{u}\,\vec{v}\,\vec{x}\,\vec{y}.\,\mathrm{and}\ (\mathrm{decide}_A\ \vec{u}\ \vec{x})\ (\mathrm{decide}_B\ \vec{v}\ \vec{y})$
- $\mathrm{decide}_{A\vee B} := \lambda b\,\vec{u}\,\vec{v}\,\vec{x}\,\vec{y}.\,\mathrm{ifz}\ b\ (\mathrm{decide}_A\ \vec{u}\ \vec{x})\ (\mathrm{decide}_B\ \vec{v}\ \vec{y})$
- $\mathrm{decide}_{A\to B} := \lambda \vec{\varphi}\,\vec{\psi}\,\vec{u}\,\vec{y}.\,\mathrm{ifz}\ (\mathrm{decide}_A\ \vec{u}\ (\vec{\psi}\ \vec{u}\ \vec{x}))\ \mathrm{true}\ (\mathrm{decide}_B\ (\vec{\varphi}\ \vec{u})\ \vec{y})$
- $\mathrm{decide}_{\forall z.\,A} := \lambda \vec{u}\,z\,\vec{x}.\,\mathrm{decide}_A\ (\vec{u}\ z)\ \vec{x}$
- $\mathrm{decide}_{\exists z.\,A} := \lambda z\,\vec{u}\,\vec{x}.\,\mathrm{decide}_A\ \vec{u}\ (\vec{x}\ z)$

Remark that decide_A shares the same free variables as A, so that as before, we need the bound variables to agree on their names in the case of quantifiers.

The specification proof is likewise proved by induction on the formula and repeated case analysis. There is nothing tricky because all of the work is in the term definitions, so we skip the boring details.

□

Otherwise stated, this result tells us that A_D relation is decidable. Therefore, we may reason with it as if we were in classical logic. It also allows us to test whether $A_D[u, x]$ holds for some terms u and x *in System T itself*. In particular, we will be critically using the following in the Dialectica translation.

Proposition 41. *For any* **HA***-formula A with free variables $\vec{n}$, there exists a sequence of λ-terms*

$$\vec{n} : \mathbb{N} \vdash \mathrm{merge}_A : \mathbb{C}(A) \to \mathbb{C}(A) \to \mathbb{W}(A) \to \mathbb{C}(A)$$

with the following property:

$$\vec{n} : \mathbb{N}, u : \mathbb{W}(A), x_1 : \mathbb{C}(A), x_2 : \mathbb{C}(A) \mid \cdot \vdash A_D[u, \mathrm{merge}_A\ x_1\ x_2\ u] \leftrightarrow A_D[u, x_1] \wedge A_D[u, x_2]$$

Proof. Just take the following sequence of terms for a given A:

$$\mathrm{merge}_A := \lambda x_1\,x_2\,u.\,\mathrm{ifz}\ (\mathrm{decide}_A\ u\ x_1)\ x_1\ x_2$$

with a little abuse of sequence notation on ifz (it should be a sequence of the same length as x_1). Let us show that this implementation satisfies the specification given above.

- Suppose $A_D[u, \mathrm{merge}_A\ x_1\ x_2\ u]$. Because A_D is decidable, we can assume that either $A_D[u, x_1]$, or $\neg A_D[u, x_1]$.

  In the first case, by the specification of decide, we have $\mathrm{decide}_A\ u\ x_1 \equiv 1$ and hence $\mathrm{merge}_A\ x_1\ x_2\ u \equiv x_2$. Therefore by convertibility, $A_D[u, x_2]$.

  In the second case, we have conversely $\mathrm{decide}_A\ u\ x_1 \equiv 0$ and $\mathrm{merge}_A\ x_1\ x_2\ u \equiv x_1$. This means that $A_D[u, x_1]$, which is absurd.

- Suppose $A_D[u, x_1] \wedge A_D[u, x_2]$. In particular $A_D[u, x_1]$ holds, hence we have

$$\mathrm{decide}_A\ u\ x_1 \equiv 1.$$

  This also entails $\mathrm{merge}_A\ x_1\ x_2\ u \equiv x_2$, from which we get that

$$A_D[u, \mathrm{merge}_A\ x_1\ x_2\ u] \equiv A_D[u, x_2]$$

  which we prove using the second component of our initial assumption.

□

Notation 7. The merge operation can also be extended to sequences, although it is slightly more complicated than the usual notations that can be extended componentwise. Here we have to do a little bit of variable bookkeeping and boolean manipulation, but it amounts to a boolean conjunction of each pointwise merge.

To finally prove the soundness theorem, we need to generalize a bit its statement to accommodate the proof of sequents, not only hypothesis-free ones.

Notation 8. We will abuse the realization relation, and write $\vec{u} \Vdash \Gamma_1, \ldots, \Gamma_n \vdash A$ whenever $\vec{u} \Vdash \Gamma_1 \wedge \ldots \wedge \Gamma_n \to A$. Note that when the environment is empty, the realizations of a formula and of a sequent coincide.

Proposition 42. *A sequence of terms realizing a proof $\mathfrak{p}$ of the sequent $\Gamma_1, \ldots, \Gamma_n \vdash A$ whose free variables range over $\vec{m}$ is equivalently given by $n+1$ sequences of terms $(\mathfrak{p}^\bullet, \mathfrak{p}_1^\circ, \ldots, \mathfrak{p}_n^\circ)$ of the following types:*

- $\vec{m} : \mathbb{N} \vdash \mathfrak{p}^\bullet : \mathbb{W}(\Gamma_1) \to \ldots \to \mathbb{W}(\Gamma_n) \to \mathbb{W}(A)$
- $\vec{m} : \mathbb{N} \vdash \mathfrak{p}_i^\circ : \mathbb{W}(\Gamma_1) \to \ldots \to \mathbb{W}(\Gamma_n) \to \mathbb{C}(A) \to \mathbb{C}(\Gamma_i)$

such that the following holds in **HA** $+ T$*:*

$$\vec{m} : \mathbb{N}, \vec{u} : \mathbb{W}(\Gamma), \vec{x} : \mathbb{C}(A) \mid \cdot \vdash \Gamma_D[\vec{u}, \mathfrak{p}^\circ\ \vec{u}\ \vec{x}] \to A_D[\mathfrak{p}^\bullet\ \vec{u}, \vec{x}]$$

where

$$\mathfrak{p}^\circ := \mathfrak{p}_1^\circ\ ;\ \ldots\ ;\ \mathfrak{p}_n^\circ$$
$$\Gamma := \Gamma_1\ ;\ \ldots\ ;\ \Gamma_n$$

Proof. By induction on n, and by simple unfolding of the definitions of $\mathbb{W}(-)$, $\mathbb{C}(-)$ and $(-)_D$. □

We will freely switch our point of view in the nature of the realizers, for instance from $\mathfrak{p}_1^\circ\ ;\ \ldots\ ;\ \mathfrak{p}_n^\circ$ to $\mathfrak{p}^\circ$ and vice versa. This will allow a handful of abuses of notation.

We are now equipped with all the necessary material to state and prove the soundness lemma, which is no more than a generalization of the theorem we wanted to prove.

Proposition 43. *If* $\mathfrak{p} : \Gamma \vdash A$ *then there exists a tuple* $(\mathfrak{p}^\bullet, \mathfrak{p}^\circ)$ *such that* $(\mathfrak{p}^\bullet, \mathfrak{p}^\circ) \Vdash \Gamma \vdash A$.

Proof. By induction on the proof $\mathfrak{p}$. We first define the realizers, and we will show afterwards that they do realize the given sequent. In order to ease readability, we may freely annotate the type of bound sequences of variables.

We classify the rules to be interpreted just as we presented them, into propositional logic, first-order logic and arithmetic. Indeed, the proof arguments tend to be similar inside each class.

Propositional logic Let us first interpret the propositional part of **HA**.

- Rule $\left[\dfrac{}{\mathfrak{p} : \Gamma_1, \dots, \Gamma_n \vdash \Gamma_i} \right]$:

  If we let ourselves be guided by the typing of realizers, the translations for $\mathfrak{p}^\bullet$ and $\mathfrak{p}_i^\circ$ are straightforward and amount to a sequence of projections.

$$\begin{aligned} \mathfrak{p}^\bullet &: \mathbb{W}(\Gamma_1) \to \dots \to \mathbb{W}(\Gamma_n) \to \mathbb{W}(\Gamma_i) \\ \mathfrak{p}^\bullet &:= \lambda(x_1 : \mathbb{W}(\Gamma_1)) \dots (x_n : \mathbb{W}(\Gamma_n)).\, x_i \end{aligned}$$

$$\begin{aligned} \mathfrak{p}_i^\circ &: \mathbb{W}(\Gamma_1) \to \dots \to \mathbb{W}(\Gamma_n) \to \mathbb{C}(\Gamma_i) \to \mathbb{C}(\Gamma_i) \\ \mathfrak{p}_i^\circ &:= \lambda(x_1 : \mathbb{W}(\Gamma_1)) \dots (x_n : \mathbb{W}(\Gamma_n))\, (w : \mathbb{C}(\Gamma_i)).\, w \end{aligned}$$

  The reader should not be fooled by the sequence notation for terms. The translation $\mathfrak{p}^\bullet$ for instance, which seems a proper term, gives something a little less palatable when fully expanded. Its full definition is actually of the following form:

$$\mathfrak{p}^\bullet := \begin{cases} \lambda x_{1,1} \dots x_{1,k_1} \dots x_{n,1} \dots x_{n,k_n}.\, x_{i,1} \ ; \\ \qquad\qquad \dots \\ \lambda x_{1,1} \dots x_{1,k_1} \dots x_{n,1} \dots x_{n,k_n}.\, x_{i,k_i} \end{cases}$$

  where $k_j := |\mathbb{W}(\Gamma_j)|$. Thus, it is not as elegant as it seems at first sight.

  Real issues arise when considering $\mathfrak{p}_j^\circ$ when $i \neq j$. Indeed, we have to provide a sequence inhabiting $\mathbb{C}(\Gamma_j)$ out of thin air, because there is no natural way to construct it out of the environment. This is precisely the place where we use the $✠_{\mathbb{C}(\Gamma_j)}$ terms.

$$\begin{aligned} \mathfrak{p}_j^\circ &: \mathbb{W}(\Gamma_1) \to \dots \to \mathbb{W}(\Gamma_n) \to \mathbb{C}(\Gamma_i) \to \mathbb{C}(\Gamma_j) \\ \mathfrak{p}_j^\circ &:= \lambda(x_1 : \mathbb{W}(\Gamma_1)) \dots (x_n : \mathbb{W}(\Gamma_n))\, (w : \mathbb{C}(\Gamma_i)).\, ✠_{\mathbb{C}(\Gamma_j)} \end{aligned}$$

Note that the use of $\maltese_{\mathbb{C}(\Gamma_j)}$ is typically required when there is some weakening occurring. We will see this pattern appearing in the other rules featuring some form of weakening.

We now show that the given terms realize the sequent. Let $\vec{u} : \mathbb{W}(\Gamma)$ and $x : \mathbb{C}(\Gamma_i)$. We must prove:

$$\Gamma_{1D}[u_1, \mathfrak{p}_1^\circ\ \vec{u}\ x] \to \ldots \to \Gamma_{nD}[u_n, \mathfrak{p}_n^\circ\ \vec{u}\ x] \to \Gamma_{iD}[\mathfrak{p}^\bullet\ \vec{u}, x]$$

But $\mathfrak{p}_i^\circ\ \vec{u}\ x \equiv_\beta x$ and $\mathfrak{p}^\bullet\ \vec{u} \equiv_\beta u_i$, while $\mathfrak{p}_j^\circ\ \vec{u}\ x \equiv_\beta \maltese_{\mathbb{C}(\Gamma_j)}$ when $j \neq i$. So the above formula is convertible to the one below.

$$\Gamma_{1D}[u_1, \maltese_{\mathbb{C}(\Gamma_1)}] \to \ldots \to \Gamma_{iD}[u_i, x] \to \ldots \to \Gamma_{nD}[u_n, \maltese_{\mathbb{C}(\Gamma_n)}] \to \Gamma_{iD}[u_i, x]$$

Hence we just apply the i-th hypothesis.

One may observe that we did not rely on any property of the dummy terms we put in the reverse proofs. The realization relation only required the realization hypothesis recovered from the hypothesis used by the underlying proof.

- Rule $\left[\dfrac{\mathfrak{q} : \Gamma, A \vdash B}{\mathfrak{p} : \Gamma \vdash A \to B}\right]$:

  We have from $\mathfrak{q}$:

$$\begin{array}{rcl} \mathfrak{q}^\bullet & : & \mathbb{W}(\Gamma)\ ;\ \mathbb{W}(A) \to \mathbb{W}(B) \\ \mathfrak{q}_A^\circ & : & \mathbb{W}(\Gamma)\ ;\ \mathbb{W}(A)\ ;\ \mathbb{C}(B) \to \mathbb{C}(A) \\ \mathfrak{q}_{\Gamma_i}^\circ & : & \mathbb{W}(\Gamma)\ ;\ \mathbb{W}(A)\ ;\ \mathbb{C}(B) \to \mathbb{C}(\Gamma_i) \end{array}$$

  from which we can form:

$$\begin{array}{rcl} \mathfrak{p}^\bullet & : & \mathbb{W}(\Gamma) \to \mathbb{W}(A \to B) \\ \mathfrak{p}^\bullet & := & \mathfrak{q}^\bullet\ ;\ \mathfrak{q}_A^\circ \end{array}$$

$$\begin{array}{rcl} \mathfrak{p}_i^\circ & : & \mathbb{W}(\Gamma) \to \mathbb{C}(A \to B) \to \mathbb{C}(\Gamma_i) \\ \mathfrak{p}_i^\circ & := & \mathfrak{q}_{\Gamma_i}^\circ \end{array}$$

  There is essentially nothing to do at the level of proofs apart from rearranging the components, as types coincide.

  Proving the realization property is just as easy. Indeed we must prove that for all $\vec{u} : \mathbb{W}(\Gamma)$, $v : \mathbb{W}(A)$ and $y : \mathbb{C}(B)$ we have:

$$\Gamma_D[\vec{u}, \mathfrak{p}^\circ\ \vec{u}\ v\ y] \to (A \to B)_D[\mathfrak{p}^\bullet\ \vec{u}, v\ ;\ y]$$

But unfolding the arrow relation and the terms within it gives us:

$$(A \to B)_D[\mathfrak{p}^\bullet\ \vec{u}, v\ ;\ y] \equiv A_D[\vec{u}, \mathfrak{q}^\circ_A\ \vec{u}\ y] \to B_D[\mathfrak{q}^\bullet\ \vec{u}, y]$$

If we unfold the remaining $\mathfrak{p}^\circ_i$, we recover exactly the realization property for $\mathfrak{q}$. We conclude by the induction hypothesis.

- Rule $\left[\dfrac{\mathfrak{q} : \Gamma \vdash A \to B \qquad \mathfrak{r} : \Gamma \vdash A}{\mathfrak{p} : \Gamma \vdash B}\right]$:

This rule is, as most rules introducing duplications of hypotheses, rather complicated to explain and write out. We will take special care to discuss it, and will pass on the other rules displaying similar constructions more quickly.

Let us start recalling a bit of typing. From $\mathfrak{q}$ we recover:

$$\begin{array}{rcl} \mathfrak{q}^\bullet & : & \mathbb{W}(\Gamma) \to \mathbb{W}(A \to B) \\ & : & \mathbb{W}(\Gamma) \to (\mathbb{W}(A) \to \mathbb{W}(B)\ ;\ \mathbb{W}(A) \to \mathbb{C}(B) \to \mathbb{C}(A)) \\ & : & (\mathbb{W}(\Gamma) \to \mathbb{W}(A) \to \mathbb{W}(B))\ ;\ (\mathbb{W}(\Gamma) \to \mathbb{W}(A) \to \mathbb{C}(B) \to \mathbb{C}(A)) \end{array}$$

We can therefore extract from $\mathfrak{q}^\bullet$ a first component $\mathfrak{q}^+ : \mathbb{W}(\Gamma)\ ;\ \mathbb{W}(A) \to \mathbb{W}(B)$ from which we can build the translation $\mathfrak{p}^\bullet$:

$$\begin{array}{rcl} \mathfrak{p}^\bullet & : & \mathbb{W}(\Gamma) \to \mathbb{W}(B) \\ \mathfrak{p}^\bullet & := & \lambda \vec{x} : \mathbb{W}(\Gamma).\, \mathfrak{q}^+\ \vec{x}\ (\mathfrak{r}^\bullet\ \vec{x}) \end{array}$$

The translation $\mathfrak{p}_i$ is much more involved. Indeed, there are two ways to recover a sequence of counters $\mathbb{C}(\Gamma_i)$ from the derivation: one coming from $\mathfrak{q}$, the other one from $\mathfrak{r}$. One could just arbitrarily choose one of the two sources and drop the other. This is acceptable if we are only interested in typing, but as we will see, this would break the fact that the translated proof realizes the formula. In order for the translation to work, we need to use the merge term we defined before, so that we dynamically choose one side over the other.

Let us first give the name $\mathfrak{q}^-$ to the remaining component of the translated proof $\mathfrak{q}^\bullet$, which has the following type.

$$\mathfrak{q}^- : \mathbb{W}(\Gamma) \to \mathbb{W}(A) \to \mathbb{C}(B) \to \mathbb{C}(A)$$

From here, we can construct the two aforementioned counters of $\mathbb{C}(\Gamma)$ from respectively $\mathfrak{q}$ and $\mathfrak{r}$.

$$\begin{array}{rcl} \mathfrak{c}_l & : & \mathbb{W}(\Gamma) \to \mathbb{C}(B) \to \mathbb{C}(\Gamma) \\ \mathfrak{c}_l & := & \lambda(\vec{x} : \mathbb{W}(\Gamma))\,(w : \mathbb{C}(B)).\, \mathfrak{q}^\circ\ \vec{x}\ (\mathfrak{r}^\bullet\ \vec{x})\ w \end{array}$$

$$\begin{array}{rcl} \mathfrak{c}_r & : & \mathbb{W}(\Gamma) \to \mathbb{C}(B) \to \mathbb{C}(\Gamma) \\ \mathfrak{c}_r & := & \lambda(\vec{x} : \mathbb{W}(\Gamma))\,(w : \mathbb{C}(B)).\, \mathfrak{r}^\circ\ \vec{x}\ (\mathfrak{q}^-\ \vec{x}\ (\mathfrak{r}^\bullet\ \vec{x})\ w) \end{array}$$

As explained before, we only have to merge the resulting counters.

$$\begin{aligned}\mathfrak{p}^\circ \quad &: \quad \mathbb{W}(\Gamma) \to \mathbb{C}(B) \to \mathbb{C}(\Gamma)\\ \mathfrak{p}^\circ \quad &:= \quad \lambda(\vec{x} : \mathbb{W}(\Gamma))\,(w : \mathbb{C}(B)).\,\mathrm{merge}_\Gamma\ (\mathfrak{c}_l\ \vec{x}\ w)\ (\mathfrak{c}_r\ \vec{x}\ w)\ \vec{x}\end{aligned}$$

The use of the merge operator is, as hinted above, necessary when there is some duplication of hypotheses occurring, dually to the $\maltese_-$ term which is used when there is some weakening. We will be keeping seeing these facts occurring in the next rule interpretations.

Let us prove now that the given term realizes the sequent. This amounts to proving the following, for any $\vec{u} : \mathbb{W}(\Gamma)$ and $y : \mathbb{C}(B)$.

$$\Gamma_D[\vec{u}, \mathfrak{p}^\circ\ \vec{u}\ y] \to B_D[\mathfrak{p}^\bullet\ \vec{u}, y]$$

A little bit of unfolding gives us:

$$\Gamma_D[\vec{u}, \mathrm{merge}_\Gamma\ (\mathfrak{c}_l\ \vec{u}\ y)\ (\mathfrak{c}_r\ \vec{u}\ y)\ \vec{u}] \to B_D[\mathfrak{p}^\bullet\ \vec{u}, y]$$

By the property of merge, this is equivalent to:

$$\Gamma_D[\vec{u}, \mathfrak{c}_l\ \vec{u}\ y] \to \Gamma_D[\vec{u}, \mathfrak{c}_r\ \vec{u}\ y] \to B_D[\mathfrak{p}^\bullet\ \vec{u}, y]$$

If we unfold the remaining definitions, we finally get:

$$\Gamma_D[\vec{u}, \mathfrak{q}^\circ\ \vec{u}\ (\mathfrak{r}^\bullet\ \vec{u})\ y] \to \Gamma_D[\vec{u}, \mathfrak{r}^\circ\ \vec{u}\ (\mathfrak{q}^-\ \vec{u}\ (\mathfrak{r}^\bullet\ \vec{u})\ y)] \to B_D[\mathfrak{q}^+\ \vec{u}\ (\mathfrak{r}^\bullet\ \vec{u}), y]$$

We can now use the induction hypotheses on $\mathfrak{q}$ and $\mathfrak{r}$. Remark that:

$$x := \mathfrak{q}^-\ \vec{u}\ (\mathfrak{r}^\bullet\ \vec{u})\ y : \mathbb{C}(A)$$

so that we can instantiate the hypothesis on $\mathfrak{r}$ with $\vec{u}$ and x, and get:

$$\Gamma_D[\vec{u}, \mathfrak{r}^\circ\ \vec{u}\ x] \to A_D[\mathfrak{r}^\bullet\ \vec{u}, x]$$

In order to check the realization relation, by applying this implication to the second hypothesis of our starting point, it is therefore sufficient to prove that:

$$\Gamma_D[\vec{u}, \mathfrak{q}^\circ\ \vec{u}\ (\mathfrak{r}^\bullet\ \vec{u})\ y] \to A_D[\mathfrak{r}^\bullet\ \vec{u}, \mathfrak{q}^-\ \vec{u}\ (\mathfrak{r}^\bullet\ \vec{u})\ y] \to B_D[\mathfrak{q}^+\ \vec{u}\ (\mathfrak{r}^\bullet\ \vec{u}), y]$$

Yet, as the careful reader may already have realized[2], this is no more than the induction hypothesis on $\mathfrak{q}$ instantiated with $\vec{u}$ and $\mathfrak{r}^\bullet\ \vec{u}$; y. Indeed, the above formula is convertible to:

$$\Gamma_D[\vec{u}, \mathfrak{q}^\circ\ \vec{u}\ (\mathfrak{r}^\bullet\ \vec{u}\ ;\ y)] \to (A \to B)_D[\mathfrak{q}^\bullet\ \vec{u}, (\mathfrak{r}^\bullet\ \vec{u}\ ;\ y)]$$

[2]No pun intended.

Thus we are done.

Remark that, in the proof, the use of the merge term was an absolute requirement. We needed both hypotheses on $\mathfrak{q}$ and $\mathfrak{r}$ for the proof to pass through. Had we chosen arbitrarily one of the two sequences of counters mentioned before, we would not have been able to show that the term realized the formula, for we would have lacked the hypothesis relative either to $\mathfrak{q}$ or $\mathfrak{r}$.

- Rule $\left[\dfrac{}{\mathfrak{p} : \Gamma \vdash \top}\right]$:

  In this case, the $\mathfrak{p}^\bullet$ sequence is empty because the $\mathbb{W}(\top)$ sequence itself is empty. To construct the reverse proofs $\mathfrak{p}_i^\circ$, we simply use the dummy term, as in the axiom case. This results in the following.

$$\begin{array}{lcl} \mathfrak{p}^\bullet & := & \emptyset \\ \mathfrak{p}_i^\circ & := & \lambda(\vec{x} : \mathbb{W}(\Gamma))\,(w : \mathbb{C}(\top)).\, \maltese_{\mathbb{C}(\Gamma_i)} \end{array}$$

  Remark that in this case, the sequence of variables w is actually empty. We only leave it here for uniformity with the other translations.

  The realization relation is, in this case, that for all $\vec{u} : \mathbb{W}(\Gamma)$:

$$\Gamma_D[\vec{u}, \maltese_{\mathbb{C}(\Gamma)}] \to \top$$

  which is trivially true so that there is nothing special to prove.

- Rule $\left[\dfrac{\mathfrak{q} : \Gamma \vdash \bot}{\mathfrak{p} : \Gamma \vdash A}\right]$:

  This is somehow the dual of the previous rule, because $\mathbb{W}(\bot) = \emptyset$, and therefore $\mathfrak{q}^\bullet = \emptyset$. Hence we need to construct a witness in $\mathbb{W}(A)$ out of nothing. Once again this is done thanks to the dummy term. Conversely, the $\mathfrak{p}_i^\circ$ translation is essentially the identity. We get:

$$\begin{array}{lcl} \mathfrak{p}^\bullet & := & \lambda\vec{x} : \mathbb{W}(\Gamma).\, \maltese_{\mathbb{W}(A)} \\ \mathfrak{p}_i^\circ & := & \lambda(\vec{x} : \mathbb{W}(\Gamma))\,(w : \mathbb{C}(A)).\, \mathfrak{q}_i^\circ\ \vec{x} \end{array}$$

  The realization relation is then, that for all $\vec{u} : \mathbb{W}(\Gamma)$ and $x : \mathbb{C}(A)$:

$$\Gamma_D[\vec{u}, \mathfrak{q}^\circ\ \vec{u}] \to A_D[\maltese_{\mathbb{W}(A)}, x]$$

  but from $\mathfrak{q}$ we get:

$$\Gamma_D[\vec{u}, \mathfrak{q}^\circ\ \vec{u}] \to \bot$$

  so we conclude by absurdity.

- Rule $\left[\dfrac{\mathfrak{q} : \Gamma \vdash A \qquad \mathfrak{r} : \Gamma \vdash B}{\mathfrak{p} : \Gamma \vdash A \wedge B}\right]$:

  This is another instance of a rule featuring duplication. The $\mathfrak{p}^\bullet$ is rather simple to describe: it amounts to packing the two proofs together. The $\mathfrak{p}_i^\circ$ is slightly more complicated because of the duplication at work, but it is handled elegantly thanks to the merge primitive. We obtain for the $\mathfrak{p}^\bullet$ component:

$$\begin{aligned} \mathfrak{p}^\bullet &: \mathbb{W}(\Gamma) \to \mathbb{W}(A)\,;\,\mathbb{W}(\Gamma) \to \mathbb{W}(B) \\ \mathfrak{p}^\bullet &:= \mathfrak{q}^\bullet\,;\,\mathfrak{r}^\bullet \end{aligned}$$

  and for the $\mathfrak{p}^\circ$ component:

$$\begin{aligned} \mathfrak{p}^\circ &: \mathbb{W}(\Gamma) \to \mathbb{C}(A) \to \mathbb{C}(B) \to \mathbb{C}(\Gamma) \\ \mathfrak{p}^\circ &:= \lambda(\vec{x} : \mathbb{W}(\Gamma))\,(w_l : \mathbb{C}(A))\,(w_r : \mathbb{C}(B)).\,\mathrm{merge}_\Gamma\ (\mathfrak{q}^\circ\ \vec{x}\ w_l)\ (\mathfrak{r}^\circ\ \vec{x}\ w_r) \end{aligned}$$

  We now need to prove that for all $\vec{u} : \mathbb{W}(\Gamma)$, $x : \mathbb{C}(A)$ and $y : \mathbb{C}(B)$ we have:

$$\Gamma_D[\vec{u}, \mathfrak{p}^\circ\ \vec{u}\ x\ y] \to A_D[\mathfrak{q}^\bullet\ \vec{u}, x] \wedge B_D[\mathfrak{r}^\bullet\ \vec{u}, y]$$

  which is convertible to:

$$\Gamma_D[\vec{u}, \mathrm{merge}_\Gamma\ (\mathfrak{q}^\circ\ \vec{u}\ x)\ (\mathfrak{r}^\circ\ \vec{u}\ y)\ \vec{u}] \to A_D[\mathfrak{q}^\bullet\ \vec{u}, x] \wedge B_D[\mathfrak{r}^\bullet\ \vec{u}, y]$$

  As usual, by the merge property, this is equivalent to:

$$\Gamma_D[\vec{u}, \mathfrak{q}^\circ\ \vec{u}\ x] \to \Gamma_D[\vec{u}, \mathfrak{r}^\circ\ \vec{u}\ y] \to A_D[\mathfrak{q}^\bullet\ \vec{u}, x] \wedge B_D[\mathfrak{r}^\bullet\ \vec{u}, y]$$

  We conclude by applying the induction hypotheses on $\mathfrak{q}$ and $\mathfrak{r}$, which give respectively:

$$\begin{aligned} \Gamma_D[\vec{u}, \mathfrak{q}^\circ\ \vec{u}\ x] &\to A_D[\mathfrak{q}^\bullet\ \vec{u}, x] \\ \Gamma_D[\vec{u}, \mathfrak{r}^\circ\ \vec{u}\ y] &\to B_D[\mathfrak{r}^\bullet\ \vec{u}, y] \end{aligned}$$

- Rule $\left[\dfrac{\mathfrak{q} : \Gamma \vdash A \wedge B}{\mathfrak{p} : \Gamma \vdash A}\right]$ (and its symmetric):

  This rule is straightforward. From

$$\mathfrak{q}^\bullet : \mathbb{W}(\Gamma) \to \mathbb{W}(A)\,;\,\mathbb{W}(\Gamma) \to \mathbb{W}(B)$$

  we can extract the first component $\mathfrak{q}^l$ and use it as the translation for $\mathfrak{p}^\bullet$:

$$\begin{aligned} \mathfrak{p}^\bullet &: \mathbb{W}(\Gamma) \to \mathbb{W}(A) \\ \mathfrak{p}^\bullet &:= \mathfrak{q}^l \end{aligned}$$

The $\mathfrak{p}^\circ_i$ component is built out of $\mathfrak{q}^\circ_i$ by feeding it with a dummy term on the right.

$$\begin{aligned} \mathfrak{p}^\circ_i &: \mathbb{W}(\Gamma) \to \mathbb{C}(A) \to \mathbb{C}(\Gamma_i) \\ \mathfrak{p}^\circ_i &:= \lambda(\vec{x} : \mathbb{W}(\Gamma))\,(w : \mathbb{C}(A)).\, \mathfrak{q}^\circ_i\ \vec{x}\ w\ \maltese_{\mathbb{C}(B)} \end{aligned}$$

We now need to show that for all $\vec{u} : \mathbb{W}(\Gamma)$ and all $x : \mathbb{C}(A)$,

$$\Gamma_D[\vec{u}, \mathfrak{p}^\circ\ \vec{u}\ x] \to A_D[\mathfrak{p}^\bullet\ \vec{u}, x]$$

which is convertible to:

$$\Gamma_D[\vec{u}, \mathfrak{q}^\circ\ \vec{u}\ x\ \maltese_{\mathbb{C}(B)}] \to A_D[\mathfrak{q}^l\ \vec{u}, x]$$

We conclude by using the induction hypothesis on $\mathfrak{q}$ applied to $\vec{u} : \mathbb{W}(\Gamma)$ and $(x\ ;\ \maltese_{\mathbb{C}(B)}) : \mathbb{C}(A \wedge B)$, which is indeed convertible to:

$$\Gamma_D[\vec{u}, \mathfrak{q}^\circ\ \vec{u}\ x\ \maltese_{\mathbb{C}(B)}] \to A_D[\mathfrak{q}^l\ \vec{u}, x] \wedge B_D[\mathfrak{q}^r\ \vec{u}, \maltese_{\mathbb{C}(B)}]$$

where $\mathfrak{q}^r$ is the second component of $\mathfrak{q}^\bullet$.

The symmetric rule is interpreted symmetrically.

- Rule $\left[\dfrac{\mathfrak{q} : \Gamma \vdash A}{\mathfrak{p} : \Gamma \vdash A \vee B}\right]$ (and its symmetric):

  Similarly to the duality between the interpretation of the rules for $\bot$ and $\top$, this rule is up to a certain point the dual of the previous one. Indeed, the $\mathfrak{p}^\bullet$ term must now provide a certain dummy term for the proof of $\mathbb{W}(A \vee B)$, while the $\mathfrak{p}^\circ_i$ is no more than a projection.

$$\begin{aligned} \mathfrak{p}^\bullet &: \mathbb{W}(\Gamma) \to \mathbb{N}\ ;\ \mathbb{W}(\Gamma) \to \mathbb{W}(A)\ ;\ \mathbb{W}(\Gamma) \to \mathbb{W}(B) \\ \mathfrak{p}^\bullet &:= \lambda\vec{x} : \mathbb{W}(\Gamma).\, (\mathrm{false}\ ;\ \mathfrak{q}^\bullet\ \vec{x}\ ;\ \maltese_{\mathbb{W}(B)}) \end{aligned}$$

$$\begin{aligned} \mathfrak{p}^\circ_i &: \mathbb{W}(\Gamma) \to \mathbb{C}(A) \to \mathbb{C}(B) \to \mathbb{C}(\Gamma_i) \\ \mathfrak{p}^\circ_i &:= \lambda(\vec{x} : \mathbb{W}(\Gamma))\,(w_l : \mathbb{C}(A))\,(w_r : \mathbb{C}(B)).\, \mathfrak{q}^\circ_i\ \vec{x}\ w_l \end{aligned}$$

We have to prove that for any $\vec{u} : \mathbb{W}(\Gamma)$, $x : \mathbb{C}(A)$ and $y : \mathbb{C}(B)$,

$$\Gamma_D[\vec{u}, \mathfrak{p}^\circ\ \vec{u}\ x\ y] \to (A \vee B)_D[\mathfrak{p}^\bullet\ \vec{u}, x\ ;\ y]$$

which is convertible to:

$$\Gamma_D[\vec{u}, \mathfrak{q}^\circ\ \vec{u}\ x] \to (0 = 0 \wedge A_D[\mathfrak{q}^\bullet\ \vec{u}, x]) \vee (0 = 1 \wedge B_D[✠_{\mathbb{W}(B)}, y])$$

Obviously, we will be proving the left formula, so that the following remains to be proved:

$$\Gamma_D[\vec{u}, \mathfrak{q}^\circ\ \vec{u}\ x] \to A_D[\mathfrak{q}^\bullet\ \vec{u}, x]$$

but this is precisely the induction hypothesis for $\mathfrak{q}$, from which we conclude.

The symmetric rule is also obtained by symmetry.

- Rule $\left[\dfrac{\mathfrak{q} : \Gamma \vdash A \vee B \qquad \mathfrak{r} : \Gamma, A \vdash C \qquad \mathfrak{s} : \Gamma, B \vdash C}{\mathfrak{p} : \Gamma \vdash C}\right]$:

  This is one of the most intricate cases, as it features two unrelated phenomena, one being the duplication of hypotheses (which entails a merge) and the other being the antique encoding of sum types through dummy values.

  The $\mathfrak{p}^\bullet$ translation is recovered by making a case analysis on the pseudo-boolean contained in the proof $\mathfrak{q}^\bullet$ and emulating a pattern matching on the encoded sum. First, we give a name to the several components of

$$\mathfrak{q}^\bullet \quad : \quad \mathbb{W}(\Gamma) \to \mathbb{N}\ ;\ \mathbb{W}(\Gamma) \to \mathbb{W}(A)\ ;\ \mathbb{W}(\Gamma) \to \mathbb{W}(B)$$

  as follows:

$$\begin{aligned} \mathfrak{q}^b \quad &: \quad \mathbb{W}(\Gamma) \to \mathbb{N} \\ \mathfrak{q}^l \quad &: \quad \mathbb{W}(\Gamma) \to \mathbb{W}(A) \\ \mathfrak{q}^r \quad &: \quad \mathbb{W}(\Gamma) \to \mathbb{W}(B) \end{aligned}$$

  We can then write out the pseudo-code explained above just as:

$$\begin{aligned} \mathfrak{p}^\bullet \quad &: \quad \mathbb{W}(\Gamma) \to \mathbb{W}(C) \\ \mathfrak{p}^\bullet \quad &:= \quad \lambda\vec{x} : \mathbb{W}(\Gamma).\ \mathrm{ifz}\ (\mathfrak{q}^b\ \vec{x})\ (\mathfrak{r}^\bullet\ \vec{x}\ (\mathfrak{q}^l\ \vec{x}))\ (\mathfrak{s}^\bullet\ \vec{x}\ (\mathfrak{q}^r\ \vec{x})) \end{aligned}$$

  with the already mentioned abuse of sequence notation for ifz.

  The $\mathfrak{p}^\circ$ translation does a similar trick, but also uses merge in addition, dynamically choosing to merge from $\mathfrak{r}$ or $\mathfrak{s}$ according to the value of the boolean. From $\mathfrak{r}^\circ$ and $\mathfrak{s}^\circ$, we can extract respectively:

$$\begin{aligned}
\mathfrak{r}^\circ_\Gamma &: \mathbb{W}(\Gamma) \to \mathbb{W}(A) \to \mathbb{C}(C) \to \mathbb{C}(\Gamma) \\
\mathfrak{r}^\circ_A &: \mathbb{W}(\Gamma) \to \mathbb{W}(A) \to \mathbb{C}(C) \to \mathbb{C}(A)
\end{aligned}$$

$$\begin{aligned}
\mathfrak{s}^\circ_\Gamma &: \mathbb{W}(\Gamma) \to \mathbb{W}(B) \to \mathbb{C}(C) \to \mathbb{C}(\Gamma) \\
\mathfrak{s}^\circ_B &: \mathbb{W}(\Gamma) \to \mathbb{W}(B) \to \mathbb{C}(C) \to \mathbb{C}(B)
\end{aligned}$$

There are now three different ways to recover a counter $\mathbb{C}(\Gamma)$, one from $\mathfrak{q}^\circ$, one from $\mathfrak{r}^\circ_\Gamma$ and one from $\mathfrak{s}^\circ_\Gamma$, which are given below.

$$\begin{aligned}
\mathfrak{c} &: \mathbb{W}(\Gamma) \to \mathbb{C}(C) \to \mathbb{C}(\Gamma) \\
\mathfrak{c} &:= \lambda(\vec{x} : \mathbb{W}(\Gamma))\,(w : \mathbb{C}(C)).\, \mathfrak{q}^\circ\ \vec{x}\ (\mathfrak{r}^\circ_A\ \vec{x}\ (\mathfrak{q}^l\ \vec{x})\ w)\ (\mathfrak{s}^\circ_B\ \vec{x}\ (\mathfrak{q}^r\ \vec{x})\ w)
\end{aligned}$$

$$\begin{aligned}
\mathfrak{c}_l &: \mathbb{W}(\Gamma) \to \mathbb{C}(C) \to \mathbb{C}(\Gamma) \\
\mathfrak{c}_l &:= \lambda(\vec{x} : \mathbb{W}(\Gamma))\,(w : \mathbb{C}(C)).\, \mathfrak{r}^\circ_\Gamma\ \vec{x}\ (\mathfrak{q}^l\ \vec{x})
\end{aligned}$$

$$\begin{aligned}
\mathfrak{c}_r &: \mathbb{W}(\Gamma) \to \mathbb{C}(C) \to \mathbb{C}(\Gamma) \\
\mathfrak{c}_r &:= \lambda(\vec{x} : \mathbb{W}(\Gamma))\,(w : \mathbb{C}(C)).\, \mathfrak{s}^\circ_\Gamma\ \vec{x}\ (\mathfrak{q}^r\ \vec{x})
\end{aligned}$$

We finally recover our interpreted term by choosing which one of the $\mathfrak{c}_l$ or $\mathfrak{c}_l$ we choose to use, according to which side of $\mathfrak{q}$ is proved, i.e. whether the boolean $\mathfrak{q}^b$ is true or false.

$$\begin{aligned}
\mathfrak{p}^\circ &: \mathbb{W}(\Gamma) \to \mathbb{C}(C) \to \mathbb{C}(\Gamma) \\
\mathfrak{p}^\circ &:= \lambda(\vec{x} : \mathbb{W}(\Gamma))\,(w : \mathbb{C}(C)).\, \mathrm{merge}_\Gamma\ (\mathfrak{c}\ \vec{x}\ w)\ (\mathrm{ifz}\ (\mathfrak{q}^b\ \vec{x})\ (\mathfrak{c}_l\ \vec{x}\ w)\ (\mathfrak{c}_r\ \vec{x}\ w))\ \vec{x}
\end{aligned}$$

Note that once again, we abused the notation of sequences for ifz.

Let us not be afraid of the apparent difficulty of such definitions and let us prove that those terms realize the formula indeed. To this end, assume $\vec{u} : \mathbb{W}(\Gamma)$ and $z : \mathbb{W}(C)$, we must show as usual that

$$\Gamma_D[\vec{u}, \mathfrak{p}^\circ\ \vec{u}\ z] \to C_D[\mathfrak{p}^\bullet\ \vec{u}, z]$$

Unfolding $\mathfrak{p}^\circ$ and applying the equivalence property of merge, this turns out to be equivalent to

$$\Gamma_D[\vec{u}, \mathfrak{c}\ \vec{u}\ z] \to \Gamma_D[\vec{u}, \mathrm{ifz}\ (\mathfrak{q}^b\ \vec{u})\ (\mathfrak{c}_l\ \vec{u}\ z)\ (\mathfrak{c}_r\ \vec{u}\ z)] \to C_D[\mathfrak{p}^\bullet\ \vec{u}, z]$$

Here, we have in particular

$$\mathfrak{c}\ \vec{u}\ z \equiv_\beta \mathfrak{q}^\circ\ \vec{u}\ (\mathfrak{r}^\circ_A\ \vec{u}\ (\mathfrak{q}^l\ \vec{u})\ z)\ (\mathfrak{s}^\circ_B\ \vec{u}\ (\mathfrak{q}^r\ \vec{u})\ z)$$

so we can apply the induction hypothesis for $\mathfrak{q}$ on the rightmost assumption of the formula we have to prove. Therefore, we only have to prove

$$\begin{array}{ll}(A \vee B)_D[\mathfrak{q}^\bullet\ \vec{u}, (\mathfrak{r}^\circ_A\ \vec{u}\ (\mathfrak{q}^l\ \vec{u})\ z\ ;\ \mathfrak{s}^\circ_B\ \vec{u}\ (\mathfrak{q}^r\ \vec{u})\ z)] & \to \\ \Gamma_D[\vec{u}, \mathrm{ifz}\ (\mathfrak{q}^b\ \vec{u})\ (\mathfrak{c}_l\ \vec{u}\ z)\ (\mathfrak{c}_r\ \vec{u}\ z)] & \to \\ C_D[\mathfrak{p}^\bullet\ \vec{u}, z] & \end{array}$$

which is convertible to

$$\begin{array}{ll}(\mathfrak{q}^b\ \vec{u} = 0 \wedge A_D[\mathfrak{q}^l\ \vec{u}, \mathfrak{r}^\circ_A\ \vec{u}\ (\mathfrak{q}^l\ \vec{u})\ z]) \vee (\mathfrak{q}^b\ \vec{u} = 1 \wedge B_D[\mathfrak{q}^r\ \vec{u}, \mathfrak{s}^\circ_B\ \vec{u}\ (\mathfrak{q}^r\ \vec{u})\ z]) & \to \\ \Gamma_D[\vec{u}, \mathrm{ifz}\ (\mathfrak{q}^b\ \vec{u})\ (\mathfrak{c}_l\ \vec{u}\ z)\ (\mathfrak{c}_r\ \vec{u}\ z)] & \to \\ C_D[\mathrm{ifz}\ (\mathfrak{q}^b\ \vec{u})\ (\mathfrak{r}^\bullet\ \vec{u}\ (\mathfrak{q}^l\ \vec{u}))\ (\mathfrak{s}^\bullet\ \vec{u}\ (\mathfrak{q}^r\ \vec{u})), z] & \end{array}$$

There are now two different cases. Either $\mathfrak{q}^b\ \vec{u} \equiv 0$ or not, which corresponds to the two branches we would have in a proper pattern matching on a sum.

Assume first that $\mathfrak{q}^b\ \vec{u} \equiv 0$. In that case, we can convert and simplify the whole formula, by eliminating the obviously false branch of the $\vee$ and making some reordering, to

$$\Gamma_D[\vec{u}, \mathfrak{r}^\circ_\Gamma\ \vec{u}\ (\mathfrak{q}^l\ \vec{u})] \to A_D[\mathfrak{q}^l\ \vec{u}, \mathfrak{r}^\circ_A\ \vec{u}\ (\mathfrak{q}^l\ \vec{u})\ z] \to C_D[\mathfrak{r}^\bullet\ \vec{u}\ (\mathfrak{q}^l\ \vec{u}), z]$$

but this is actually exactly the induction hypothesis of $\mathfrak{r}$ applied to $(\vec{u}\ ;\ \mathfrak{q}^l\ \vec{u}) : \mathbb{W}(\Gamma\ ;\ A)$ and $z : \mathbb{C}(C)$. Hence we are done.

The other case $\mathfrak{q}^b\ \vec{u} \not\equiv 0$ is treated the same, choosing the other branch and using $\mathfrak{s}$ instead of $\mathfrak{r}$.

First-order First-order rules manipulate variables of **HA** formulae, so there is special care to be taken in order not to create wild free variables.

- Rule $\left[\dfrac{\mathfrak{q} : \Gamma \vdash A}{\mathfrak{p} : \Gamma \vdash \forall z.\, A}\right]$:

  This rule is almost transparent computationally, because the translation only binds a variable that was free beforehand. Indeed, z is free in A, so the proofterms derived from $\mathfrak{q}$ may mention it and therefore we have to bind it. This is why we willingly use the same name for the term binder below and for the binder of the quantifier in the rule above. The translation goes as follows.

$$\begin{aligned}
\mathfrak{p}^\bullet &: \mathbb{W}(\Gamma) \to \mathbb{N} \to \mathbb{W}(A) \\
\mathfrak{p}^\bullet &:= \lambda(\vec{x} : \mathbb{W}(\Gamma))\,(z : \mathbb{N}).\, \mathfrak{q}^\bullet\ \vec{x}
\end{aligned}$$

$$\begin{aligned}
\mathfrak{p}_i^\circ &: \mathbb{W}(\Gamma) \to \mathbb{N} \to \mathbb{C}(A) \to \mathbb{C}(\Gamma_i) \\
\mathfrak{p}_i^\circ &:= \lambda(\vec{x} : \mathbb{W}(\Gamma))\,(z : \mathbb{N})\,(w : \mathbb{C}(A)).\, \mathfrak{q}_i^\circ\ \vec{x}\ w
\end{aligned}$$

We shall now prove that for all $\vec{u} : \mathbb{W}(\Gamma)$, $z : \mathbb{N}$ and $x : \mathbb{C}(A)$, the following formula holds.

$$\Gamma_D[\vec{u}, \mathfrak{p}^\circ\ \vec{u}\ (z\ ;\ x)] \to (\forall z.\, A)_D[\mathfrak{p}^\bullet\ \vec{u}, (z\ ;\ x)]$$

This formula is actually convertible to:

$$\Gamma_D[\vec{u}, \mathfrak{q}^\circ\ \vec{u}\ x] \to A_D[\mathfrak{q}^\bullet\ \vec{u}, x]$$

where the variable z has been silently substituted in $\mathfrak{q}$ and A. But this is exactly the realization property of $\mathfrak{q}$. So the translation realizes the formula.

- Rule $\left[\dfrac{\mathfrak{q} : \Gamma \vdash \forall z.\, A}{\mathfrak{p} : \Gamma \vdash A[z := t]}\right]$:

  For the same reasons as the previous rule, the translated terms are here almost transparent. The one difference lies in the fact that instead of binding a free variable, we must provide a term of the corresponding type to the resulting proofterms. All is well because we precisely have the t term lurking around.

$$\begin{aligned}
\mathfrak{p}^\bullet &: \mathbb{W}(\Gamma) \to \mathbb{W}(A) \\
\mathfrak{p}^\bullet &:= \lambda\vec{x} : \mathbb{W}(\Gamma).\, \mathfrak{q}^\bullet\ \vec{x}\ t
\end{aligned}$$

$$\begin{aligned}
\mathfrak{p}_i^\circ &: \mathbb{W}(\Gamma) \to \mathbb{C}(A) \to \mathbb{C}(\Gamma_i) \\
\mathfrak{p}_i^\circ &:= \lambda(\vec{x} : \mathbb{W}(\Gamma))\,(w : \mathbb{C}(A)).\, \mathfrak{q}_i^\circ\ \vec{x}\ t\ w
\end{aligned}$$

Let $\vec{u} : \mathbb{W}(\Gamma)$ and $x : \mathbb{C}(A)$, we must show that

$$\Gamma_D[\vec{u}, \mathfrak{p}^\circ\ \vec{u}\ x] \to (A[z := t])_D[\mathfrak{p}^\bullet\ \vec{u}, x]$$

which amounts to

$$\Gamma_D[\vec{u}, \mathfrak{q}^\circ\ \vec{u}\ t\ x] \to (A[z := t])_D[\mathfrak{q}^\bullet\ \vec{u}\ t, x]$$

This is the induction hypothesis applied to $\mathfrak{q}$ with $\vec{u} : \mathbb{W}(\Gamma)$ and $(t\ ;\ x) : \mathbb{C}(\forall z.\, A)$. This allows us to conclude immediately.

- Rule $\left[\dfrac{\mathfrak{q} : \Gamma \vdash A[z := t]}{\mathfrak{p} : \Gamma \vdash \exists z.\, A}\right]$:

  The introduction of the existential quantifier is interpreted similarly to the previous rule, for duality reasons. As $\mathbb{W}(\exists z.\, A) := \mathbb{N}\,;\,\mathbb{W}(A)$, we split the $\mathfrak{p}^\bullet$ translation into the two following terms

$$\begin{aligned} \mathfrak{p}^w &: \mathbb{W}(\Gamma) \to \mathbb{N} \\ \mathfrak{p}^w &:= \lambda\_ : \mathbb{W}(\Gamma).\, t \end{aligned}$$

$$\begin{aligned} \mathfrak{p}^p &: \mathbb{W}(\Gamma) \to \mathbb{W}(A) \\ \mathfrak{p}^p &:= \mathfrak{q}^\bullet \end{aligned}$$

  Note that the proof term corresponding to the integer witnessed does not depend on the computational content of the logical part of the derivation. The resulting translation is then:

$$\begin{aligned} \mathfrak{p}^\bullet &: \mathbb{W}(\Gamma) \to \mathbb{N}\,;\,\mathbb{W}(\Gamma) \to \mathbb{W}(A) \\ \mathfrak{p}^\bullet &:= \mathfrak{p}^w\,;\,\mathfrak{p}^p \end{aligned}$$

$$\begin{aligned} \mathfrak{p}_i^\circ &: \mathbb{W}(\Gamma) \to (\mathbb{N} \to \mathbb{C}(A)) \to \mathbb{C}(\Gamma_i) \\ \mathfrak{p}_i^\circ &:= \lambda(\vec{x} : \mathbb{W}(\Gamma))\,(w : \mathbb{N} \to \mathbb{C}(A)).\, \mathfrak{q}_i^\circ\ \vec{x}\ (w\ t) \end{aligned}$$

  We must show that for all $\vec{u} : \mathbb{W}(\Gamma)$ and $x : \mathbb{N} \to \mathbb{C}(A)$, the following holds

$$\Gamma_D[\vec{u}, \mathfrak{p}^\circ\ \vec{u}\ x] \to (\exists z.\, A)_D[\mathfrak{p}^\bullet\ \vec{u}, x]$$

  which is, by unfolding, exactly the same as

$$\Gamma_D[\vec{u}, \mathfrak{q}^\circ\ \vec{u}\ (w\ t)] \to (A[z := t])_D[\mathfrak{q}^\bullet\ \vec{u}, x\ t]$$

  that is to say, the induction hypothesis from $\mathfrak{q}$, from which we conclude.

- Rule $\left[\dfrac{\mathfrak{q} : \Gamma \vdash \exists z.\, A \qquad \mathfrak{r} : \Gamma, A \vdash B}{\mathfrak{p} : \Gamma \vdash B}\right]$:

  The complexity of this case is similar to the one of the elimination rule for the $\vee$ connective. Here, we need to handle at the same time duplication and capture of free variables. As in the previous rule, let us first give a name to the two components of

$$\mathfrak{q}^\bullet : \mathbb{W}(\Gamma) \to \mathbb{N}\,;\,\mathbb{W}(\Gamma) \to \mathbb{W}(A)$$

  as

$$\begin{aligned} \mathfrak{q}^w &: \mathbb{W}(\Gamma) \to \mathbb{N} \\ \mathfrak{q}^p &: \mathbb{W}(\Gamma) \to \mathbb{W}(A) \end{aligned}$$

In order to construct $\mathfrak{q}^\bullet$, we are going to use $\mathfrak{r}^\bullet$ because it produces a $\mathbb{W}(B)$, and feed the missing $\mathbb{W}(A)$ thanks to $\mathfrak{q}^p$. Be careful though that z is free in $\mathfrak{r}$, so that we need to eliminate it. This will be done thanks to $\mathfrak{q}^w$. We finally obtain:

$$\begin{aligned} \mathfrak{p}^\bullet &: \mathbb{W}(\Gamma) \to \mathbb{W}(B) \\ \mathfrak{p}^\bullet &:= \lambda \vec{x} : \mathbb{W}(\Gamma).\, (\lambda z.\, \mathfrak{r}^\bullet\ \vec{x}\ (\mathfrak{q}^p\ \vec{x}))\ (\mathfrak{q}^w\ \vec{x}) \end{aligned}$$

The $\mathfrak{p}^\circ$ translation is constructed by taking care of merging the two sets of hypotheses coming respectively from $\mathfrak{q}$ and $\mathfrak{r}$. Let us recall that we can split $\mathfrak{r}^\circ$ in two parts typed as:

$$\begin{aligned} \mathfrak{r}^\circ_\Gamma &: \mathbb{W}(\Gamma) \to \mathbb{W}(A) \to \mathbb{C}(B) \to \mathbb{C}(\Gamma) \\ \mathfrak{r}^\circ_A &: \mathbb{W}(\Gamma) \to \mathbb{W}(A) \to \mathbb{C}(B) \to \mathbb{C}(A) \end{aligned}$$

We can now build the two sequences of $\mathbb{C}(\Gamma)$ counters as:

$$\begin{aligned} \mathfrak{c}_l &: \mathbb{W}(\Gamma) \to \mathbb{C}(B) \to \mathbb{C}(\Gamma) \\ \mathfrak{c}_l &:= \lambda(\vec{x} : \mathbb{W}(\Gamma))\,(w : \mathbb{C}(B)).\, \mathfrak{q}^\circ\ \vec{x}\ (\lambda z.\, \mathfrak{r}^\circ_A\ \vec{x}\ (\mathfrak{q}^p\ \vec{x})\ w) \end{aligned}$$

$$\begin{aligned} \mathfrak{c}_r &: \mathbb{W}(\Gamma) \to \mathbb{C}(B) \to \mathbb{C}(\Gamma) \\ \mathfrak{c}_r &:= \lambda(\vec{x} : \mathbb{W}(\Gamma))\,(w : \mathbb{C}(B)).\, (\lambda z.\, \mathfrak{r}^\circ_\Gamma\ \vec{x}\ (\mathfrak{q}^p\ \vec{x})\ w)\ (\mathfrak{q}^w\ \vec{x}) \end{aligned}$$

We finally get:

$$\begin{aligned} \mathfrak{p}^\circ &: \mathbb{W}(\Gamma) \to \mathbb{C}(B) \to \mathbb{C}(\Gamma) \\ \mathfrak{p}^\circ &:= \lambda(\vec{x} : \mathbb{W}(\Gamma))\,(w : \mathbb{C}(B)).\, \mathrm{merge}_\Gamma\ (\mathfrak{c}_l\ \vec{x}\ w)\ (\mathfrak{c}_r\ \vec{x}\ w) \end{aligned}$$

Let us now prove that these terms realize the formula indeed. For this, we need to prove that for any $\vec{u} : \mathbb{W}(\Gamma)$ and $y : \mathbb{C}(B)$, the following holds.

$$\Gamma_D[\vec{u}, \mathfrak{p}^\circ\ \vec{u}\ y] \to B_D[\mathfrak{p}^\bullet\ \vec{u}, y]$$

It is equivalent, by applying the merge property, to

$$\Gamma_D[\vec{u}, \mathfrak{c}_l\ \vec{u}\ y] \to \Gamma_D[\vec{u}, \mathfrak{c}_r\ \vec{u}\ y] \to B_D[\mathfrak{p}^\bullet\ \vec{u}, y]$$

If we unfold $\mathfrak{c}_l$ and apply the realization property for $\mathfrak{q}$ on the left hypothesis, we recover:

$$(\exists z.\, A)_D[\mathfrak{q}^\bullet\ \vec{u}, \lambda z.\, \mathfrak{r}_A^\circ\ \vec{u}\ (\mathfrak{q}^p\ \vec{u})\ y] \to \Gamma_D[\vec{u}, \mathfrak{c}_r\ \vec{u}\ y] \to B_D[\mathfrak{p}^\bullet\ \vec{u}, y]$$

that is, by unfolding $(\exists z.\, A)_D$:

$$(A_D[\mathfrak{q}^p\ \vec{u}, \mathfrak{r}_A^\circ\ \vec{u}\ (\mathfrak{q}^p\ \vec{u})\ y])[z := \mathfrak{q}^w\ \vec{u}] \to \Gamma_D[\vec{u}, \mathfrak{c}_r\ \vec{u}\ y] \to B_D[\mathfrak{p}^\bullet\ \vec{u}, y]$$

and after subsequent unfolding of $\mathfrak{c}_r$ and $\mathfrak{p}^\bullet$, this is the same as:

$$A_D[\mathfrak{q}^p\ \vec{u}, \mathfrak{r}_A^\circ\ \vec{u}\ (\mathfrak{q}^p\ \vec{u})\ y] \to \Gamma_D[\vec{u}, \mathfrak{r}_\Gamma^\circ\ \vec{u}\ (\mathfrak{q}^p\ \vec{u})\ y] \to B_D[\mathfrak{r}^\bullet\ \vec{u}\ (\mathfrak{q}^p\ \vec{u}), y]$$

where all occurrences of z in A_D, $\mathfrak{r}_\Gamma^\circ$, $\mathfrak{r}_A^\circ$ and $\mathfrak{r}^\bullet$ have been substituted by $\mathfrak{q}^w\ \vec{u}$.

But, up to the reordering of hypotheses, this is the realizability property of the $\mathfrak{r}$ term applied to $(\vec{u}\ ;\ \mathfrak{q}^p\ \vec{u}) : \mathbb{W}(\Gamma\ ;\ A)$ and $y : \mathbb{C}(B)$. Thus we can conclude.

Arithmetic Most of arithmetic rules are equalities, which are erased by the type translation. This means that proof terms for equalities are empty and, as it often occurs with realizability, soundness is externalized in the metatheory. We only treat in detail one case of an equality axiom, as all the other ones are just identical.

- Rule $\left[\dfrac{}{\mathfrak{p} : \Gamma \vdash t = t}\right]$:

  As explained above, $\mathbb{W}(t = t) := \emptyset$, so the $\mathfrak{p}^\bullet$ sequence is necessarily empty. The reverse translation $\mathfrak{p}^\circ$ is, as usual, built out of dummy terms. Note that $\mathbb{C}(t = t) := \emptyset$, but as in the $\top$ introduction rule, we choose to leave an empty sequence of variables to respect the same presentation as in the other rules.

$$\begin{aligned} \mathfrak{p}^\bullet &: \mathbb{W}(\Gamma) \to \mathbb{W}(t = t) \\ \mathfrak{p}^\bullet &:= \emptyset \end{aligned}$$

$$\begin{aligned} \mathfrak{p}_i^\circ &: \mathbb{W}(\Gamma) \to \mathbb{C}(t = t) \to \mathbb{C}(\Gamma_i) \\ \mathfrak{p}_i^\circ &:= \lambda(\vec{x} : \mathbb{W}(\Gamma))\,(w : \mathbb{C}(t = t)).\, \maltese_{\mathbb{C}(\Gamma_i)} \end{aligned}$$

In order to prove the soundness of these realizers, we only have to show that for all $\vec{u} : \mathbb{W}(\Gamma)$:

$$\Gamma_D[\vec{u}, \mathfrak{p}^\circ\ \vec{u}] \to (t = t)_D[\mathfrak{p}^\bullet\ \vec{u}, \emptyset]$$

which is convertible to:

$$\Gamma_D[\vec{u}, \maltese_{\mathbb{C}(\Gamma)}] \to t = t$$

which is itself obtained by reflexivity in $\mathbf{HA} + T$.

- Rule $\left[\dfrac{\mathfrak{q} : \Gamma \vdash t_1 = t_2 \qquad \mathfrak{r} : \Gamma \vdash A[z := t_2]}{\mathfrak{p} : \Gamma \vdash A[z := t_1]}\right]$:
  The substitution rule is almost transparent, because witness and counter types are insensitive to substitution. Yet, in order to be able to recover the equality coming from $\mathfrak{q}$, we need to be able to prove its assumptions. The only way to do this is by using the merge primitive. This necessity will be obvious in the proof of realization.

$$\begin{aligned} \mathfrak{p}^\bullet &: \mathbb{W}(\Gamma) \to \mathbb{W}(A) \\ \mathfrak{p}^\bullet &:= \mathfrak{r}^\bullet \end{aligned}$$

$$\begin{aligned} \mathfrak{p}^\circ &: \mathbb{W}(\Gamma) \to \mathbb{C}(A) \to \mathbb{C}(\Gamma) \\ \mathfrak{p}^\circ &:= \lambda(\vec{x} : \mathbb{W}(\Gamma))\,(w : \mathbb{C}(A)).\, \mathrm{merge}_\Gamma\ (\mathfrak{r}^\circ\ \vec{x}\ w)\ (\mathfrak{q}^\circ\ \vec{x})\ \vec{x} \end{aligned}$$

  Let us show the realization property. We must prove that for all $\vec{u} : \mathbb{W}(\Gamma)$ and $x : \mathbb{C}(A)$, the following holds:

$$\Gamma_D[\vec{u}, \mathfrak{p}^\circ\ \vec{u}\ x] \to (A[z := t_1])_D[\mathfrak{p}^\bullet\ \vec{u}, x]$$

  Unfolding definitions and using the property of merge, this is equivalent to:

$$\Gamma_D[\vec{u}, \mathfrak{r}^\circ\ \vec{u}\ x] \to \Gamma_D[\vec{u}, \mathfrak{q}^\circ\ \vec{u}] \to (A[z := t_1])_D[\mathfrak{r}^\bullet\ \vec{u}, x]$$

  We can now apply the hypothesis on $\mathfrak{r}$ to the first assumption, and on $\mathfrak{q}$ to the second assumption, so that we now have to prove:

$$(A[z := t_2])_D[\mathfrak{r}^\circ\ \vec{u}, x] \to t_1 = t_2 \to (A[z := t_1])_D[\mathfrak{r}^\circ\ \vec{u}, x]$$

  but as z is fresh, this is equivalent to:

$$(A_D[\mathfrak{r}^\circ\ \vec{u}, x])[z := t_2] \to t_1 = t_2 \to (A_D[\mathfrak{r}^\circ\ \vec{u}, x])[z := t_1]$$

  which we prove by rewriting the given equality.

- Rule $\left[\dfrac{\mathfrak{q} : \Gamma \vdash A[z := \mathbf{0}] \qquad \mathfrak{r} : \Gamma, A \vdash A[z := \mathbf{S}\, z]}{\mathfrak{p} : \Gamma \vdash A[z := t]}\right]$:
  This rule is by far one of the most complicated to translate. We need to handle at the same time recursion, opening of a fresh variable and duplication of hypotheses. The global picture is as follows: $\mathfrak{p}^\bullet$ and $\mathfrak{p}^\circ$ are going to repeat the computational behaviour of the underlying induction through the recursor, but while $\mathfrak{p}^\bullet$ is really a simple extraction, $\mathfrak{p}^\circ$ needs to keep track of duplication, leading to the use of a merge at each recursion step. We need to perform both steps at once, so that

we define $\mathfrak{p}^\bullet$ and $\mathfrak{p}^\circ$ at the same time. The definition makes use of a generalized recursor that works on sequences rather than plain System T types, which is as usual defined in a straightforward manner.

We will not discuss too much the intuitions here, and wait for the realizability proof to observe why the translation below is the only sensible one. As done before, we split $\mathfrak{r}^\circ$ in two parts typed as

$$\begin{aligned} \mathfrak{r}^\circ_\Gamma &: \mathbb{W}(\Gamma) \to \mathbb{W}(A) \to \mathbb{C}(A) \to \mathbb{C}(\Gamma) \\ \mathfrak{r}^\circ_A &: \mathbb{W}(\Gamma) \to \mathbb{W}(A) \to \mathbb{C}(A) \to \mathbb{C}(A) \end{aligned}$$

from which we can built the desired term as follows.

$$\begin{aligned} \mathfrak{p}^\bullet\,;\mathfrak{p}^\circ \quad &: \quad \mathbb{W}(\Gamma) \to (\mathbb{W}(A)\,;(\mathbb{C}(A) \to \mathbb{C}(\Gamma))) \\ \mathfrak{p}^\bullet\,;\mathfrak{p}^\circ \quad &:= \quad \lambda\vec{x} : \mathbb{W}(\Gamma).\,\mathbf{rec}\ t\ ((\mathfrak{q}^\bullet\,;\mathfrak{q}^\circ)\ \vec{x}) \\ &\qquad (\lambda z\,y\,f.\,(\mathfrak{r}^\bullet\ \vec{x}\ y\,;\lambda w.\,\mathrm{merge}_\Gamma\ (f\ (\mathfrak{r}^\circ_A\ \vec{x}\ y\ w))\ (\mathfrak{r}^\circ_\Gamma\ \vec{x}\ y\ w)\ \vec{x})) \end{aligned}$$

Remark how we chose carefully to name the bound variable z in the recursive call, so as to match the corresponding free variable of $\mathfrak{r}$.

We must now prove that for all $\vec{u} : \mathbb{W}(\Gamma)$ and $x : \mathbb{C}(A)$, the following holds:

$$\Gamma_D[\vec{u}, \mathfrak{p}^\circ\ \vec{u}\ x] \to (A[z := t])_D[\mathfrak{p}^\bullet\ \vec{u}, x]$$

and we do this by induction over t, which is a natural number.

To ease the understanding of the reduction of the translated proof, we need to parameterize $\mathfrak{p}^\bullet\,;\mathfrak{p}^\circ$ by the actual natural number t it is applied to. Let us write ${\mathfrak{p}^\bullet}_z$ and ${\mathfrak{p}^\circ}_z$ for the corresponding terms where t is some variable z. Then we have the following System T equivalences.

$$\begin{aligned} {\mathfrak{p}^\bullet}_{\mathbf{0}}\ \vec{u} &\equiv_\beta \mathfrak{q}^\bullet\ \vec{u} \\ {\mathfrak{p}^\circ}_{\mathbf{0}}\ \vec{u}\ x &\equiv_\beta \mathfrak{q}^\circ\ \vec{u}\ x \\ {\mathfrak{p}^\bullet}_{(\mathbf{S}\,z)}\ \vec{u} &\equiv_\beta \mathfrak{r}^\bullet\ \vec{u}\ ({\mathfrak{p}^\bullet}_z\ \vec{u}) \\ {\mathfrak{p}^\circ}_{(\mathbf{S}\,z)}\ \vec{u}\ x &\equiv_\beta \mathrm{merge}_\Gamma\ ({\mathfrak{p}^\circ}_z\ \vec{u}\ (\mathfrak{r}^\circ_A\ \vec{u}\ ({\mathfrak{p}^\bullet}_z\ \vec{u})\ x))\ (\mathfrak{r}^\circ_\Gamma\ \vec{u}\ ({\mathfrak{p}^\bullet}_z\ \vec{u})\ x)\ \vec{u} \end{aligned}$$

By applying these equivalences, we can greatly simplify the recursive proof of our realization property. We look at each case separatly.

Assume first that $t = \mathbf{0}$. Then we only have to prove that

$$\Gamma_D[\vec{u}, \mathfrak{q}^\circ\ \vec{u}\ x] \to (A[z := \mathbf{0}])_D[\mathfrak{q}^\bullet\ \vec{u}, x]$$

which is simply the realization hypothesis coming from $\mathfrak{q}$.

Now, assume that $t = \mathbf{S}\, z$, and in addition that the realization property holds for ${\mathfrak{p}^\bullet}_z$ and ${\mathfrak{p}^\circ}_z$. We must show the following.

$$\Gamma_D[\vec{u}, {\mathfrak{p}^\circ}_{(\mathbf{S}\, z)}\ \vec{u}\ x] \to (A[z := \mathbf{S}\, z])_D[\mathfrak{r}^\bullet\ \vec{u}\ ({\mathfrak{p}^\bullet}_z\ \vec{u}), x]$$

Applying the merge property results in the following proposition.

$$\Gamma_D[\vec{u}, {\mathfrak{p}^\circ}_z\ \vec{u}\ (\mathfrak{r}^\circ_A\ \vec{u}\ ({\mathfrak{p}^\bullet}_z\ \vec{u})\ x)] \to \Gamma_D[\vec{u}, \mathfrak{r}^\circ_\Gamma\ \vec{u}\ ({\mathfrak{p}^\bullet}_z\ \vec{u})\ x] \to (A[z := \mathbf{S}\, z])_D[\mathfrak{r}^\bullet\ \vec{u}\ ({\mathfrak{p}^\bullet}_z\ \vec{u}), x]$$

The realization property for z coming from the induction hypothesis can be applied to the leftmost hypothesis, leading to the formula

$$A_D[{\mathfrak{p}^\bullet}_z\ \vec{u}, \mathfrak{r}^\circ_A\ \vec{u}\ ({\mathfrak{p}^\bullet}_z\ \vec{u})\ x] \to \Gamma_D[\vec{u}, \mathfrak{r}^\circ_\Gamma\ \vec{u}\ ({\mathfrak{p}^\bullet}_z\ \vec{u})\ x] \to (A[z := \mathbf{S}\, z])_D[\mathfrak{r}^\bullet\ \vec{u}\ ({\mathfrak{p}^\bullet}_z\ \vec{u}), x]$$

which is, up to a commutation, no more than the realization property for the term $\mathfrak{r}$, so that we can conclude.

As usual, observe how we needed the merge property to recover the right hypothesis coming from the other term. The interpretation of this rule features furthermore a strong parallelism between the source proof and the corresponding term, which both reason by induction on the considered integer.

□

7.6 A bit of classical logic

Gödel's Dialectica interprets strictly more than purely intuitionistic arithmetic, which was its interest in the first place. As explained in Section 7.4, it allows to realize Markov's principle and the independence of premise.

In the following, we will be looking at the actual realizers of those axioms in the Dialectica interpretation.

7.6.1 Irrelevant types

Before going further in the implementation of those two principles, we first have to formally define what we were assuming above for a type to be irrelevant. This will be crucial in the understanding of the computational content of our axioms.

The definition itself is actually fairly simple, because we have a built-in notion of *having no content* for a list of types: namely, to be empty.

Definition 88 (Irrelevance)**.** We say that a proposition A is irrelevant if

$$\mathbb{W}(A) = \emptyset \quad \text{and} \quad \mathbb{C}(A) = \emptyset$$

Quite a lot of propositions are irrelevant. In particular, the so-called negative fragment is irrelevant.

Proposition 44. *Assume A and B two irrelevant types, and t, u two terms.*

- *$\bot$ and $\top$ are irrelevant*
- *$t = u$ is irrelevant*
- *$A \to B$ is irrelevant*
- *$A \wedge B$ is irrelevant*

Proof. By unfolding of the type interpretations. □

This is not too much of a restriction if looking only from the logical point of view. Indeed, we can freely cast decidable propositions into irrelevant ones.

Proposition 45. *Let P be a decidable proposition, that is, there exists some term b such that*

$$\vdash (P \wedge b = 0) \vee (\neg P \wedge b \neq 0)$$

then there exists a decidable irrelevant proposition $[P]$ such that

$$\vdash (P \to [P]) \wedge ([P] \to P)$$

Proof. Simply take $[P] := b = 0$. □

7.6.2 Markov's principle

We are now trying to realize in the Dialectica translation Markov's principle, i.e. the formula $\neg\forall x.\, \neg A \to \exists x.\, A$, assuming that A is decidable and irrelevant. Thanks to Proposition 45 we can even safely assume without loss of generality that A is an equality.

The realizer displays a fancy blending of the assumption that its inner proposition is both decidable and irrelevant, while actively taking advantage of the reverse proofs $(-)^\circ$ coming from the translation. We will not insist on this particular feature in this chapter, but we will fully dedicate Chapter 8 to it.

For now we unfold the type translation of the negation in order to make typing clearer.

Proposition 46. *Let A be a* **HA***-formula. Then we have the following equalities.*

$$\begin{aligned}
\mathbb{W}(\neg A) &= \mathbb{W}(A) \to \mathbb{C}(A) \\
\mathbb{C}(\neg A) &= \mathbb{W}(A) \\
(\neg A)_D[\vec{\varphi}, \vec{u}] &= \neg A_D[\vec{u}, \vec{\varphi}\ \vec{u}]
\end{aligned}$$

Proof. Simple unfolding of definitions. □

We can finally write a proof term $\mathfrak{mp}$ realizing Markov's principle at any formula A. If we completely unfold the type of witnesses of the principle, we get

$$\begin{array}{rcl}
\mathbb{W}(\neg\forall x.\,\neg A \to \exists x.\,A) & = & \left\{\begin{array}{l}\mathbb{W}(\neg\forall x.\,\neg A) \to \mathbb{W}(\exists x.\,A)\ ;\\ \mathbb{W}(\neg\forall x.\,\neg A) \to \mathbb{C}(\exists x.\,A) \to \mathbb{C}(\neg\forall x.\,\neg A)\end{array}\right.\\
& = & \left\{\begin{array}{l}\mathbb{W}(\neg\forall x.\,\neg A) \to \mathbb{N}\ ;\\ \mathbb{W}(\neg\forall x.\,\neg A) \to \mathbb{W}(A)\ ;\\ \mathbb{W}(\neg\forall x.\,\neg A) \to \mathbb{C}(\exists x.\,A) \to \mathbb{C}(\neg\forall x.\,\neg A)\end{array}\right.\\
& = & \left\{\begin{array}{l}(\mathbb{W}(\forall x.\,\neg A) \to \mathbb{N}) \to (\mathbb{W}(\forall x.\,\neg A) \to \mathbb{W}(A)) \to \mathbb{N}\ ;\\ (\mathbb{W}(\forall x.\,\neg A) \to \mathbb{N}) \to (\mathbb{W}(\forall x.\,\neg A) \to \mathbb{W}(A)) \to \mathbb{W}(A)\ ;\\ \mathbb{W}(\neg\forall x.\,\neg A) \to \mathbb{C}(\exists x.\,A) \to \mathbb{W}(\forall x.\,\neg A)\end{array}\right.
\end{array}$$

so that we will be giving the following names to each component, in this order: $\mathfrak{mp}^w$, $\mathfrak{mp}^p$ and $\mathfrak{mp}^\circ$. The trick is that because A is irrelevant, we may perform further simplification of the expected types. Indeed, we have then

$$\begin{array}{rcl}\mathbb{W}(A) & = & \emptyset\\ \mathbb{W}(\forall x.\,\neg A) & = & \emptyset\end{array}$$

so that the three components actually collapse to

$$\mathbb{W}(\neg\forall x.\,\neg A \to \exists x.\,A) \quad = \quad \left\{\begin{array}{l}\mathbb{N} \to \mathbb{N}\ ;\\ \emptyset\ ;\\ \emptyset\end{array}\right.$$

It is now rather obvious how to define those three components: the second and third ones are totally specified by their (empty) sequence of types, and the first one begs to be identity for naturality reasons. Therefore, we pose

$$\begin{array}{rcl}\mathfrak{mp}^w & := & \lambda n : \mathbb{N}.\,n\\ \mathfrak{mp}^p & := & \emptyset\\ \mathfrak{mp}^\circ & := & \emptyset\end{array}$$

Observe how trivial are the terms thanks to the collapsing due to irrelevance.

The only remaining thing to do is to prove that these terms do realize Markov's principle. This is actually straightforward. Assume $n : \mathbb{W}(\forall x.\,\neg A) \to \mathbb{N}$, $f : \mathbb{W}(\forall x.\,\neg A) \to \mathbb{W}(A)$ and $k : \mathbb{N} \to \mathbb{C}(A)$, we must show that

$$(\neg\forall x.\,\neg A)_D[(n\ ;\ f), \mathfrak{mp}^\circ\ n\ f\ k] \to (\exists x.\,A)_D[(\mathfrak{mp}^w\ n\ f\ ;\ \mathfrak{mp}^p\ n\ f), k]$$

which is, by collapsing all the irrelevant terms, exactly the same as

$$(\neg\forall x.\,\neg A)_D[n, \emptyset] \to (\exists x.\,A)_D[n, \emptyset]$$

Further unfolding of this proposition results in

$$(\forall x.\,\neg A)_D[\emptyset, n] \to (\exists x.\,A)_D[n, \emptyset]$$

$$\neg(\neg A)_D[x := n] \to A_D[x := n]$$
$$\neg\neg A_D[x := n] \to A_D[x := n]$$

Luckily, the proposition A is decidable so that A_D is also decidable, and we can conclude by elimination of double-negation which is intuitionistically valid in this case.

As one can see, the computational content of the realizer is so simple that one could call it disappointing. Most of the magic occurs at the logical level of the realizability condition. Nonetheless, the very fact we have been able to write out the realizers relies on the presence of reverse proofs. Indeed, the source natural number for the function $\mathfrak{mp}^w$ comes from the reverse component of a proof of $\mathbb{W}(\neg\forall x.\, \neg A)$, so that the usual intuitionistic arrow alone would not have been sufficient to provide us with this witness.

7.6.3 Independence of premise

We now turn to the interpretation of the scheme $(A \to \exists x.\, B) \to \exists x.\, (A \to B)$ for any irrelevant formula A. As in the case of Markov's principle, the irrelevance hypothesis will result in the collapse of the translated formula.

Let us once again unfold the translation of witnesses of this principle.

$$\begin{aligned}
& \mathbb{W}((A \to \exists x.\, B) \to \exists x.\, (A \to B)) \\
= & \begin{cases} \mathbb{W}(A \to \exists x.\, B) \to \mathbb{W}(\exists x.\, (A \to B)) \,; \\ \mathbb{W}(A \to \exists x.\, B) \to \mathbb{C}(A \to \exists x.\, B) \to \mathbb{C}(\exists x.\, (A \to B)) \end{cases} \\
= & \begin{cases} \mathbb{W}(A \to \exists x.\, B) \to \mathbb{N} \,; \\ \mathbb{W}(A \to \exists x.\, B) \to \mathbb{W}(A \to B) \,; \\ \mathbb{W}(A \to \exists x.\, B) \to \mathbb{C}(A \to \exists x.\, B) \to \mathbb{C}(\exists x.\, (A \to B)) \end{cases}
\end{aligned}$$

It is helpful to notice that since A is irrelevant, then for any proposition C, $\mathbb{W}(A \to C) := \mathbb{W}(C)$ by an obvious unfolding, and for the same reasons $\mathbb{C}(A \to C) := \mathbb{C}(C)$. This allows to further simplify the witness type as follows.

$$\begin{aligned}
& \mathbb{W}((A \to \exists x.\, B) \to \exists x.\, (A \to B)) \\
= & \begin{cases} \mathbb{W}(\exists x.\, B) \to \mathbb{N} \,; \\ \mathbb{W}(\exists x.\, B) \to \mathbb{W}(B) \,; \\ \mathbb{W}(\exists x.\, B) \to \mathbb{C}(\exists x.\, B) \to \mathbb{C}(\exists x.\, (A \to B)) \end{cases} \\
= & \begin{cases} \mathbb{N} \to \mathbb{W}(B) \to \mathbb{N} \,; \\ \mathbb{N} \to \mathbb{W}(B) \to \mathbb{W}(B) \,; \\ \mathbb{N} \to \mathbb{W}(B) \to (\mathbb{N} \to \mathbb{C}(B)) \to \mathbb{N} \to \mathbb{C}(B) \end{cases}
\end{aligned}$$

Following the same argument as in the case of Markov's principle, we will call those three components $\mathfrak{ip}^w$, $\mathfrak{ip}^p$ and $\mathfrak{ip}^\circ$ respectively, and we will take the natural interpretation for them, that is,

$$\begin{aligned}
\mathfrak{ip}^w &:= \lambda(x : \mathbb{N})\,(p : \mathbb{W}(B)).\, x \\
\mathfrak{ip}^p &:= \lambda(x : \mathbb{N})\,(p : \mathbb{W}(B)).\, p \\
\mathfrak{ip}^\circ &:= \lambda(x : \mathbb{N})\,(p : \mathbb{W}(B))\,(k : \mathbb{N} \to \mathbb{C}(B)).\, k
\end{aligned}$$

It remains to prove that these realizers do realize the independence of premise. Once again, this is mostly a series of definition unfolding, even more than in the previous case. We have to show that, given any $x : \mathbb{N}$, $p : \mathbb{W}(B)$ and $k : \mathbb{N} \to \mathbb{C}(B)$, we have (as usual, using the same bound name as the argument in the quantifier to avoid writing explicitly the substitution) the following unfolding steps:

$$(A \to \exists x.\, B)_D[(x\ ;\ p), \mathfrak{ip}^\circ\ k] \to (\exists x.\, (A \to B))_D[(\mathfrak{ip}^w\ x\ p\ ;\ \mathfrak{ip}^p\ x\ p), k]$$

$$(A \to \exists x.\, B)_D[(x\ ;\ p), k] \to (\exists x.\, (A \to B))_D[(x\ ;\ p), k]$$

$$(A_D \to B_D[p, k\ x]) \to A_D \to B_D[p, k\ x]$$

the latter one being a trivial proposition.

Note that we relied on the fact that x was not free in A when unfolding the realization relation for the right-hand side existential in the second-to-last line, as it would have also been substituted in A.

Although the realizer of independence of premise is essentially as trivial as its counterpart for Markov's principle, there is an important difference that deserves to be highlighted. We remarked that in the case of Markov's principle, the realizer relied on the reverse translations. Yet, independence of premises does not require the reverse translation at all. It merely uses the fact that decidable arguments may be made irrelevant, and thus erased from the realizer, for they are lost in the translation. This is in no way as strong as the requirement of reverse translations.

8 A proof-theoretical Dialectica translation

> Utinam tam facile vera invenire possem, quam falsa convincere.
>
> *Cicero about proof theory.*

In order to move to a more modern setting, we are going to forget about arithmetic, and focus on the propositional content of the Dialectica transformation. In the course of this evolution, we will remove some of the encoding tricks of the historical presentation.

8.1 Down with System T

The first thing we are going to get rid of is the use of sequences, as well as the ad-hoc encoding of sum types. To this end, we will use a simply-typed λ-calculus with inductive datatypes $\lambda^{\times+}$ instead of System T.

It is not difficult to see that the derivation rules of the propositional fragment of **HA** are mostly the computational erasure of the typing rules of $\lambda^{\times+}$, following the Curry-Howard paradigm. It is indeed almost sufficient to simply forget about the λ-terms to recover the corresponding **HA**-proofs.

The presentation of some rules is slightly different, though. The main difference lies in the presentation of positive connectives, because while **HA** features *additive* n-ary conjunctions, $\lambda^{\times+}$ uses the *multiplicative* presentation. Compare for instance the elimination rule for $\lambda^{\times+}$ and its computational erasure:

$$\frac{\Gamma \vdash t : A \times B \qquad \Gamma, x : A, y : B \vdash u : C}{\Gamma \vdash \texttt{match}\ t\ \texttt{with}\ (x, y) \mapsto u : C} \qquad \frac{\Gamma \vdash A \times B \qquad \Gamma, A, B \vdash C}{\Gamma \vdash C}$$

versus the equivalent rules we took for **HA**:

$$\frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A} \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}$$

One can witness here that **HA** eliminates conjunctions by projections, while $\lambda^{\times+}$ does it through pattern-matching. Although the logical expressiveness is preserved, their operational behaviour is different. If we forget about this little mismatch, we can nonetheless encode $\lambda^{\times+}$-terms as **HA** derivations.

This has one obvious application: we could apply directly the historical Dialectica translation on $\lambda^{\times+}$ and get a term translation for free. Yet, the historical translation is

impractical. The use of sequences is cumbersome, and because we now have true inductive connectives, we can get rid of it. In addition, some nice proof-theoretical properties of the Dialectica translation are hidden beneath a pile of technicalities.

By presenting it directly as a proof translation, we will show that it is better behaved that one could have thought of.

8.2 A proof system over $\lambda^{\times+}$

Similarly to the arithmetical case, we need a system to state properties about the $\lambda^{\times+}$-terms we are going to manipulate. The system we are going to choose is a very simple first-order dependently-typed theory. Indeed, we hardly need something more powerful than the arithmetical case.

We will be taking special care to be able to reason about inductive types, though, under the form of dependent elimination rules upon their inhabitants. This will be realized thanks to the use of proposition-level matching constructs mimicking the ones from term levels. In order not to confuse the two levels of types and propositions, we will make this stratification explicit in the syntax by writing propositions $\mathbf{A}, \mathbf{B}$ in boldface while retaining the normal typeface A, B for $\lambda^{\times+}$ types.

Definition 89 (Formulae)**.** The formulae of our logic are inductively defined below.

$$\begin{aligned}\mathbf{A}, \mathbf{B} \quad := \quad & \top \mid \bot \mid \mathbf{A} \wedge \mathbf{B} \mid \mathbf{A} \vee \mathbf{B} \mid \mathbf{A} \to \mathbf{B} \\ & \mid \texttt{match}\ t\ \texttt{with}\ (x, y) \mapsto \mathbf{A} \mid \texttt{match}\ t\ \texttt{with}\ () \mapsto \mathbf{A} \\ & \mid \texttt{match}\ t\ \texttt{with}\ [\cdot] \mid \texttt{match}\ t\ \texttt{with}\ [x \mapsto \mathbf{A} \mid y \mapsto \mathbf{B}]\end{aligned}$$

Environments are list of hypotheses, as usual.

$$\mathbf{\Gamma} := \cdot \mid \mathbf{\Gamma}, \mathbf{A}$$

As in the $\mathbf{HA} + T$ case, our propositions may contain terms, so we need to ensure that they are well-formed, which amounts in this case to check that the dependent elimination rules are well-typed.

Definition 90 (Well-formedness)**.** We inductively define below the well-formedness $\Sigma \vdash_{\text{wf}} \mathbf{A}$ of a formula $\mathbf{A}$ in a $\lambda^{\times+}$-context Σ.

$$\frac{}{\Sigma \vdash_{\text{wf}} \top} \qquad \frac{}{\Sigma \vdash_{\text{wf}} \bot} \qquad \frac{\Sigma \vdash_{\text{wf}} \mathbf{A} \qquad \Sigma \vdash_{\text{wf}} \mathbf{B}}{\Sigma \vdash_{\text{wf}} \mathbf{A} \wedge \mathbf{B}} \qquad \frac{\Sigma \vdash_{\text{wf}} \mathbf{A} \qquad \Sigma \vdash_{\text{wf}} \mathbf{B}}{\Sigma \vdash_{\text{wf}} \mathbf{A} \vee \mathbf{B}}$$

$$\frac{\Sigma \vdash_{\text{wf}} \mathbf{A} \qquad \Sigma \vdash_{\text{wf}} \mathbf{B}}{\Sigma \vdash_{\text{wf}} \mathbf{A} \to \mathbf{B}}$$

$$\frac{\Sigma \vdash t : 1 \qquad \Sigma \vdash_{\text{wf}} \mathbf{C}}{\Sigma \vdash_{\text{wf}} \texttt{match}\ t\ \texttt{with}\ () \mapsto \mathbf{C}} \qquad \frac{\Sigma \vdash t : A \times B \qquad \Sigma, x : A, y : B \vdash_{\text{wf}} \mathbf{C}}{\Sigma \vdash_{\text{wf}} \texttt{match}\ t\ \texttt{with}\ (x, y) \mapsto \mathbf{C}}$$

$$\frac{\Sigma \vdash t : 0}{\Sigma \vdash_{\text{wf}} \texttt{match}\ t\ \texttt{with}\ [\cdot]} \qquad \frac{\Sigma \vdash t : A + B \qquad \Sigma, x : A \vdash_{\text{wf}} \mathbf{C}_1 \qquad \Sigma, y : B \vdash_{\text{wf}} \mathbf{C}_2}{\Sigma \vdash_{\text{wf}} \texttt{match}\ t\ \texttt{with}\ [x \mapsto \mathbf{C}_1 \mid y \mapsto \mathbf{C}_2]}$$

Well-formedness is extended to lists of hypotheses in a direct way.

$$\frac{}{\Sigma \vdash_{\text{wf}} \cdot} \qquad \frac{\Sigma \vdash_{\text{wf}} \mathbf{\Gamma} \qquad \Sigma \vdash_{\text{wf}} \mathbf{A}}{\Sigma \vdash_{\text{wf}} \mathbf{\Gamma}, \mathbf{A}}$$

There is nonetheless a subtle difference between the current system and $\mathbf{HA} + T$. Indeed, in $\mathbf{HA}{+}T$, computation was restricted to terms, and we used plain substitution to define convertibility of formulae. Now, because of the presence of dependent elimination, formulae themselves may be converted into one another.

Definition 91 (Convertibility)**.** Two propositions $\mathbf{A}$ and $\mathbf{B}$ are convertible, written $\mathbf{A} \equiv_\beta \mathbf{B}$, if they are in relation by the context closure of the following generators.

$$\frac{t \equiv_\beta u \qquad x \text{ fresh}}{\mathbf{A}[x := t] \equiv_\beta \mathbf{A}[x := u]} \qquad \frac{}{\texttt{match}\ ()\ \texttt{with}\ () \mapsto \mathbf{B} \equiv_\beta \mathbf{B}}$$

$$\frac{}{\texttt{match}\ (t, u)\ \texttt{with}\ (x, y) \mapsto \mathbf{B} \equiv_\beta \mathbf{B}[x := t, y := u]}$$

$$\frac{}{\texttt{match}\ \mathbf{inl}\ t\ \texttt{with}\ [x \mapsto \mathbf{B}_1 \mid y \mapsto \mathbf{B}_2] \equiv_\beta \mathbf{B}_1[x := t]}$$

$$\frac{}{\texttt{match}\ \mathbf{inr}\ u\ \texttt{with}\ [x \mapsto \mathbf{B}_1 \mid y \mapsto \mathbf{B}_2] \equiv_\beta \mathbf{B}_2[y := u]}$$

We finish the definition of our metatheory by actually giving the derivation rules for the whole system.

Definition 92 (Rules)**.** Sequents are of the form $\Sigma \mid \mathbf{\Gamma} \vdash \mathbf{A}$. The derivation rules of the system are given below.

$$\frac{\Sigma \vdash_{\mathrm{wf}} \mathbf{\Gamma}, \mathbf{A}, \mathbf{\Delta}}{\Sigma \mid \mathbf{\Gamma}, \mathbf{A}, \mathbf{\Delta} \vdash \mathbf{A}} \qquad \frac{\Sigma \mid \mathbf{\Gamma} \vdash \mathbf{A} \to \mathbf{B} \qquad \Sigma \mid \mathbf{\Gamma} \vdash \mathbf{A}}{\Sigma \mid \mathbf{\Gamma} \vdash \mathbf{B}} \qquad \frac{\Sigma \mid \mathbf{\Gamma}, \mathbf{A} \vdash \mathbf{B}}{\Sigma \mid \mathbf{\Gamma} \vdash \mathbf{A} \to \mathbf{B}}$$

$$\frac{\mathbf{A} \equiv_\beta \mathbf{B} \qquad \Sigma \mid \mathbf{\Gamma} \vdash \mathbf{B} \qquad \Sigma \vdash_{\mathrm{wf}} \mathbf{A}}{\Sigma \mid \mathbf{\Gamma} \vdash \mathbf{A}} \qquad \frac{\Sigma \vdash_{\mathrm{wf}} \mathbf{\Gamma}}{\Sigma \mid \mathbf{\Gamma} \vdash \top} \qquad \frac{\Sigma \mid \mathbf{\Gamma} \vdash \bot}{\Sigma \mid \mathbf{\Gamma} \vdash \mathbf{A}}$$

$$\frac{\Sigma \mid \mathbf{\Gamma} \vdash \mathbf{A} \qquad \Sigma \mid \mathbf{\Gamma} \vdash \mathbf{B}}{\Sigma \mid \mathbf{\Gamma} \vdash \mathbf{A} \wedge \mathbf{B}} \qquad \frac{\Sigma \mid \mathbf{\Gamma} \vdash \mathbf{A} \wedge \mathbf{B}}{\Sigma \mid \mathbf{\Gamma} \vdash \mathbf{A}} \qquad \frac{\Sigma \mid \mathbf{\Gamma} \vdash \mathbf{A} \wedge \mathbf{B}}{\Sigma \mid \mathbf{\Gamma} \vdash \mathbf{B}}$$

$$\frac{\Sigma \mid \mathbf{\Gamma} \vdash \mathbf{A}}{\Sigma \mid \mathbf{\Gamma} \vdash \mathbf{A} \vee \mathbf{B}} \qquad \frac{\Sigma \mid \mathbf{\Gamma} \vdash \mathbf{B}}{\Sigma \mid \mathbf{\Gamma} \vdash \mathbf{A} \vee \mathbf{B}}$$

$$\frac{\Sigma \mid \mathbf{\Gamma} \vdash \mathbf{A} \vee \mathbf{B} \qquad \Sigma \mid \mathbf{\Gamma}, \mathbf{A} \vdash \mathbf{C} \qquad \Sigma \mid \mathbf{\Gamma}, \mathbf{B} \vdash \mathbf{C}}{\Sigma \mid \mathbf{\Gamma} \vdash \mathbf{C}}$$

$$\frac{\Sigma \vdash t : 1 \qquad \Sigma \mid \mathbf{\Gamma} \vdash \mathbf{C}[z := ()] \qquad z \text{ fresh}}{\Sigma \mid \mathbf{\Gamma} \vdash \mathbf{C}[z := t]}$$

$$\frac{\Sigma \vdash t : A \times B \qquad \Sigma, x : A, y : B \mid \mathbf{\Gamma} \vdash \mathbf{C}[z := (x, y)] \qquad x, y, z \text{ fresh}}{\Sigma \mid \mathbf{\Gamma} \vdash \mathbf{C}[z := t]}$$

$$\frac{\Sigma \vdash t : 0 \qquad \Sigma \vdash_{\mathrm{wf}} \mathbf{\Gamma}, \mathbf{C}[z := t] \qquad z \text{ fresh}}{\Sigma \mid \mathbf{\Gamma} \vdash \mathbf{C}[z := t]}$$

$$\frac{\Sigma \vdash t : A + B \qquad \begin{cases} \Sigma \mid \mathbf{\Gamma}, x : A \vdash \mathbf{C}[z := \mathbf{inl}\ x] \\ \Sigma \mid \mathbf{\Gamma}, y : B \vdash \mathbf{C}[z := \mathbf{inr}\ y] \end{cases} \qquad x, y, z \text{ fresh}}{\Sigma \mid \mathbf{\Gamma} \vdash \mathbf{C}[z := t]}$$

8.3 Dialectica with inductive types

The goal of this section is to adapt the historical presentation to our calculus with inductive datatypes. This is actually rather straightforward. The Dialectica translation on types will be mostly following the Curry-Howard interpretation, that is, we will translate logical connectives into their corresponding computational equivalent.

There is actually nothing involved in here. We will parallel the structure used in the historical definition again, as we actually need the very same ingredients that will be defined in the same order:

- Two translations $\mathbb{W}(-)$ and $\mathbb{C}(-)$ on types;
- A notion of dummy terms $✠_-$ (here we actually need two such terms, one for each type translation) in the target language;

- For each source type A a predicate A_D on proofs and counters in the proof system;
- A decide primitive in the target language;
- Two families of terms translations to the target language.

The reader may find that there is much repetition from the historical definition, which is not untrue. Nevertheless, there are some subtle differences to point out. When things are too similar, we will skip the details.

The one difference lies in the fact that we do not use sequences anymore. Thanks to the presence of true positive types in $\lambda^{\times+}$, instead of resorting to the metatheoretical artifact of sequences, we can encode them directly in the terms. Typically, sequences are going to be turned into products, and we will get rid of the bizarre sum type encoding through the use of a proper sum type.

Let us focus on the type translations to clear things up.

8.3.1 Witnesses and counters

Definition 93 (Type translation)**.** The translation on types is inductively defined in the table below. The $\mathbb{W}(-)$ and $\mathbb{C}(-)$ translations associate to a type of $\lambda^{\times+}$ another type of $\lambda^{\times+}$.

	$\mathbb{W}(-)$	$\mathbb{C}(-)$
$A \to B$	$\begin{cases} \mathbb{W}(A) \to \mathbb{W}(B) \\ \times \\ \mathbb{W}(A) \to \mathbb{C}(B) \to \mathbb{C}(A) \end{cases}$	$\mathbb{W}(A) \times \mathbb{C}(B)$
1	1	1
$A \times B$	$\mathbb{W}(A) \times \mathbb{W}(B)$	$\mathbb{C}(A) + \mathbb{C}(B)$
0	1	1
$A + B$	$\mathbb{W}(A) + \mathbb{W}(B)$	$\mathbb{C}(A) \times \mathbb{C}(B)$

The reader may be surprised by the translation of the empty type 0, which is translated as 1. This is actually a necessity of the translation inherited from the historical Dialectica: we need to be able to construct a dummy proof-term at any type. The unsound proof-terms which do not realize a given formula will be ruled out by the realizability relation defined by the interpretation matrix.

8.3.2 Orthogonality

In our jump to a proof-as-program paradigm, we will take advantage of the occasion to rename the interpretation matrix, and adorn it with a more realizability-friendly name, viz. *orthogonality*. There is nothing complicated here: this is a direct adaptation of the historical case to inductive types.

Definition 94 (Orthogonality)**.** Given two fresh variables u and x, we inductively define the orthogonality relation $A_D[u, x]$ on A below.

- $0_D[u, x] := \bot$
- $1_D[u, x] := \top$
- $(A + B)_D[w, z] := \texttt{match}\ w\ \texttt{with}\ [u \mapsto A_D[u, \mathrm{fst}\ z] \mid v \mapsto B_D[v, \mathrm{snd}\ z]]$
- $(A \times B)_D[w, z] := \texttt{match}\ z\ \texttt{with}\ [x \mapsto A_D[\mathrm{fst}\ w, x] \mid y \mapsto B_D[\mathrm{snd}\ w, y]]$
- $(A \to B)_D[w, z] := A_D[\mathrm{fst}\ z, \mathrm{snd}\ w\ (\mathrm{fst}\ z)\ (\mathrm{snd}\ z)] \to B_D[\mathrm{fst}\ w\ (\mathrm{fst}\ z), \mathrm{snd}\ z]$

Proposition 47. *For any $\lambda^{\times +}$ type A,*

$$u : \mathbb{W}(A), x : \mathbb{C}(A) \vdash_{\mathrm{wf}} A_D[u, x]$$

Proof. By induction on A. □

8.3.3 Interpretation

We still have to rely on dummy terms to write the interpretation. Luckily, we chose the type interpretation such that all target types are inhabited. We construct this default inhabitant in a way similar to the one of the historical presentation.

Proposition 48. *For all type A, there exist two closed terms $\maltese_A : \mathbb{C}(A)$ and $\overline{\maltese}_A : \mathbb{W}(A)$.*

Proof. By induction on A. For $\maltese_\_$:

- $\maltese_0 := ()$
- $\maltese_1 := ()$
- $\maltese_{A \times B} := \mathbf{inl}\ \maltese_A$
- $\maltese_{A + B} := (\maltese_A, \maltese_B)$
- $\maltese_{A \to B} := (\overline{\maltese}_A, \maltese_B)$

and for $\overline{\maltese}_\_$:

- $\overline{\maltese}_0 := ()$
- $\overline{\maltese}_1 := ()$
- $\overline{\maltese}_{A \times B} := (\overline{\maltese}_A, \overline{\maltese}_B)$
- $\overline{\maltese}_{A + B} := \mathbf{inl}\ \overline{\maltese}_A$
- $\overline{\maltese}_{A \to B} := (\lambda\_.\, \overline{\maltese}_B, \lambda\_\ \_.\, \maltese_A)$

□

Remark 12. The non-canonicity of such a term is obvious in the context of $\lambda^{\times+}$-terms. Indeed, the dummy term $\maltese_{A+B}$ (resp. $\maltese_{A\times B}$) is built by arbitrarily choosing one side of the sum type to fill. This will be the source of a serious issue later on.

Definition 95 (Booleans)**.** The $\lambda^{\times+}$ language allows us to write true booleans, by setting $\mathbb{B} := 1+1$. We can write the usual operations on it without any problem. We define some useful constants below

$$\begin{aligned} \text{true} &:= \mathbf{inl}\ () \\ \text{false} &:= \mathbf{inr}\ () \\ \text{if} &:= \lambda b\, t\, f.\, \texttt{match}\ b\ \texttt{with}\ [x \mapsto t \mid y \mapsto f] \end{aligned}$$

as well as the following macro

$$\mathbf{True}[b] \quad := \quad \texttt{match}\ b\ \texttt{with}\ [x \mapsto \top \mid y \mapsto \bot]$$

Likewise, as in the arithmetical case, we can still decide the orthogonality relation.

Proposition 49. *For all type A, there exists a $\lambda^{\times+}$-term $\vdash \text{decide}_A : \mathbb{W}(A) \to \mathbb{C}(A) \to \mathbb{B}$ with the following property:*

$$u : \mathbb{W}(A), x : \mathbb{C}(A) \mid \cdot \vdash \mathbf{True}[\text{decide}_A\ u\ x] \leftrightarrow A_D[u, x]$$

Proof. By induction on A.

- $\text{decide}_0 := \lambda u\, x.\, \text{false}$
- $\text{decide}_1 := \lambda u\, x.\, \text{true}$
- $\text{decide}_{A+B} := \lambda w\, z.\, \texttt{match}\ w\ \texttt{with}\ [u \mapsto \text{decide}_A\ u\ (\text{fst}\ z) \mid v \mapsto \text{decide}_B\ v\ (\text{snd}\ z)]$
- $\text{decide}_{A\times B} := \lambda w\, z.\, \texttt{match}\ z\ \texttt{with}\ [x \mapsto \text{decide}_A\ (\text{fst}\ w)\ x \mid y \mapsto \text{decide}_B\ (\text{snd}\ w)\ y]$
- $\text{decide}_{A\to B} := \lambda w\, z.\, \text{if}\ \ (\text{decide}_A\ (\text{fst}\ z)\ (\text{snd}\ z\ (\text{fst}\ z)\ (\text{snd}\ z)))\ (\text{decide}_B\ (\text{fst}\ w\ (\text{fst}\ z))\ (\text{snd}\ z))\ \text{true}$

Proof of the specification follows immediately by a parallel induction on A. □

Proposition 50. *Using the* decide *term, we can build for all $\lambda^{\times+}$-type A a term*

$$\vdash \text{merge}_A : \mathbb{C}(A) \to \mathbb{C}(A) \to \mathbb{W}(A) \to \mathbb{C}(A)$$

with the following property:

$$u : \mathbb{W}(A), x_1 : \mathbb{C}(A), x_2 : \mathbb{C}(A) \mid \cdot \vdash A_D[u, \text{merge}_A\ x_1\ x_2\ u] \leftrightarrow A_D[u, x_1] \wedge A_D[u, x_2]$$

Proof. As in the historical case, we pose

$$\mathrm{merge}_A := \lambda x_1\, x_2\, u.\, \mathrm{if}\ (\mathrm{decide}_A\ u\ x_1)\ x_2\ x_1$$

The proof uses exactly the same arguments as in the historical case, that is, a case analysis on the boolean leading on one case to the expected result, and in the other case to an absurdity.

□

Similarly to the historical case, were we needed merge over sequences, we need to lift the merging operation to environments, i.e. list of hypotheses. For all environment Γ, we need a term

$$\mathrm{merge}^i_\Gamma : \mathbb{C}(\Gamma_1) \to \ldots \to \mathbb{C}(\Gamma_n) \to \mathbb{C}(\Gamma_1) \to \ldots \to \mathbb{C}(\Gamma_n) \to \mathbb{W}(\Gamma_1) \to \ldots \to \mathbb{W}(\Gamma_n) \to \mathbb{C}(\Gamma_i)$$

with the property that

$$\vec{u} : \mathbb{W}(\Gamma), \vec{x} : \mathbb{C}(\Gamma), \vec{y} : \mathbb{C}(\Gamma) \mid \cdot \vdash \left(\bigwedge\nolimits_i \Gamma_{iD}[u_i, \mathrm{merge}^i_\Gamma\ \vec{x}\ \vec{y}\ \vec{u}]\right) \leftrightarrow \left(\bigwedge\nolimits_i \Gamma_{iD}[u_i, x_i] \wedge \Gamma_{iD}[u_i, y_i]\right)$$

We do not give the explicit construction here, but it is easily constructed from each $\mathrm{merge}_{\Gamma_j}$ by induction over Γ. We will abuse the notation by writing merge^x_Γ for merge^i_Γ where i is the index of x in Γ.

Now we can write out the translation acting on $\lambda^{\times+}$-terms. Similarly to the case of **HA** proofs, where a proof $\mathfrak{p}$ was mapped to a tuple of sequences $(\mathfrak{p}^\bullet, \mathfrak{p}_1, \ldots, \mathfrak{p}_n)$, the translation of a term is going to be given by a tuple of $\lambda^{\times+}$-terms.

This translation is typed, meaning the we still need the underlying typing derivation. This is not as bad as it seems, because our system is simply-typed so inference is decidable, hence we can somehow recover it from the term itself, but this does not make it as readable as it should. The requirement is to be able to recover the type of a variable in an environment, as defined below.

Definition 96. Let Γ be an environment, and x a variable. We define the type $\Gamma(x)$ by induction on Γ.

- $(\Gamma, x : A)(x) := A$
- $(\Gamma, y : B)(x) := \Gamma(x)$
- $(\cdot)(x) := 0$

The choice of 0 for the value of a variable in the empty environment is arbitrary, well-typed terms will not be using it anyway.

Definition 97 (Term translation)**.** Let t be a term, x a variable, Γ a list of hypotheses and A a type. We define mutually recursively the terms $(\Gamma \vdash t : A)^\bullet$ and $(\Gamma \vdash t : A)_x$ below by induction on the derivation $\Gamma \vdash t : A$.

First, the $(- \vdash - : -)^\bullet$ translation:

$$(\Gamma \vdash x : A)^\bullet := x$$

$$(\Gamma \vdash \lambda x.\, t : A \to B)^\bullet := (\lambda x.\, (\Gamma, x : A \vdash t : B)^\bullet, \lambda x\, \pi.\, (\Gamma, x : A \vdash t : B)_x\ \pi)$$

$$(\Gamma \vdash t\ u : B)^\bullet := \mathrm{fst}\ (\Gamma \vdash t : A \to B)^\bullet\ (\Gamma \vdash u : A)^\bullet$$

$$(\Gamma \vdash \mathtt{match}\ t\ \mathtt{with}\ () \mapsto u : C)^\bullet := \mathtt{match}\ (\Gamma \vdash t : 1)^\bullet\ \mathtt{with}\ () \mapsto (\Gamma \vdash u : C)^\bullet$$

$$(\Gamma \vdash \mathtt{match}\ t\ \mathtt{with}\ [\cdot] : C)^\bullet := \overline{\blacksquare}_C$$

$$(\Gamma \vdash \mathtt{match}\ t\ \mathtt{with}\ (x, y) \mapsto u : C)^\bullet := \mathtt{match}\ (\Gamma \vdash t : A \times B)^\bullet\ \mathtt{with}\ (x, y) \mapsto (\Gamma, x : A, y : B \vdash u : C)^\bullet$$

$$(\Gamma \vdash \mathtt{match}\ t\ \mathtt{with}\ [x \mapsto u_1 \mid y \mapsto u_2] : C)^\bullet := \mathtt{match}\ (\Gamma \vdash t : A + B)^\bullet\ \mathtt{with}\ [x \mapsto (\Gamma, x : A \vdash u_1 : C)^\bullet \mid y \mapsto (\Gamma, y : C \vdash u_2 : C)^\bullet]$$

$$(\Gamma \vdash () : 1)^\bullet := ()$$

$$(\Gamma \vdash (t, u) : A \times B)^\bullet := ((\Gamma \vdash t : A)^\bullet, (\Gamma \vdash u : B)^\bullet)$$

$$(\Gamma \vdash \mathbf{inl}\ t : A + B)^\bullet := \mathbf{inl}\ (\Gamma \vdash t : A)^\bullet$$

$$(\Gamma \vdash \mathbf{inr}\ t : A + B)^\bullet := \mathbf{inr}\ (\Gamma \vdash t : B)^\bullet$$

and the $(- \vdash - : -)_x$ translation:

$$(\Gamma \vdash x : A)_x := \lambda\pi.\, \pi$$

$$(\Gamma \vdash y : A)_x := \lambda\pi.\, \maltese_{\Gamma(x)}$$

$$(\Gamma \vdash \lambda y.\, t : A \to B)_x := \lambda\rho.\, \mathtt{match}\ \rho\ \mathtt{with}\ (y, \pi) \mapsto (\Gamma, y : A \vdash t : B)_x\ \pi$$

$$(\Gamma \vdash t\ u : B)_x := \lambda\pi.\, \mathrm{merge}^x_\Gamma \begin{array}{l} ((\Gamma \vdash t : A \to B)_{\vec{x}}\ ((\Gamma \vdash u : A)^\bullet, \pi)) \\ ((\Gamma \vdash u : A)_{\vec{x}}\ (\mathrm{snd}\ (\Gamma \vdash t : A \to B)^\bullet\ (\Gamma \vdash u : A)^\bullet\ \pi)) \\ \vec{x} \end{array}$$

$$(\Gamma \vdash \mathtt{match}\ t\ \mathtt{with}\ () \mapsto u : C)_x := \lambda\pi.\, \mathrm{merge}^x_\Gamma \begin{array}{l} ((\Gamma \vdash t : 1)_{\vec{x}}\ ()) \\ ((\Gamma \vdash u : C)_{\vec{x}}\ \pi) \\ \vec{x} \end{array}$$

$$(\Gamma \vdash \mathtt{match}\ t\ \mathtt{with}\ [\cdot] : C)_x := \lambda\pi.\, (\Gamma \vdash t : 0)_x\ \pi$$

$$(\Gamma \vdash \mathtt{match}\ t\ \mathtt{with}\ (y, z) \mapsto u : C)_x := \lambda\pi.\, \mathrm{merge}^x_\Gamma\ (\mathtt{match}\ (\Gamma \vdash t : A \times B)^\bullet\ \mathtt{with}\ (y, z) \mapsto (\Gamma, y : A, z : B \vdash u : C)_{\vec{x}}\ \pi) \left(\begin{array}{l} \mathrm{merge}^{\vec{x}}_\Gamma\ \ (\Gamma \vdash t : A \times B)_{\vec{x}}\ (\mathrm{inl}\ ((\Gamma, y : A, z : B \vdash u : C)_y\ \pi)) \\ \qquad\qquad\ (\Gamma \vdash t : A \times B)_{\vec{x}}\ (\mathrm{inr}\ ((\Gamma, y : A, z : B \vdash u : C)_z\ \pi)) \\ \qquad\qquad\ \vec{x} \end{array} \right)_{\vec{x}}$$

$$\begin{array}{l} (\Gamma \vdash \mathtt{match}\ t\ \mathtt{with}\ [y \mapsto u_1 \mid z \mapsto u_2] : C)_x := \\ \lambda\pi.\, \mathrm{merge}^x_\Gamma\ \ (\mathtt{match}\ (\Gamma \vdash t : A + B)^\bullet\ \mathtt{with}\ [y \mapsto (\Gamma, y : A \vdash u_1 : C)_{\vec{x}}\ \pi \mid z \mapsto (\Gamma, z : B \vdash u_2 : C)_{\vec{x}}\ \pi]) \\ \qquad\qquad ((\Gamma \vdash t : A + B)_{\vec{x}}\ (((\Gamma, y : A \vdash u_1 : C)_{\vec{x}}\ \pi), ((\Gamma, z : B \vdash u_2 : C)_{\vec{x}}\ \pi))) \\ \qquad\qquad \vec{x} \end{array}$$

$$(\Gamma \vdash () : 1)_x := \lambda\pi.\, \maltese_{\Gamma(x)}$$

$$(\Gamma \vdash (t, u) : A \times B)_x := \lambda\pi.\, \mathtt{match}\ \pi\ \mathtt{with}\ [\pi \mapsto (\Gamma \vdash t : A)_x\ \pi \mid \pi \mapsto (\Gamma \vdash u : B)_x\ \pi]$$

$$(\Gamma \vdash \mathbf{inl}\ t : A + B)_x := \lambda\pi.\, (\Gamma \vdash t : A)_x\ (\mathrm{fst}\ \pi)$$

$$(\Gamma \vdash \mathbf{inr}\ t : A + B)_x := \lambda\pi.\, (\Gamma \vdash t : B)_x\ (\mathrm{snd}\ \pi)$$

As one may witness, the term translation is really close to the historical transformation, except that we are using true inductive types instead of sequences. There is one exception though, which is found in the elimination rules of conjunctions. While in the historical presentation, we described them as projections, leading to a quite direct interpretation, here, the interpretation of the pattern-matching over products is fairly more involved, requiring two merge operations. This is due, as we will see later, to a mismatch of interpretation: the type interpretation of products given here is typically negative, thus giving naturally rise to a projection-oriented system.

Theorem 19 (Type soundness)**.** *For all typed $\lambda^{\times+}$-term $\Gamma \vdash t : A$, the translation*

preserves typing, i.e.

$$\begin{array}{lcll} \mathbb{W}(\Gamma) & \vdash & (\Gamma \vdash t : A)^\bullet & : \quad \mathbb{W}(A) \\ \mathbb{W}(\Gamma) & \vdash & (\Gamma \vdash t : A)_x & : \quad \mathbb{C}(A) \to \mathbb{C}(\Gamma_i) \quad \textit{when } (x : \Gamma_i) \in \Gamma \end{array}$$

Proof. By induction on the typing derivation. □

Theorem 20 (Realization)**.** *For all closed $\lambda^{\times+}$-term $\vdash t : A$, the translated term $(\cdot \vdash t : A)^\bullet$ realizes A, that is:*

$$\pi : \mathbb{C}(A) \mid \cdot \vdash A_D[(\cdot \vdash t : A)^\bullet, \pi]$$

The theorem actually follows from the following generalization.

Proposition 51. *For all $\lambda^{\times+}$-term $\vec{x} : \Gamma \vdash t : A$, the translated term $(\vec{x} : \Gamma \vdash t : A)^\bullet$ realizes A, that is:*

$$\vec{x} : \mathbb{W}(\Gamma), \pi : \mathbb{C}(A) \mid \cdot \vdash A_D[(\Gamma \vdash t : A)^\bullet, \pi]$$

Proof. By induction on the typing derivation. The arguments are essentially the same as the one given in the historical case, except that we use inductives instead of sequences. There is nothing novel compared to the historical presentation, so we skip the details. □

8.4 Linear Dialectica

We are going to reformulate a result from the end of the 80's due to De Paiva [92], namely that Gödel's Dialectica can be expressed as a translation acting on linear logic rather than intuitionistic arithmetic.

8.4.1 The linear decomposition

We now turn to consider linear logic types. For the sake of conciseness, we will focus on positive types while defining negative types by duality. The grammar of types we consider is then:

$$A, B := 1 \mid 0 \mid A \otimes B \mid A \oplus B \mid !A \mid A^\perp$$

As in the case of $\lambda^{\times+}$-types, we define a pair of translations $\mathbb{W}(-)$ and $\mathbb{C}(-)$ on linear logic types.

Definition 98 (Linear interpretation)**.** For any linear type A, we inductively define on A two $\lambda^{\times+}$-types, the witness type $\mathbb{W}(A)$ and counter type $\mathbb{C}(A)$ as follows.

	$\mathbb{W}(-)$	$\mathbb{C}(-)$
1	1	1
$A \otimes B$	$\mathbb{W}(A) \times \mathbb{W}(B)$	$\begin{cases} \mathbb{W}(A) \to \mathbb{C}(B) \\ \times \\ \mathbb{W}(B) \to \mathbb{C}(A) \end{cases}$
0	1	1
$A \oplus B$	$\mathbb{W}(A) + \mathbb{W}(B)$	$\mathbb{C}(A) \times \mathbb{C}(B)$
$!A$	$\mathbb{W}(A)$	$\mathbb{W}(A) \to \mathbb{C}(A)$
$A^\perp$	$\mathbb{C}(A)$	$\mathbb{W}(A)$

The orthogonality relation can be easily adapted to this linear interpretation. For the sake of completeness, we recall it here, although it is rather straightforwardly obtained by letting oneself guide by the typing hints.

Definition 99 (Linear orthogonality)**.** Given two fresh variables u and x, we inductively define the orthogonality relation $A_D[u, x]$ on a linear type A below.

- $0_D[u, x] := \bot$
- $1_D[u, x] := \top$
- $(A \oplus B)_D[w, z] := \texttt{match}\ w\ \texttt{with}\ [u \mapsto A_D[u, \mathrm{fst}\ z] \mid v \mapsto B_D[v, \mathrm{snd}\ z]]$
- $(A \otimes B)_D[w, z] := A_D[\mathrm{fst}\ w, \mathrm{snd}\ z\ (\mathrm{fst}\ w)] \wedge B_D[\mathrm{snd}\ w, \mathrm{fst}\ z\ (\mathrm{snd}\ w)]$
- $(!A)_D[w, z] := A_D[w, z\ w]$
- $(A^\perp)_D[x, u] := \neg A_D[u, x]$

Proposition 52. *For any linear type A,*

$$u : \mathbb{W}(A), x : \mathbb{C}(A) \vdash_{\mathrm{wf}} A_D[u, x]$$

As expected, correct proofs of linear logic are mapped to $\lambda^{\times+}$-terms satisfying the usual orthogonality property.

Theorem 21 (Linear soundness)**.** *From every proof in linear logic of a formula A, one can construct a $\lambda^{\times+}$-term p of type $\mathbb{W}(A)$ such that*

$$\pi : \mathbb{C}(A) \mid \cdot \vdash A_D[p, \pi]$$

We will not discuss or even prove this result here because it is not our current point of attention. For more details see for instance De Paiva's thesis [92].

8.4.2 Factorizing

As described by De Paiva in her thesis [92], the previous construction can be recovered by carefully choosing a decomposition of intuitionistic types into linear types. It is indeed an exotic mix of call-by-name and call-by-value translations.

- Arrows are interpreted through the call-by-name decomposition
- Sums are interpreted through a call-by-value decomposition
- Products are interpreted through a negative call-by-value decomposition

This is formally expressed below.

Definition 100 (Exotic calling convention)**.** We inductively define the translation $[\![\cdot]\!]_e$ from $\lambda^{\times+}$ types to linear logic types as follows.

- $[\![0]\!]_e := 0$
- $[\![1]\!]_e := 1$
- $[\![A \times B]\!]_e := [\![A]\!]_e \,\&\, [\![B]\!]_e$
- $[\![A + B]\!]_e := [\![A]\!]_e \oplus [\![B]\!]_e$
- $[\![A \to B]\!]_e := ![\![A]\!]_e \multimap [\![B]\!]_e$

We recall that in the above definition, we used the usual encodings by duality:

$$\begin{array}{lcl} A \,\&\, B & := & (A^\perp \oplus B^\perp)^\perp \\ A \multimap B & := & (A \otimes B^\perp)^\perp \end{array}$$

This translation is also known as the *economical* translation according to Di Cosmo [34].

The exotic translation is more natural to handle when considering sequents, because it prevents requiring a lot of exponential operations, hence its nickname. Nonetheless, it is quite alien from the point of view of programming, because it mixes calling conventions and polarities. The fact that pairs are negative, and thus defined by projections, contrasts with the semantics we would like to give them in call-by-name. This explains in particular the oddity of the translation of pattern-matching over pairs in the term interpretation.

As we were hoping for, the intuitionistic type translation factorizes through the linear decomposition.

Proposition 53. *Let A be a $\lambda^{\times+}$-type, then the $\lambda^{\times+}$-types $\mathbb{W}(A)$ and $\mathbb{W}([\![A]\!]_e)$ (resp. $\mathbb{C}(A)$ and $\mathbb{C}([\![A]\!]_e)$) are isomorphic.*

Please remark that we abused the $\mathbb{W}(-)$ and $\mathbb{C}(-)$ notations: in the first case, this is a translation acting on $\lambda^{\times+}$-types, while in the second case, this translation is defined on linear logic types.

Proof. By induction on A. The only interesting case is the translation of the arrow. We have indeed:

$$\mathbb{W}(A \to B) \quad := \quad (\mathbb{W}(A) \to \mathbb{W}(B)) \times (\mathbb{W}(A) \to \mathbb{C}(B) \to \mathbb{C}(A))$$

and on the other hand:

$$\begin{aligned}
\mathbb{W}([\![A \to B]\!]_e) \quad &:= \quad \mathbb{W}(!{[\![A]\!]_e} \multimap [\![B]\!]_e) \\
&\equiv \quad \mathbb{C}(!{[\![A]\!]_e} \otimes {[\![B]\!]_e}^{\perp}) \\
&\equiv \quad (\mathbb{W}(!{[\![A]\!]_e}) \to \mathbb{C}({[\![B]\!]_e}^{\perp})) \times (\mathbb{W}({[\![B]\!]_e}^{\perp}) \to \mathbb{C}(!{[\![A]\!]_e})) \\
&\equiv \quad (\mathbb{W}([\![A]\!]_e) \to \mathbb{W}([\![B]\!]_e)) \times (\mathbb{C}([\![B]\!]_e) \to \mathbb{W}([\![A]\!]_e) \to \mathbb{C}([\![A]\!]_e)) \\
&\cong \quad (\mathbb{W}([\![A]\!]_e) \to \mathbb{W}([\![B]\!]_e)) \times (\mathbb{W}([\![A]\!]_e) \to \mathbb{C}([\![B]\!]_e) \to \mathbb{C}([\![A]\!]_e))
\end{aligned}$$

We conclude by the induction hypothesis applied to A and B. The counter case is treated directly.

□

The strange mix of calling conventions in the current state of our translation may seem unsettling in the light of modern proof theory. Actually, it is not really an issue yet, because we did not focus on the dynamic behaviour of our translated terms. It is indeed possible to give a realizability account for the Dialectica translation with this decomposition. Yet, the description we will be doing of it in the next chapter will stick to the more uniform linear decompositions, in order not to complicate too much the structure of the machines involved.

8.5 A not-so proof-theoretical translation

One could think that the presentation we gave of a revised Dialectica translation above is enough to label it as a proof-theoretical transformation. For sure, we took any typed $\lambda^{\times+}$-term and translated it into another $\lambda^{\times+}$-term, while preserving typing. This sounds Curry-Howardesque enough... or does it?

The following fact about the current translation shoud wash our hope away.

Proposition 54. *There exists two $\lambda^{\times+}$-terms $\Gamma \vdash t : A$ and $\Gamma \vdash u : A$ such that $t \equiv_\beta u$ but $(\Gamma \vdash t : A)^\bullet \not\equiv_\beta (\Gamma \vdash u : A)^\bullet$.*

This incompatibility is not merely a technical issue related to some missing administrative reduction step. We can find indeed two terms whose translations are not equatable for any sensible equivalence over $\lambda^{\times+}$-terms.

Example 4. Take for instance $\vdash \lambda y.\, (\lambda x.\, x)\ y : A \to A$ and $\vdash \lambda y.\, y : A \to A$. Obviously the two terms are convertible, but we have

$$\begin{array}{lll}
(\vdash \lambda y.\, y : A \to A)^\bullet & \equiv_\beta & (\lambda y.\, y, \lambda y\, \pi.\, \pi) \\
(\vdash \lambda y.\, (\lambda x.\, x)\ y : A \to A)^\bullet & \equiv_\beta & (\lambda y.\, y, \lambda y\, \pi.\, \mathrm{merge}_A\ \maltese_A\ \pi\ y)
\end{array}$$

Now, for a careful choice of A we can show that $\mathrm{merge}_A \ \maltese_A \ \pi \ y$ has no reason to be convertible to π. Say for example that $A := 0 \to A_0$ for some type A_0. Then because the orthogonality test on witnesses of 0 is trivial, we have

$$\mathrm{merge}_A \ \maltese_A \ \pi \ y \equiv_\beta ((), \maltese_{A_0})$$

which is in general distinct of π. Note that this incompatibility is actually already present in the historical translation, as long as we think of **LJ** and System T as two variants of the simply-typed λ-calculus.

We are facing quite a bothersome issue there. We claimed that our translation was an instance of a proof-as-program interpretation of logic, but it turns out it does not even preserve the semantics of the source proof seen as a program. This is only half of a failure, though. Even though the translation does not agree with the semantics, we have nice type decompositions respecting the dualities of linear logic.

It should be obvious to a pair of trained eyes that the source of the above failure is the requirement of the merge operations and their evil counterparts, the dummy terms, for they are totally lacking any form of naturality in the categorical acceptance of the term. The next chapter shows that it is actually possible to get around it, in a clean and algebraic fashion.

9 A realizability account

> La logique mène à tout, à condition d'en sortir.
>
> *Alphonse Allais about realizability.*

This chapter is dedicated to the elaboration of a presentation of the Dialectica translation that does work well, based on De Paiva's linear decomposition presented before. By carefully choosing one decomposition or another, we will be able to describe precisely the computational content hidden beneath the translation and unravel the true nature of Dialectica as a translation adding a certain class of side-effects in our programming language. This line of work is heavily influenced by Krivine and Miquel's presentation of forcing in Krivine's realizability [81].

9.1 Introducing multisets

In this section, we are first going to get rid of the last encoding artifact of the previous presentations, ultimately inherited from the historical Dialectica.

9.1.1 Motivations

The reader should have remarked by now that all the previous presentations required the same tricks to allow to write the realizers.

- on the one hand, we need to be able to write out dummy terms, to fill holes when there is no way to construct a term of the given type from the context
- on the other, we need to be able to decide orthogonality, in order to dynamically choose from a witness or another

Recall that the dummy term is not canonical in general; in the case of witnesses for proofs of the sum + one could actually choose either side and feed it with the corresponding dummy term of that side. This is one of the issue we encounter when carelessly trying to prove that our translation should preserve β-equivalence of λ-terms. In particular, we do not have the desirable equality

$$\mathrm{merge}_A \ \maltese_A \ \pi \ t \equiv_\beta \pi$$

even when π and t are well-typed.

Worse, the very need to be able to define dummy terms forces use to have all interpreted types inhabited, and in particular requires that we interpret the empty type 0 by the singleton type 1, furthermore requiring a term placeholder in the elimination rule of 0. Recall that in the translation of the $\lambda^{\times+}$-calculus, we posed indeed

$$(\Gamma \vdash \texttt{match}\ t\ \texttt{with}\ [\cdot] : C)^\bullet := \overline{\maltese}_C$$

because $t^\bullet$ had type 1, so we could not use the usual elimination rule of falsity on it to produce a proof of C.

All these observations hint at the fact that the issue stems from the use of dummy terms and merge. If we look carefully at the soundness proof, be it from the historical translation or in the $\lambda^{\times+}$-case, we do not require much from those structures. The following suffices:

- at the level of terms, $\maltese_A$ must exists;
- at the level of realizability, merge must commute with the realization relation, as in Proposition 41.

More importantly, in the translation of pure intuitionistic sequents or $\lambda^{\times+}$-terms, if we forget about the elimination of 0, those two terms are only used when defining the reverse translations $(-)^\circ$ and $(-)_x$. In particular, they only appear on counter types coming from a bang type in the linear decomposition. This is no surprise, because as outlined in the historical case, dummy terms are used when there is some form of weakening occurring, while merge is used for duplication. And those use cases are precisely the task of exponentials in linear logic.

9.1.2 Formal definition

Therefore, we are going to algebraize this behaviour by assuming an abstract datatype featuring the same properties as $\maltese$ and merge, but compatible with β-equivalence. This will be the rôle of *abstract multisets*.

Definition 101 (Abstract multisets)**.** An abstract multiset datastructure is given by:

- a parameterized type $\mathfrak{M}\,(-)$;
- two operations giving it a monad signature:

$$\begin{array}{rcl} \{\cdot\} & : & A \to \mathfrak{M}\,A \\ \gg\!= & : & \mathfrak{M}\,A \to (A \to \mathfrak{M}\,B) \to \mathfrak{M}\,B \end{array}$$

- two operations giving it a monoid signature:

$$\begin{array}{rcl} \varnothing & : & \mathfrak{M}\,A \\ \odot & : & \mathfrak{M}\,A \to \mathfrak{M}\,A \to \mathfrak{M}\,A \end{array}$$

We do not ask for particular equality of terms now. We will make them explicit when we need it later on. For now, one may truthfully think of it as respecting monad laws and equipped with a monoidal structure.

Note that we will be using infix notations in λ-terms in the remaining of this chapter, to ease the reading of terms. As usual, $\gg\!=$ is left associative and binds less tightly than application but more than pattern matching. The $\odot$ operation is left associative and binds more tightly than anything except application.

As explained before, we change the type interpretation of the bang connective in the translation of linear logic type to make it use the abstract multiset structure.

Definition 102 (Linear translation revisited)**.** Given a linear logic type A, we will define two $\lambda^{\times+}$-types $\mathbb{W}(A)$ and $\mathbb{C}(A)$ by induction on A as given below.

	$\mathbb{W}(-)$	$\mathbb{C}(-)$
1	1	1
$A \otimes B$	$\mathbb{W}(A) \times \mathbb{W}(B)$	$\begin{cases} \mathbb{W}(A) \to \mathbb{C}(B) \\ \mathbb{W}(B) \to \mathbb{C}(A) \end{cases}$
0	0	1
$A \oplus B$	$\mathbb{W}(A) + \mathbb{W}(B)$	$\mathbb{C}(A) \times \mathbb{C}(B)$
$!A$	$\mathbb{W}(A)$	$\mathbb{W}(A) \to \mathfrak{M}\,\mathbb{C}(A)$
$A^\perp$	$\mathbb{C}(A)$	$\mathbb{W}(A)$

Remark that compared to the definition from Section 8.4, we only changed the interpretation of $\mathbb{W}(0)$ and $\mathbb{C}(!A)$. The first change is motivated by the removal of now useless dummy terms, while the second has been discussed more thoroughly above.

9.1.3 A taste of déjà-vu

Remark 13. The use of abstract multisets corresponds to the notion of Hyland's well-behaved monoids [58]. While the former uses it in the context of double-glueing, we use it here in the Dialectica translation, which was one of the roots for the design of the double-glueing construction.

Actually, the use of such a structure is not novel. It is a generalization of the Diller-Nahm construction [40].

Proposition 55. *If we take $\mathfrak{M} := \mathfrak{P}_f$ to be the finite set structure with the expected operations, and compose with the $[\![\cdot]\!]_e$ translation, we recover the so-called Diller-Nahm variant of the Dialectica translation.*

There is quite a methodologic difference though. The Diller-Nahm translation is motivated by the fact that the original translation requires atomic types to be inhabited,

which may be much too strong from the logical point of view. That is not our base motivation, as we aim to recover the preservation of β-equivalence, which is a much more proof-theoretical demeanour. Moreover, the target system of the Diller-Nahm translation is often thought of as a variant of some set theory, which is essentially non-computable. We, in turn, take it to be a true programming language equipped with types. This is another difference.

9.1.4 The whereabouts of orthogonality

The reader should have noticed by now that we did not mention the orthogonality relation defined on the multiset-using types. This is perfectly doable, as long as we have a notion of being true at each elements for our abstract multisets. In that case, we could define

$$(!A)_D[w, z] := \forall x \in z\ w.A_D[w, x]$$

where the $\in$ notation stands for the property of pointwise truth. This is actually doable, basing ourselves on the case of the Diller-Nahm translation which is a particular instance of this datastructure. Nonetheless, there are some particular points that will refrain ourselves from pursueing into this direction.

- First, as far as preservation of reduction is concerned, we actually do not need orthogonality anymore. A careful design of translation will be sufficient to ensure that, if we bothered about orthogonality, our translated terms would actually verify the realizability relation. Another way to state it is that we now only care of the computational contents of the skeleton of our proofs, not on the internalized logical properties they convey.
- Second, orthogonality can chiefly be seen as a way to rule out unsoundness of the resulting model when allowing proof terms inhabiting any type. Once we get rid of the dummy terms we were using all along, soundness will be preserved by construction, i.e. by choosing to interpret the empty type as itself.

For these reasons, we will now drop the orthogonality conditions of our translated terms and only focus on their respective definitions and types.

9.2 The call-by-name translation

In this section, we present the Dialectica translation obtained by composing its linear version with the call-by-name translation from intuitionistic logic into linear logic. We will be letting ourselves guide by the type translation to elaborate the term translation.

We first focus on the pure λ-calculus as a source language, for its simplicity, although we will extend it to positive connectives in a subsequent section. Therefore, we will restrict to the following type grammar.

$$A, B := \alpha \mid A \to B$$

As we will see, the translation requires pairs, so our target language will be the subset of the $\lambda^{\times+}$-calculus deprived of sum types but enriched with an abstract multiset datatype.

9.2.1 Type translation

Definition 103 (Call-by-name decomposition)**.** We recall that the call-by-name linear decomposition $[\![\cdot]\!]_{\mathrm{n}}$ described at Section 3.4.1 is inductively defined over our types as

$$\begin{array}{lcl} [\![\alpha]\!]_{\mathrm{n}} & := & \alpha \\ [\![A \to B]\!]_{\mathrm{n}} & := & ![\![A]\!]_{\mathrm{n}} \multimap [\![B]\!]_{\mathrm{n}} \end{array}$$

while sequents are interpreted as

$$[\![\Gamma_1, \ldots, \Gamma_n \vdash A]\!]_{\mathrm{n}} := ![\![\Gamma_1]\!]_{\mathrm{n}} \multimap \ldots \multimap ![\![\Gamma_n]\!]_{\mathrm{n}} \multimap [\![A]\!]_{\mathrm{n}}$$

Proposition 56. *The resulting type translations on the arrow type are:*

$$\begin{array}{lcl} \mathbb{W}([\![A \to B]\!]_{\mathrm{n}}) & := & (\mathbb{W}([\![A]\!]_{\mathrm{n}}) \to \mathbb{W}([\![B]\!]_{\mathrm{n}})) \times (\mathbb{C}([\![B]\!]_{\mathrm{n}}) \to \mathbb{W}([\![A]\!]_{\mathrm{n}}) \to \mathfrak{M}\,\mathbb{C}([\![A]\!]_{\mathrm{n}})) \\ \mathbb{C}([\![A \to B]\!]_{\mathrm{n}}) & := & \mathbb{W}([\![A]\!]_{\mathrm{n}}) \times \mathbb{C}([\![B]\!]_{\mathrm{n}}) \end{array}$$

Note that these types are, up to isomorphism and to the introduction of multisets, the same as those obtained in the exotic decomposition.

If we were to use the linear type interpretations $\mathbb{W}(-)$ and $\mathbb{C}(-)$ on sequent translations, we would obtain a convoluted translated type. Instead of doing so, we choose to see sequent translation through an isomorphism, resulting in a presentation close to the previous sections.

Proposition 57. *Let $\Gamma_1, \ldots, \Gamma_n$ and A be intuitionistic types, then we have the following isomorphism.*

$$\mathbb{W}([\![\Gamma_1, \ldots, \Gamma_n \vdash A]\!]_{\mathrm{n}}) \cong \left\{ \begin{array}{c} \mathbb{W}([\![\Gamma_1]\!]_{\mathrm{n}}) \to \ldots \to \mathbb{W}([\![\Gamma_n]\!]_{\mathrm{n}}) \to \mathbb{W}([\![A]\!]_{\mathrm{n}}) \\ \times \\ \mathbb{W}([\![\Gamma_1]\!]_{\mathrm{n}}) \to \ldots \to \mathbb{W}([\![\Gamma_n]\!]_{\mathrm{n}}) \to \mathbb{C}([\![A]\!]_{\mathrm{n}}) \to \mathfrak{M}\,\mathbb{C}([\![\Gamma_1]\!]_{\mathrm{n}}) \\ \times \\ \ldots \\ \times \\ \mathbb{W}([\![\Gamma_1]\!]_{\mathrm{n}}) \to \ldots \to \mathbb{W}([\![\Gamma_n]\!]_{\mathrm{n}}) \to \mathbb{C}([\![A]\!]_{\mathrm{n}}) \to \mathfrak{M}\,\mathbb{C}([\![\Gamma_n]\!]_{\mathrm{n}}) \end{array} \right.$$

Proof. By induction on n. □

We can clearly see that the translation of a sequent produces two types of objects, as in the previously presented translations.

- the first component of type

  $$\mathbb{W}([\![\Gamma_1]\!]_{\mathrm{n}}) \to \ldots \to \mathbb{W}([\![\Gamma_n]\!]_{\mathrm{n}}) \to \mathbb{W}([\![A]\!]_{\mathrm{n}})$$

  is what corresponds to the direct translations $(-)^\bullet$ of the historical and revised cases.

- for each free variable $(x_i : \Gamma_i)$, a term of type

$$\mathbb{W}(\llbracket \Gamma_1 \rrbracket_{\mathrm{n}}) \to \ldots \to \mathbb{W}(\llbracket \Gamma_n \rrbracket_{\mathrm{n}}) \to \mathbb{C}(\llbracket A \rrbracket_{\mathrm{n}}) \to \mathfrak{M}\,\mathbb{C}(\llbracket \Gamma_i \rrbracket_{\mathrm{n}})$$

  which corresponds to the i-th projection of the historical reverse translation $(-)^\circ$ and to the revised translation $(-)_{x_i}$ itself.

Thanks to the isomorphism, we can make the obvious observation that those two type translations share a common prefix, which is precisely the $\mathbb{W}(\llbracket - \rrbracket_{\mathrm{n}})$ translation applied to the environment pointwise.

This observation greatly simplifies the translation: we can handle free variables directly, by simply making the design choice that the translation preserves them (up to a type lifting).

9.2.2 Term translation

As in the previous variant of the translation, we keep the respective names $(-)^\bullet$ and $(-)_x$ for some variable x for the two translations, which we formally define below.

Definition 104 (Term translation)**.** Given a λ-term t and a variable x, we mutually define the translations $t^\bullet$ and t_x by induction on t below.

$$\begin{array}{lcl}
x^\bullet & := & x \\
(\lambda x.\, t)^\bullet & := & (\lambda x.\, t^\bullet, \lambda \pi\, x.\, t_x\ \pi) \\
(t\ u)^\bullet & := & \mathrm{fst}\ t^\bullet\ u^\bullet \\
x_x & := & \lambda \pi.\, \{\pi\} \\
x_y & := & \lambda \pi.\, \emptyset \\
(\lambda y.\, t)_x & := & \lambda (y, \pi).\, t_x\ \pi \\
(t\ u)_x & := & \lambda \pi.\, (\mathrm{snd}\ t^\bullet\ \pi\ u^\bullet \gg\!= \lambda \rho.\, u_x\ \rho) \odot (t_x\ (u^\bullet, \pi))
\end{array}$$

As explained above, this translation copreserves free variables.

Proposition 58. *For all term t and variables x, y, if x is free in $t^\bullet$ or in t_y, then it is free in t.*

Proof. By induction on t. The interesting detail one should look at is the fact that translations of the λ-abstraction also close the bound variable. □

Another interesting remark regarding free variables has to be made in the translation of λ-abstractions. Assume a term $\lambda x.\, t$ for instance. Then x is (potentially) free in t, but not in $\lambda x.\, t$. So there is an additional reverse translation t_x available to t, but not to $\lambda x.\, t$. In order not to lose this information, $(\lambda x.\, t)^\bullet$ packs it into a pair. Hence we can see the reverse translation of arrows as a way to keep information coming from the variable being bound.

9.2.3 Typing soundness

As with the previous translations, and as expected, this translation preserves typing in the following sense.

Theorem 22 (Typing soundness)**.** *If*

$$x_1 : \Gamma_1, \ldots, x_n : \Gamma_n \vdash t : A$$

then

$$x_1 : \mathbb{W}(\llbracket \Gamma_1 \rrbracket_{\mathrm{n}}), \ldots, x_n : \mathbb{W}(\llbracket \Gamma_n \rrbracket_{\mathrm{n}}) \vdash t^\bullet : \mathbb{W}(\llbracket A \rrbracket_{\mathrm{n}})$$

and

$$x_1 : \mathbb{W}(\llbracket \Gamma_1 \rrbracket_{\mathrm{n}}), \ldots, x_n : \mathbb{W}(\llbracket \Gamma_n \rrbracket_{\mathrm{n}}) \vdash t_{x_i} : \mathbb{C}(\llbracket A \rrbracket_{\mathrm{n}}) \to \mathfrak{M}\,\mathbb{C}(\llbracket \Gamma_i \rrbracket_{\mathrm{n}})$$

for all $1 \leq i \leq n$.

Proof. By induction on the typing derivation.

- Suppose $x_1 : \Gamma_1, \ldots, x_n : \Gamma_n \vdash x_i : \Gamma_i$. We must provide the following three typing derivations, where $i \neq j$.

  $$x_1 : \mathbb{W}(\llbracket \Gamma_1 \rrbracket_{\mathrm{n}}), \ldots, x_n : \mathbb{W}(\llbracket \Gamma_n \rrbracket_{\mathrm{n}}) \vdash x_i : \mathbb{W}(\llbracket \Gamma_i \rrbracket_{\mathrm{n}})$$
  $$x_1 : \mathbb{W}(\llbracket \Gamma_1 \rrbracket_{\mathrm{n}}), \ldots, x_n : \mathbb{W}(\llbracket \Gamma_n \rrbracket_{\mathrm{n}}) \vdash \lambda\pi.\,\{\pi\} : \mathbb{C}(\llbracket \Gamma_i \rrbracket_{\mathrm{n}}) \to \mathfrak{M}\,\mathbb{C}(\llbracket \Gamma_i \rrbracket_{\mathrm{n}})$$
  $$x_1 : \mathbb{W}(\llbracket \Gamma_1 \rrbracket_{\mathrm{n}}), \ldots, x_n : \mathbb{W}(\llbracket \Gamma_n \rrbracket_{\mathrm{n}}) \vdash \lambda\pi.\,\emptyset : \mathbb{C}(\llbracket \Gamma_i \rrbracket_{\mathrm{n}}) \to \mathfrak{M}\,\mathbb{C}(\llbracket \Gamma_j \rrbracket_{\mathrm{n}})$$

  These derivations are obvious.

- Suppose $\Gamma \vdash \lambda x.\, t : A \to B$. We must check that:

  $$\mathbb{W}(\llbracket \Gamma \rrbracket_{\mathrm{n}}) \vdash \lambda x.\, t^\bullet : \mathbb{W}(\llbracket A \rrbracket_{\mathrm{n}}) \to \mathbb{W}(\llbracket B \rrbracket_{\mathrm{n}})$$
  $$\mathbb{W}(\llbracket \Gamma \rrbracket_{\mathrm{n}}) \vdash \lambda\pi\, x.\, t_x\ \pi : \mathbb{C}(\llbracket B \rrbracket_{\mathrm{n}}) \to \mathbb{W}(\llbracket A \rrbracket_{\mathrm{n}}) \to \mathfrak{M}\,\mathbb{C}(\llbracket A \rrbracket_{\mathrm{n}})$$
  $$\mathbb{W}(\llbracket \Gamma \rrbracket_{\mathrm{n}}) \vdash \lambda(x, \pi).\, t_y\ \pi : \mathbb{W}(\llbracket A \rrbracket_{\mathrm{n}}) \times \mathbb{C}(\llbracket B \rrbracket_{\mathrm{n}}) \to \mathfrak{M}\,\mathbb{C}(\llbracket C \rrbracket_{\mathrm{n}})$$

  when $(y : C) \in \Gamma$. Then the first typing derivation comes from the induction hypothesis on $t^\bullet$, the second one from the hypothesis on t_x and the third one from the hypothesis on t_y.

- Suppose $\Gamma \vdash t\ u : B$, where $\Gamma \vdash t : A \to B$. We must check that:

  $$\mathbb{W}(\llbracket \Gamma \rrbracket_{\mathrm{n}}) \vdash \mathrm{fst}\ t^\bullet\ u^\bullet : \mathbb{W}(\llbracket B \rrbracket_{\mathrm{n}})$$
  $$\mathbb{W}(\llbracket \Gamma \rrbracket_{\mathrm{n}}) \vdash \lambda\pi.\, (\mathrm{snd}\ t^\bullet\ \pi\ u^\bullet \gg\!= \lambda\rho.\, u_x\ \rho) \odot (t_x\ (u^\bullet, \pi)) : \mathbb{C}(\llbracket B \rrbracket_{\mathrm{n}}) \to \mathfrak{M}\,\mathbb{C}(\llbracket C \rrbracket_{\mathrm{n}})$$

when $(x : C) \in \Gamma$. The first derivation is obtained by applying the induction hypothesis to $t^\bullet$ and $u^\bullet$. The second one can be decomposed in the two following derivations.

$$\mathbb{W}(\llbracket\Gamma\rrbracket_{\mathrm{n}}), \pi : \mathbb{C}(\llbracket B\rrbracket_{\mathrm{n}}) \vdash \mathrm{snd}\ t^\bullet\ \pi\ u^\bullet \gg\!= \lambda\rho.\, u_x\ \rho : \mathfrak{M}\,\mathbb{C}(\llbracket C\rrbracket_{\mathrm{n}})$$
$$\mathbb{W}(\llbracket\Gamma\rrbracket_{\mathrm{n}}), \pi : \mathbb{C}(\llbracket B\rrbracket_{\mathrm{n}}) \vdash t_x\ (u^\bullet, \pi) : \mathfrak{M}\,\mathbb{C}(\llbracket C\rrbracket_{\mathrm{n}})$$

But by induction hypothesis, we get

$$\mathbb{W}(\llbracket\Gamma\rrbracket_{\mathrm{n}}), \pi : \mathbb{C}(\llbracket B\rrbracket_{\mathrm{n}}) \vdash \mathrm{snd}\ t^\bullet\ \pi\ u^\bullet : \mathfrak{M}\,\mathbb{C}(\llbracket A\rrbracket_{\mathrm{n}})$$
$$\mathbb{W}(\llbracket\Gamma\rrbracket_{\mathrm{n}}) \vdash u_x : \mathbb{C}(\llbracket A\rrbracket_{\mathrm{n}}) \to \mathfrak{M}\,\mathbb{C}(\llbracket C\rrbracket_{\mathrm{n}})$$
$$\mathbb{W}(\llbracket\Gamma\rrbracket_{\mathrm{n}}) \vdash t_x : \mathbb{C}(\llbracket A \to B\rrbracket_{\mathrm{n}}) \to \mathfrak{M}\,\mathbb{C}(\llbracket C\rrbracket_{\mathrm{n}})$$
$$\mathbb{W}(\llbracket\Gamma\rrbracket_{\mathrm{n}}), \pi : \mathbb{C}(\llbracket B\rrbracket_{\mathrm{n}}) \vdash (u^\bullet, \pi) : \mathbb{C}(\llbracket A \to B\rrbracket_{\mathrm{n}})$$

so we conclude by reassembling those typing derivations.

□

9.2.4 Computational soundness

We will now prove properties of preservation of term reduction through the translation. Recall that this is the default that led us to the design of the multiset-using version of Dialectica. Therefore, this theorem is probably the most important about the revised translation!

In order for us to be able to do so, we first need to explain the expected computational behaviour of abstract multisets.

Definition 105 (Multiset reductions)**.** We expect the following β-equivalences on abstract multisets.

- Monadic laws:

$$\begin{array}{lcl} \{t\} \gg\!= f & \equiv_\beta & f\ t \\ t \gg\!= \lambda x.\, \{x\} & \equiv_\beta & t \\ t \gg\!= f \gg\!= g & \equiv_\beta & t \gg\!= \lambda x.\, g\ (f\ x) \end{array}$$

- Monoidal laws:

$$\begin{array}{lcl} \emptyset \odot t & \equiv_\beta & t \\ t \odot u & \equiv_\beta & u \odot t \\ t \odot (u \odot v) & \equiv_\beta & (t \odot u) \odot v \end{array}$$

- Distributivity laws:

$$\begin{array}{lll} \emptyset \gg\!= f & \equiv_\beta & \emptyset \\ t \odot u \gg\!= f & \equiv_\beta & (t \gg\!= f) \odot (u \gg\!= f) \\ t \gg\!= \lambda x.\, \emptyset & \equiv_\beta & \emptyset \\ t \gg\!= \lambda x.\, (f\ x) \odot (g\ x) & \equiv_\beta & (t \gg\!= f) \odot (u \gg\!= g) \end{array}$$

- Commutative cuts:

$$\begin{array}{lll} \texttt{match}\ t\ \texttt{with}\ (x, y) \mapsto \emptyset & \equiv_\beta & \emptyset \\ \texttt{match}\ t\ \texttt{with}\ (x, y) \mapsto u_1 \odot u_2 & \equiv_\beta & \\ \qquad (\texttt{match}\ t\ \texttt{with}\ (x, y) \mapsto u_1) \odot (\texttt{match}\ t\ \texttt{with}\ (x, y) \mapsto u_2) & & \\ \texttt{match}\ t\ \texttt{with}\ (x, y) \mapsto \{u\} & \equiv_\beta & \{\texttt{match}\ t\ \texttt{with}\ (x, y) \mapsto u\} \\ \texttt{match}\ t\ \texttt{with}\ (x, y) \mapsto (u \gg\!= f) & \equiv_\beta & \texttt{match}\ t\ \texttt{with}\ (x, y) \mapsto u \gg\!= f \end{array}$$

The rules that manipulate bound variables are only valid with the usual convention that they cannot create free variables.

We should discuss those rewriting rules a bit. The first three sets are what would be expected for a structure exhibiting the same algrebraical structure as a monad equipped with an internal commutative monoidal structure. The commutative cuts are a technical impediment appearing frequently when mixing positive types such as pairs together with other structure. Such artifacts tend to hint at the fact that elimination rules of positive connectives are badly behaved in natural deduction; and indeed they work more neatly in sequent calculus, see for instance Munch [84].

Concerning their implementability in our λ-calculus, one should not worry too much. Indeed, we do not really care about this in the translation. These reduction rules should be thought of as applying to an abstract datatype, where all those equivalences are handled directly in the machine interpreting our calculus.

Notation 9. Because the union is associative, we will drop the parentheses around multiple union applications for legibility.

Thanks to the rewriting rules we now have at hand, we can state and prove the reduction properties of the translation.

Proposition 59 (Emptiness)**.** *Let t be a λ-term and x a variable which is not free in t. Then the following holds.*

$$t_x \equiv_\beta \lambda\pi.\, \emptyset$$

Proof. By induction on t.

- Case y. Necessarily $x \neq y$, hence $y_x := \lambda\pi.\, \emptyset$.
- Case $\lambda y.\, t$. We have
$$(\lambda y.\, t)_x := \lambda(y, \pi).\, t_x\ \pi$$
so that by induction hypothesis, we recover
$$(\lambda y.\, t)_x \equiv_\beta \lambda(y, \pi).\, \emptyset$$

but recall that the right-hand side is just a notation, which, once unfolded results in the following equivalence

$$(\lambda y.\, t)_x \equiv_\beta \lambda\rho.\, \mathtt{match}\ \rho\ \mathtt{with}\ (y, \pi) \mapsto \emptyset$$

where ρ is fresh. We conclude by applying the commutative cut rule.

- Case $t\ u$. We have

$$(t\ u)_x := \lambda\pi.\, (\mathrm{snd}\ t^\bullet\ \pi\ u^\bullet \gg\!= \lambda\rho.\, u_x\ \rho) \odot (t_x\ (u^\bullet, \pi))$$

which results in the following when we rewrite the induction hypotheses:

$$(t\ u)_x \equiv_\beta \lambda\pi.\, (\mathrm{snd}\ t^\bullet\ \pi\ u^\bullet \gg\!= \lambda\rho.\, \emptyset) \odot \emptyset$$

We conclude by applying distributivity and monoidal rewriting rules.

□

The following is a purely technical lemma, which does not provide any insight in the translation.

Proposition 60 (Expansion lemma)**.** *For all term t and variable x, $t_x \equiv_\beta \lambda\pi.\, t_x\ \pi$ for some fresh variable π.*

Proof. Trivial case analysis on the term. All of our reverse translations start with a λ-abstraction, so that there is no need to apply some form of η-rule (which we do not have anyway). □

We can now state the generic substitution lemma. Note that we have to generalize it a little to make it pass it through. The second part of the equivalence may seem cryptic at first view, but its meaning will be deciphered when studying the computational content of the translation.

Proposition 61 (Substitution lemma)**.** *Let t and r be terms and x, y variables. Then the following holds.*

$$(t[x := r])^\bullet \equiv_\beta t^\bullet[x := r^\bullet]$$

and

$$(t[x := r])_y \equiv_\beta \lambda\pi.\, (t_y[x := r^\bullet]\ \pi) \odot (t_x[x := r^\bullet]\ \pi \gg\!= \lambda\rho.\, r_y\ \rho)$$

when $x \neq y$ and x is not free in r.

Proof. By simultaneous induction on the λ-term t.

- Case x. We have for the first equivalence:

$$(x[x := r])^\bullet \equiv_\beta r^\bullet \equiv_\beta x[x := r^\bullet] \equiv_\beta x^\bullet[x := r^\bullet]$$

while the second one gives for the left-hand side:

$$(x[x := r])_y := r_y$$

when the right-hand side is:

$$\begin{array}{ll} & \lambda\pi.\,(x_y[x := r^\bullet]\ \pi) \odot (x_x[x := r^\bullet]\ \pi \gg\!= \lambda\rho.\,r_y\ \rho) \\ \equiv_\beta & \lambda\pi.\,\emptyset \odot (\{\pi\} \gg\!= \lambda\rho.\,r_y\ \rho) \\ \equiv_\beta & \lambda\pi.\,r_y\ \pi \end{array}$$

We conclude by applying the expansion lemma.

- Case y. For the first equivalence we have

$$(y[x := r])^\bullet \equiv_\beta y \equiv_\beta y[x := r^\bullet] \equiv_\beta y^\bullet[x := r^\bullet]$$

For the left-hand side of the second we get:

$$(y[x := r])_y \equiv_\beta y_y \equiv_\beta \lambda\pi.\,\{\pi\}$$

while for the right-hand side we have:

$$\begin{array}{ll} & \lambda\pi.\,(y_y[x := r^\bullet]\ \pi) \odot (y_x[x := r^\bullet]\ \pi \gg\!= \lambda\rho.\,r_y\ \rho) \\ \equiv_\beta & \lambda\pi.\,\{\pi\} \odot (\emptyset \gg\!= \lambda\rho.\,r_y\ \rho) \\ \equiv_\beta & \lambda\pi.\,\{\pi\} \end{array}$$

- Case $z \neq x, y$. The first equivalence is the same as the previous case. We focus on the second one instead. The left-hand side gives:

$$(z[x := r])_y \equiv_\beta z_y \equiv_\beta \lambda\pi.\,\emptyset$$

and the right-hand side gives:

$$\begin{array}{ll} & \lambda\pi.\,(z_y[x := r^\bullet]\ \pi) \odot (z_x[x := r^\bullet]\ \pi \gg\!= \lambda\rho.\,r_y\ \rho) \\ \equiv_\beta & \lambda\pi.\,\emptyset \odot (\emptyset \gg\!= \lambda\rho.\,r_y\ \rho) \\ \equiv_\beta & \lambda\pi.\,\emptyset \end{array}$$

from which we conclude.

- Case $\lambda z.\,t$. We have for the first equation:

$$\begin{array}{ll} & ((\lambda z.\,t)[x := r])^\bullet \\ \equiv_\beta & (\lambda z.\,t[x := r])^\bullet \\ \equiv_\beta & (\lambda z.\,(t[x := r])^\bullet, \lambda\pi\,z.\,(t[x := r])_z\ \pi) \end{array}$$

while the right-hand side is:

$$\begin{array}{ll} & (\lambda z.\,t)^\bullet[x := r^\bullet] \\ \equiv_\beta & (\lambda z.\,t^\bullet[x := r^\bullet], \lambda\pi\,z.\,t_z[x := r^\bullet]\ \pi) \end{array}$$

The left projections of these pairs are provably β-equivalent thanks to the induction hypothesis. It remains to be proved that the right projections are also β-equivalent. Thanks to the reverse induction hypothesis applied to t with the variable z, we get:

$$\begin{array}{ll} & \lambda\pi\, z.\, (t[x := r])_z\ \pi \\ \equiv_\beta & \lambda\pi\, z.\, (t_z[x := r^\bullet]\ \pi) \odot (t_x[x := r^\bullet]\ \pi \gg\!= \lambda\rho.\, r_z\ \rho) \end{array}$$

Yet, z is not free in r because it was a bound variable of the considered term. So thanks to the emptiness lemma, we obtain:

$$\begin{array}{ll} & \lambda\pi\, z.\, (t[x := r])_z\ \pi \\ \equiv_\beta & \lambda\pi\, z.\, (t_z[x := r^\bullet]\ \pi) \odot (t_x[x := r^\bullet]\ \pi \gg\!= \lambda\rho.\, \emptyset) \end{array}$$

and hence, with a bit of multiset rewriting:

$$\lambda\pi\, z.\, (t[x := r])_z\ \pi \quad \equiv_\beta \quad \lambda\pi\, z.\, t_z[x := r^\bullet]\ \pi$$

which is precisely the equivalence we had to prove.

We now turn to the reverse equation. The left-hand side gives us:

$$\begin{array}{ll} & ((\lambda z.\, t)[x := r])_y \\ \equiv_\beta & (\lambda z.\, t[x := r])_y \\ \equiv_\beta & \lambda(z, \pi).\, (t[x := r])_y\ \pi \end{array}$$

so that by induction hypothesis, we get:

$$\begin{array}{ll} & ((\lambda z.\, t)[x := r])_y \\ \equiv_\beta & \lambda(z, \pi).\, (t_y[x := r^\bullet]\ \pi) \odot (t_x[x := r^\bullet]\ \pi \gg\!= \lambda\rho.\, r_y\ \rho) \\ \equiv_\beta & \lambda\pi.\, \mathtt{match}\ \pi\ \mathtt{with}\ (z, \pi) \mapsto (t_y[x := r^\bullet]\ \pi) \odot (t_x[x := r^\bullet]\ \pi \gg\!= \lambda\rho.\, r_y\ \rho) \end{array}$$

As z is free in r, we may apply a series of commutative cut rules by making the head pattern-matching dive into the term. This results in the following:

$$\begin{array}{ll} & ((\lambda z.\, t)[x := r])_y \\ \equiv_\beta & \lambda\pi.\, (\mathtt{match}\ \pi\ \mathtt{with}\ (z, \pi) \mapsto t_y[x := r^\bullet]\ \pi) \odot \\ & (\mathtt{match}\ \pi\ \mathtt{with}\ (z, \pi) \mapsto t_x[x := r^\bullet]\ \pi \gg\!= \lambda\rho.\, r_y\ \rho) \end{array}$$

but it turns out this is, up to introduction of a dummy β-redex on π on each side, the unfolding of the right-hand side we were looking for. Therefore we are done.

- Case $t\ u$.

  We have:

$$\begin{array}{lll} ((t\ u)[x := r])^\bullet & \equiv_\beta & (t[x := r]\ u[x := r])^\bullet \\ & \equiv_\beta & \mathrm{fst}\ (t[x := r])^\bullet\ (u[x := r])^\bullet \\ & \equiv_\beta & \mathrm{fst}\ t^\bullet[x := r^\bullet]\ u^\bullet[x := r^\bullet] \\ & \equiv_\beta & (t\ u)^\bullet[x := r^\bullet] \end{array}$$

by applying the induction hypotheses to t and u, giving us the expected result.

The reverse equation is quite technical, as this is the most complicated translation of all λ-terms. We derive each term and check that they agree. The left-hand side gives us:

$$
\begin{array}{ll}
 & ((t\ u)[x := r])_y \\
\equiv_\beta & (t[x := r]\ u[x := r])_y \\
\equiv_\beta & \lambda\pi.\ (\mathrm{snd}\ (t[x := r])^\bullet\ \pi\ (u[x := r])^\bullet \gg\!= \lambda\rho.\, (u[x := r])_y\ \rho) \odot \\
 & \qquad ((t[x := r])_y\ ((u[x := r])^\bullet, \pi)) \\
\equiv_\beta & \lambda\pi.\ (\mathrm{snd}\ t^\bullet[x := r^\bullet]\ \pi\ u^\bullet[x := r^\bullet] \gg\!= \lambda\rho.\, (u[x := r])_y\ \rho) \odot \\
 & \qquad ((t[x := r])_y\ (u^\bullet[x := r^\bullet], \pi)) \\
\equiv_\beta & \lambda\pi.\ (\mathrm{snd}\ t^\bullet[x := r^\bullet]\ \pi\ u^\bullet[x := r^\bullet] \gg\!= \lambda\rho.\, u_y[x := r^\bullet]\ \rho) \odot \\
 & \qquad (\mathrm{snd}\ t^\bullet[x := r^\bullet]\ \pi\ u^\bullet[x := r^\bullet] \gg\!= \lambda\rho.\, u_x[x := r^\bullet]\ \rho \gg\!= \lambda\chi.\, r_y\ \chi) \odot \\
 & \qquad (t_y[x := r^\bullet]\ (u^\bullet[x := r^\bullet], \pi)) \odot \\
 & \qquad (t_x[x := r^\bullet]\ (u^\bullet[x := r^\bullet], \pi) \gg\!= \lambda\rho.\, r_y\ \rho)
\end{array}
$$

To go from the penultimate line to the last one, we used the disjunction properties of the union operator w.r.t. the bind of the monad.

We now reduce the right-hand side of the equation:

$$
\begin{array}{ll}
 & \lambda\pi.\, ((t\ u)_y[x := r^\bullet]\ \pi) \odot ((t\ u)_x[x := r^\bullet]\ \pi \gg\!= \lambda\rho.\, r_y\ \rho) \\
\equiv_\beta & \lambda\pi.\ (\mathrm{snd}\ t^\bullet[x := r^\bullet]\ \pi\ u^\bullet[x := r^\bullet] \gg\!= \lambda\rho.\, u_y[x := r^\bullet]\ \rho) \odot \\
 & \qquad (t_y[x := r^\bullet]\ (u^\bullet[x := r^\bullet], \pi)) \odot \\
 & \qquad (\mathrm{snd}\ t^\bullet[x := r^\bullet]\ \pi\ u^\bullet[x := r^\bullet] \gg\!= \lambda\chi.\, u_x[x := r^\bullet]\ \chi \gg\!= \lambda\rho.\, r_y\ \rho) \odot \\
 & \qquad (t_x[x := r^\bullet]\ (u^\bullet[x := r^\bullet], \pi) \gg\!= \lambda\rho.\, r_y\ \rho)
\end{array}
$$

We used the left-disjunction of union over bind in the rewriting. But a careful gaze at the term obtained reveals that is no more than the result of the left-hand side, up to α-renaming and commutativity of the union operator. Thus we proved the property.

□

We are now able to prove the main theorem that states that β-equivalence is preserved by the translation.

Theorem 23 (Computational soundness)**.** *If* $t_1 \equiv_\beta t_2$ *then*

$$
\begin{array}{lll}
t_1{}^\bullet & \equiv_\beta & t_2{}^\bullet \\
t_{1x} & \equiv_\beta & t_{2x}
\end{array}
$$

for any variable x*.*

Proof. By induction on the β-equivalence. The contextual rules are all straightforward, because our translation is built by induction over the structure of the considered λ-term. Note that the one place in the forward translation were we need equating reverse translations is the λ-abstraction, which packs a reverse translation together with a forward translation. This is what forces us to also prove the second equation at the same time.

Hence we only describe the complicated case, that is, β-reduction itself.

We must therefore first show the first equation, i.e.:

$$((\lambda y.\, t)\ u)^\bullet \equiv_\beta (t[y := u])^\bullet$$

By deriving the left-hand side, we get:

$$\begin{array}{rcl} ((\lambda y.\, t)\ u)^\bullet & \equiv_\beta & \mathrm{fst}\ ((\lambda y.\, t^\bullet), (\lambda \pi\, y.\, t_y\ \pi))\ u^\bullet \\ & \equiv_\beta & (\lambda y.\, t^\bullet)\ u^\bullet \\ & \equiv_\beta & t^\bullet[y := u^\bullet] \\ & \equiv_\beta & (t[y := u])^\bullet \end{array}$$

which is precisely the equation we were looking for. The last line is obtained thanks to the substitution lemma.

We now turn to the second equation. We must show:

$$((\lambda y.\, t)\ u)_x \equiv_\beta (t[y := u])_x$$

From the left-hand side, we get:

$$\begin{array}{rcl} ((\lambda y.\, t)\ u)_x & \equiv_\beta & \lambda\pi.\ (\mathrm{snd}\ ((\lambda y.\, t^\bullet), (\lambda \pi\, y.\, t_y\ \pi))\ \pi\ u^\bullet \gg\!= \lambda\rho.\, u_x\ \rho) \odot \\ & & \qquad ((\lambda(y, \rho).\, t_x\ \rho)\ (u^\bullet, \pi)) \\ & \equiv_\beta & \lambda\pi.\, ((\lambda \pi\, y.\, t_y\ \pi)\ \pi\ u^\bullet \gg\!= \lambda\rho.\, u_x\ \rho) \odot ((\lambda(y, \rho).\, t_x\ \rho)\ (u^\bullet, \pi)) \\ & \equiv_\beta & \lambda\pi.\, (t_y[y := u^\bullet]\ \pi \gg\!= \lambda\rho.\, u_x\ \rho) \odot (t_x[y := u^\bullet]\ \pi) \end{array}$$

By applying the reverse substitution lemma, it turns out this is, up to a commutation of the union, the right-hand-side term. Thus we proved the equation.

□

As expected, the multiset-using Dialectica translation features a much more proof-theoretical behaviour than its more antique variants. There is something strange enough to be highlighted in the introduction of multisets. Indeed, as explained above, such a construction is not novel, because the Diller-Nahm translation is an instance of our own translation, with multisets being instantiated as finite sets. Nonetheless, the motivation for their introduction is not the same in the two cases.

- We wanted to recover preservation of β-equivalence;
- The Diller-Nahm variant aimed at getting rid of the requirement that orthogonality on atoms is decidable.

We find it rather peculiar that the same modification brings us with two sharp improvements over the traditional Dialectica. Yet, the most intriguing fact to us is that up to the present work, it seems that nobody ever realized that the Diller-Nahm variant also solved the β-equivalence preservation issue. We conjecture that the Dialectica community was not interested in the proof-theoretical properties *per se* of their various translations, preferring the study of their logical expressiveness instead.

9.3 KAM simulation

This section is dedicated to our main result on the Dialectica translation. It essentially explains what the translation is actually doing from the point of view of the dynamics of the program. We want to exploit the Curry-Howard isomorphism in the proof-to-program direction, and describe the target of the Dialectica translation as a λ-calculus enriched with some side-effect.

The essential tool on which this work is based is the Krivine abstract machine, already described at Section 2.4.1. We will use in quite an involved way the structures it uses.

To spill the beans quickly for the impatients, the Dialectica translation is a way to manipulate first-class observable stacks in a program, in a way similar to delimited continuations. Such stacks are constructed thanks to the reverse translations.

First, let us give a thorough explanation of what we mean by first-class stacks.

9.3.1 Stacks as first-class objects

We want to explain here how we fooled the reader by hiding a fact that was in plain sight, that is, we were already manipulating stacks in the Dialectica translation long before this section. The main idea is that the counter types $\mathbb{C}(-)$ from the Dialectica translation are actually the types of stacks accepting the corresponding witnesses. Let us give a bit of typing to make things clearer.

Definition 106 (Stack typing)**.** We define a typing system over stacks and environments. We write $\vdash \pi : A$ when the stack π has type A, and $\sigma \vdash \Gamma$ when the environment σ has type Γ, where Γ is a list of hypotheses as in the typing of terms. The derivations rules are given below.

$$\frac{}{\vdash \varepsilon : A} \qquad \frac{\vdash \pi : B \qquad \sigma \vdash \Gamma \qquad \Gamma \vdash t : A}{\vdash (t, \sigma) \cdot \pi : A \to B}$$

$$\frac{}{\cdot \vdash \cdot} \qquad \frac{\tau \vdash \Delta \qquad \Delta \vdash t : A \qquad \sigma \vdash \Gamma}{\sigma + (x := (t, \tau)) \vdash \Gamma, x : A}$$

One should not be upset by the convention we use in this definition. In particular, the fact we give a stack an arrow type should be thought of as coming through the looking glass: the actual type for the stack is the dual of an arrow, where the meaning of dual is the one of linear logic. Indeed, we have

$$(\llbracket A \to B \rrbracket_{\mathrm{n}})^{\perp} \equiv (!\llbracket A \rrbracket_{\mathrm{n}} \multimap \llbracket B \rrbracket_{\mathrm{n}})^{\perp} \equiv !\llbracket A \rrbracket_{\mathrm{n}} \otimes (\llbracket B \rrbracket_{\mathrm{n}})^{\perp}$$

and the stacks of type $A \to B$ are precisely given by a pair of a closure of type A and a stack of type B (i.e. the dual). Should we have taken another linear decomposition, we would then have recovered a different notion of stacks.

The idea that stacks matter and that their nature differs in the various calling conventions is pervasive in modern works, and can be found in a whole range of rather distinct systems. We consider that the current matter is in the wake of the various works mentioned below.

- Krivine's realizability, briefly recalled in Section 2.4.2. Although rooted in computation, it is a technique used to construct models of various theories, see for instance Krivine [70] or Miquel [82]. For a high-level synthetic presentation, the paper by Oliva and Streicher [90] is probably one of the best summaries. In classical realizability, stacks play a central rôle because the realization property on terms is a notion derived from the so-called falsity values, that are more primitive and defined in terms of stacks.

- The $\bar{\lambda}\mu\tilde{\mu}$-calculus, developed by Curien and Herbelin [35]. The main goal of this calculus is to highlight the duality that exists between objects in call-by-name and call-by-value. Stacks are given a first-class treatment, as they are actually dual to terms, and thus named *co-terms* in this setting.

- Munch's System L [84] is the offspring of the two previous objects. It can be seen as a simplification of $\bar{\lambda}\mu\tilde{\mu}$ guided by intuitions provided by classical realizability. It also shares a strong relationship with linear logic, and in particular polarization principles, up to the point where it can be considered as a term syntax for Girard's unified logic framework **LC** [43].

- This point of view is somehow implicitly present in Levy's Call-by-Push-Value [76] to some extent. This is due to the fact that CBPV and $\bar{\lambda}\mu\tilde{\mu}$ share the same purpose, namely being a universal framework to describe various calling conventions.

Something to be insisted on in this trend of relevant stacks is the property that stacks are not merely continuations, i.e. blackbox closures that can only be fired to escape the context, but instead inductive objects that can be observed and pattern-matched as well. We will make this intuition more concrete later on.

The interesting fact about stack typing is that the KAM reduction verifies a sort of a subject reduction property.

Proposition 62 (Subject reduction)**.** *Let t be a term, σ an environment and π a stack such that there exists Γ and A such that we have the following typing derivations.*

$$\Gamma \vdash t : A \qquad \vdash \pi : A \qquad \sigma \vdash \Gamma$$

Assume now that there exists a process $\langle (r, \tau) \mid \rho \rangle$ such that the reduction below holds.

$$\langle (t, \sigma) \mid \pi \rangle \longrightarrow \langle (r, \tau) \mid \rho \rangle$$

Then there exists Δ and B such that we have the following typing derivations.

$$\Delta \vdash r : B \qquad \vdash \rho : B \qquad \tau \vdash \Delta$$

Proof. By case analysis over the possible reductions.

- Assume $\langle (x, \sigma + (x := (t, \tau))) \mid \pi \rangle \longrightarrow \langle (t, \tau) \mid \pi \rangle$. Inverting the typing hypotheses on the first process ensures us that there exist some Γ, Δ and A such that we have the typing derivations below.

$$\sigma \vdash \Gamma \qquad \tau \vdash \Delta \qquad \Delta \vdash t : A \qquad \vdash \pi : A$$

  Therefore the reduced process is trivially typable by forgetting about the typability of σ.

- Assume $\langle (x, \sigma + (y := (t, \tau))) \mid \pi \rangle \longrightarrow \langle (x, \sigma) \mid \pi \rangle$. The typing hypotheses on the first process once inverted gives us the following typing derivations, for some Γ, Δ and A where $(x : A) \in \Gamma$.

$$\sigma \vdash \Gamma \qquad \tau \vdash \Delta \qquad \Delta \vdash t : B \qquad \vdash \pi : A$$

  The reduced process is trivially typable by forgetting this time about the typability of t.

- Assume $\langle (t\ u, \sigma) \mid \pi \rangle \longrightarrow \langle (t, \sigma) \mid (u, \sigma) \cdot \pi \rangle$. This means we have the following typing derivations for some Γ, A and B.

$$\Gamma \vdash t : A \to B \qquad \Gamma \vdash u : A \qquad \vdash \pi : B \qquad \sigma \vdash \Gamma$$

  But then we can derive straightforwardly

$$\vdash (u, \sigma) \cdot \pi : A \to B$$

  so that we found the required typability.

- Assume $\langle (\lambda x.\, t, \sigma) \mid (u, \tau) \cdot \pi \rangle \longrightarrow \langle (t, \sigma + (x := (u, \tau))) \mid \pi \rangle$. We have therefore the typing derivations below for some Γ, Δ, A and B.

$$\Gamma, x : A \vdash t : B \qquad \vdash \pi : B \qquad \Delta \vdash t : A \qquad \tau \vdash \Delta \qquad \sigma \vdash \Gamma$$

  The desired typing is obtained by repacking everything, resulting in the following:

$$\sigma + (x := (u, \tau)) \vdash \Gamma, x : A$$

□

Let us see how we can translate typed stacks from the KAM into counters of the Dialectica translation. We first need a small technical apparatus to get rid of closures, but this is not difficult.

Definition 107 (Closure flattening)**.** If t is a term and σ an environment, we define the flattening of the closure (t, σ) as the term $t \ltimes \sigma$, inductively defined on σ as follows.

$$\begin{array}{rcl} t \ltimes \cdot & := & t \\ t \ltimes \sigma + (x := (u, \tau)) & := & (t[x := u \ltimes \tau]) \ltimes \sigma \end{array}$$

We will only be manipulating closed terms in the translation, so that the exact choice of when we do the substitution in the second case does not matter, as x is not inadvertently captured.

As expected, the flattening of a closure behaves well with respect to typing.

Proposition 63. *Let t and σ be respectively a term and a closure such that we have the following typing derivations.*

$$\Gamma \vdash t : A \qquad \sigma \vdash \Gamma$$

Then $\vdash t \ltimes \sigma : A$.

Proof. By induction on σ.

- If the environment is empty, then the statement is trivially true.
- Assume now that we have the typing derivation below.

$$\Gamma, x : B \vdash t : A \qquad \sigma + (x := (u, \tau)) \vdash \Gamma, x : B$$

The typing of the environment provides us with a certain Δ such that $\Delta \vdash u : B$ and $\tau \vdash \Delta$. Therefore, by induction hypothesis on τ, we have that $\vdash u \ltimes \tau : B$. By substituting, we immediately get that

$$\Gamma \vdash t[x := u \ltimes \tau] : A$$

from which we conclude using the induction hypothesis on σ.

□

Now we have gathered all ingredients at hand to be able to dispell the growing impatience of the reader: we will show how to properly translate stacks from the KAM through the Dialectica translation. It is rather simple actually, because call-by-name stacks are themselves rather dull, at least when only considering the λ-calculus. Call-by-name stacks of the negative fragment are indeed a purely first-order datastructure. Thence the translation is very straightforward, and it interprets stacks as pairs.

Definition 108 (Stack translation)**.** Let π be a KAM stack. We define the $\lambda^{\times +}$-term $\pi^\bullet$ by induction over π.

$$\begin{array}{rcl} \varepsilon^\bullet & := & \varepsilon \\ ((t, \sigma) \cdot \pi)^\bullet & := & ((t \ltimes \sigma)^\bullet, \pi^\bullet) \end{array}$$

Here ε is a special λ-variable reserved for that use.

Note that we do not bother with the stack bottom, in the stack typing as well as in the stack translation: in the first case we arbitrarily decided that we could give it any type, while in the second we just translated it as a special variable. We could have, and should have been more precise if we were interested in translations actually messing with the stack bottom, as it is the case for instance in call-by-name delimited continuations. This issue is not relevant in this section, so we will just forget about it.

As we were hoping for, the stack translation is compatible with the typing translation induced by the typing Dialectica translation on terms.

Proposition 64 (Stack typing soundness)**.** *Let π be a stack such that $\vdash \pi : A$. Then there exist a type R such that*

$$\varepsilon : \mathbb{C}(\llbracket R \rrbracket_{\mathrm{n}}) \vdash \pi^{\bullet} : \mathbb{C}(\llbracket A \rrbracket_{\mathrm{n}})$$

Proof. By induction on the typing derivation of π.

- If $\vdash \varepsilon : A$, then $R := A$ and this is trivially true.
- If $\vdash (t, \sigma) \cdot \pi : A \to B$, by combining the typing preservation of flattening and the Dialectica typing soundness, we get that

  $$\vdash (t \ltimes \sigma)^{\bullet} : \mathbb{W}(\llbracket A \rrbracket_{\mathrm{n}})$$

  and from the induction hypothesis on π we know that there exists some R such that

  $$\varepsilon : \mathbb{C}(\llbracket R \rrbracket_{\mathrm{n}}) \vdash \pi^{\bullet} : \mathbb{C}(\llbracket B \rrbracket_{\mathrm{n}})$$

  Recall that $\mathbb{C}(\llbracket A \to B \rrbracket_{\mathrm{n}}) := \mathbb{W}(\llbracket A \rrbracket_{\mathrm{n}}) \times \mathbb{C}(\llbracket B \rrbracket_{\mathrm{n}})$, so that we immediately get the typing derivation below.

  $$\varepsilon : \mathbb{C}(\llbracket R \rrbracket_{\mathrm{n}}) \vdash ((t, \sigma) \cdot \pi)^{\bullet} : \mathbb{C}(\llbracket A \to B \rrbracket_{\mathrm{n}})$$

  This is what we were looking for.

□

The result above ensures us that the KAM stacks have an actual existence in the Dialectica translation. Indeed, they exist in the target language as well-typed objects. There is a slight mismatch though, because we have to flatten all closures in the translation. This results in a small loss of information. As we will see, the main result of this section crucially uses closures, but it only does so on the head closure of the machine, not on the ones coming from the stack. This is why this mismatch is of little consequence.

9.3.2 Realizability interpretation

Now that we have given a real existence to stacks that otherwise only exist under the form of ghostly objects known as contexts, we can start trying to explain how they appear in the Dialectica translation.

Notice that indeed, the Dialectica translation makes stacks appear on intuitionistic objects: recall that the witness translation of the arrow type is defined as

$$\mathbb{W}([\![A \to B]\!]_{\mathrm{n}}) := (\mathbb{W}([\![A]\!]_{\mathrm{n}}) \to \mathbb{W}([\![B]\!]_{\mathrm{n}})) \times (\mathbb{C}([\![B]\!]_{\mathrm{n}}) \to \mathbb{W}([\![B]\!]_{\mathrm{n}}) \to \mathfrak{M}\,\mathbb{C}([\![A]\!]_{\mathrm{n}}))$$

so that stacks gatecrash into the translation even in purely intuitionistic types. This means that if we wish to give a realizability account of the Dialectica translation, we better have to explain what rôle those stacks actually have.

The main hint that should be leading us is the strange-looking equality found in the substitution lemma for the reverse translation. Recall that we have indeed

$$(t[x := r])_y \equiv_\beta \lambda\pi.\, (t_y[x := r^\bullet]\ \pi) \odot (t_x[x := r^\bullet]\ \pi \gg\!= \lambda\rho.\, r_y\ \rho)$$

when $x \neq y$ and y not free in r. While the left side of the union is what should be the naturally expected term, the right one is a bit at odds with the remainder. What does r_y has to do with all this? The very existence of the right-hand side term hints at the fact that something fancy is happening during substitution. Likewise, the translation of the λ-abstraction suggests that we do care about free variables: the $-_x$ terms obviously deal with them.

Precisely, the KAM features itself a fancy substitution: instead of substituting terms upfront when encountering a β-redex, the KAM delays the substitution until a substituted variable ends in head position of the machine, while hitherto keeping the substitutive term in a closure. In particular, closures retain their free variables, giving more structure to substituted terms.

Looking at types is even more interesting: the π and ρ in the resulting term of the substitution are themselves typed with a counter type, i.e. those variables stand for a stack in the KAM.

Its featured first-class stacks and delayed substitution support the idea that the KAM is probably one of the best object to describe the dynamic behaviour of the Dialectica translation.

We will now phrase the simulation result in a few suggestive words, then formally state and prove it, to finally come back on it afterwards.

Theorem 24 (Pseudo-KAM simulation)**.** *Given a λ-term t, the term t_x captures all the stacks that appear in front of the head variable x in the reduction of $\langle t \mid \pi \rangle$.*

The above claim is not formal, but it gives the broad idea. We now turn to the formal property, which is less intelligible than the previous handwavish version. We first need some formal definitions to ease writing it out formally.

Notation 10. Let m be a term[1] and t some other term. Then we evocatively write $t \in m$ if there exists some term n such that

$$m \equiv_\beta \{t\} \odot n$$

This notation should be understood as the natural way to denote abstract multiset membership. It indeed features most of the properties we would be expecting from the membership relation.

Because we have to handle terms with free variables packed with environments, we need to ensure a generalized version of Barendregt's conditions on closures.

Definition 109 (Hereditary α-conversion)**.** Let σ be an environment and t be a term. We can lift α-conversion to the closure (t, σ) in a natural way, i.e. by regarding all variables of t defined in σ as being bound, and recursively over the closures defined in σ.

Definition 110 (Hereditary freedom)**.** Let σ be an environment and x be a variable. We say that x is not free in σ if it does not appear in any term bindings of σ as well as being recursively not free in any environment τ present in σ.

Remark 14. An alternative definition for a variable x not being hereditarily free in an environment σ can be given as: x is not free in σ if for all term t, if x is a free variable of $t \ltimes \sigma$, then it is also a free variable of t.

Definition 111 (Closure translation)**.** If t is a λ-term and σ an environment, we write $t \ltimes \sigma^\bullet$ by analogy of the flattening for the term defined by induction on σ as follows.

$$\begin{array}{rcl} t \ltimes (\cdot)^\bullet & := & t \\ t \ltimes (\sigma + (x := (u, \tau)))^\bullet & := & (t[x := (u \ltimes \tau)^\bullet]) \ltimes \sigma^\bullet \end{array}$$

We suppose that there is no capture of variables when doing such a substitution (otherwise we α-convert the closure (t, σ)).

This suggestive notation commutes with the interpretation as expected.

Proposition 65. *Let t be a term and σ an environment. Then*

$$(t \ltimes \sigma)^\bullet \equiv_\beta t^\bullet \ltimes \sigma^\bullet$$

Proof. By induction on σ and repetitive application of the substitution lemma. □

We finish by phrasing the simulation theorem below.

Theorem 25 (Formal KAM simulation)**.** *Let t be a λ-term, σ an KAM environment and π a KAM stack, such that (t, σ) is closed. Assume some variable x not free in σ nor in any closure of π.*

If $\langle (t, \sigma) \mid \pi \rangle \longrightarrow^* \langle (x, \tau) \mid \rho \rangle$ for some τ and ρ, then $\rho^\bullet \in (t_x \ltimes \sigma^\bullet)\ \pi^\bullet$.

[1] Hopefully denoting a multiset, but the definition makes sense for any term.

Proof. By induction on the length of the reduction, and case analysis of the first reduction rule.

- Assume a reduction of length zero on $\langle (x, \sigma) \mid \pi \rangle$. In this case, we have

  $$(x_x \ltimes \sigma^\bullet)\ \pi^\bullet \equiv_\beta (\lambda\pi.\, \{\pi\})\ \pi^\bullet \equiv_\beta \{\pi^\bullet\}$$

  and obviously $\pi^\bullet \in \{\pi^\bullet\}$ because

  $$\{\pi^\bullet\} \equiv_\beta \{\pi^\bullet\} \odot \emptyset$$

- Assume $\langle (y, \sigma + (y := (u, \tau))) \mid \pi \rangle \longrightarrow \langle (u, \tau) \mid \pi \rangle$. By the freedom hypothesis on x with respect to the original environment, we know that x cannot occur in head position during the reduction of $\langle (u, \tau) \mid \pi \rangle$ (up to α-renaming, it does not appear anywhere in this process).

  There are therefore two distinct cases: either $x = y$, or $x \neq y$. The first case has already been handled above. The second case is vacuously true, because no x appears in head of the machine at all. Note that in this case, we have

  $$(y_x \ltimes \sigma^\bullet)\ \pi^\bullet \equiv_\beta (\lambda\pi.\, \emptyset)\ \pi^\bullet \equiv_\beta \emptyset$$

- Assume $\langle (\lambda y.\, t, \sigma) \mid (u, \tau) \cdot \pi \rangle \longrightarrow \langle (t, \sigma + (y := (u, \tau))) \mid \pi \rangle$. This case is straightforward. Indeed, by induction hypothesis we know that

  $$\rho^\bullet \in (t_x \ltimes (\sigma + (y := (u, \tau)))^\bullet)\ \pi^\bullet$$

  as x is still free in the closure and in the stack of the reduced process if we take y to be fresh. But then we have

  $$\begin{array}{rll} ((\lambda y.\, t)_x \ltimes \sigma^\bullet)\ ((u, \tau) \cdot \pi)^\bullet & \equiv_\beta & ((\lambda(y, \pi).\, t_x) \ltimes \sigma^\bullet)\ ((u \ltimes \tau)^\bullet, \pi^\bullet) \\ & \equiv_\beta & (\lambda(y, \pi).\, (t_x \ltimes \sigma^\bullet))\ ((u \ltimes \tau)^\bullet, \pi^\bullet) \\ & \equiv_\beta & (t_x[y := (u \ltimes \tau)^\bullet] \ltimes \sigma^\bullet)\ \pi^\bullet \\ & \equiv_\beta & (t_x \ltimes (\sigma + (y := (u, \tau)))^\bullet)\ \pi^\bullet \end{array}$$

  so that we can conclude directly that

  $$\rho^\bullet \in ((\lambda y.\, t)_x \ltimes \sigma^\bullet)\ ((u, \tau) \cdot \pi)^\bullet$$

- Assume $\langle (t\ u, \sigma) \mid \pi \rangle \longrightarrow \langle (t, \sigma) \mid (u, \sigma) \cdot \pi \rangle$. This is the most complicated case. Let us first unfold the Dialectica interpretation of the process.

  $$\begin{array}{rl} & ((t\ u)_x \ltimes \sigma^\bullet)\ \pi^\bullet \\ \equiv_\beta & (\mathrm{snd}\ (t^\bullet \ltimes \sigma^\bullet)\ \pi^\bullet\ (u^\bullet \ltimes \sigma^\bullet) \gg\!= \lambda\rho.\, (u_x \ltimes \sigma^\bullet)\ \rho) \odot ((t_x \ltimes \sigma^\bullet)\ (u^\bullet \ltimes \sigma^\bullet, \pi^\bullet)) \end{array}$$

There are two ways x may appear in head position of the process: either in a reduct of t or in a reduct of u. It cannot appear anywhere else because of the freedom hypothesis made on x. Each of these cases will correspond to one side of the union in the reduced term above.

– In the first case, we can α-rename the closure (u, σ) into $(\hat{u}, \hat{\sigma})$ such that x does not appear in u. Then there exists a reduction

$$\langle (t, \sigma) \mid (\hat{u}, \hat{\sigma}) \cdot \pi \rangle \longrightarrow^* \langle (x, \tau) \mid \rho \rangle$$

which is strictly shorter than the reduction we were considering, and which behaves identically as the α-renaming is transparent for the KAM. The x variable is also free in $(\hat{u}, \hat{\sigma}) \cdot \pi$ so that we can apply the induction hypothesis on this process. Therefore

$$\rho^\bullet \in (t_x \ltimes \sigma^\bullet)\ ((\hat{u}, \hat{\sigma}) \cdot \pi)^\bullet$$

Because we only α-renamed the closure, we still have

$$(u \ltimes \sigma)^\bullet \equiv_\beta u^\bullet \ltimes \sigma^\bullet \equiv \hat{u}^\bullet \ltimes \hat{\sigma}^\bullet \equiv_\beta (\hat{u} \ltimes \hat{\sigma})^\bullet$$

from which we immediately get

$$\rho^\bullet \in (t_x \ltimes \sigma^\bullet)\ (u^\bullet \ltimes \sigma^\bullet, \pi^\bullet)$$

and in particular

$$\rho^\bullet \in ((t\ u)_x \ltimes \sigma^\bullet)\ \pi^\bullet$$

by a simple unfolding of the definition of $\in$ and by using the reduction of the term given at the beginning.

– In the second case, x appears because the closure (u, σ) came in head position. To do so, (t, σ) must have reduced to a λ-abstraction. In particular, a quick analysis of the possible reductions shows that there exists a term t_0, a fresh variable x_0 and an environment σ_0 extending σ such that

$$\langle (t, \sigma) \mid (u, \sigma) \cdot \pi \rangle \longrightarrow^* \langle (\lambda x_0.\, t_0, \sigma_0) \mid (u, \sigma) \cdot \pi \rangle$$

followed by the pattern of reduction given below

$$\begin{array}{rcl} \langle (\lambda x_0.\, t_0, \sigma_0) \mid (u, \sigma) \cdot \pi \rangle & \longrightarrow & \langle (t_0, \sigma_0 + (x_0 := (u, \sigma))) \mid \pi \rangle \\ & \longrightarrow^* & \langle (x_0, \sigma_0 + (x_0 := (u, \sigma))) \mid \chi \rangle \\ & \longrightarrow & \langle (u, \sigma) \mid \chi \rangle \\ & \longrightarrow^* & \langle (x, \tau) \mid \rho \rangle \end{array}$$

for some stack χ, where we can safely assume that x is not free in χ (otherwise we α-rename it without affecting the considered reduction). But we can now apply the induction hypothesis on the reduction sequence of (u, σ) against χ, so that we know that:

$$\rho^\bullet \in (u_x \ltimes \sigma^\bullet)\ \chi^\bullet$$

Likewise, we can apply the induction hypothesis to the process $\langle (t_0, \sigma_0 + (x_0 := (u, \sigma))) \mid \pi \rangle$ but this time considering the reverse translation for the variable x_0. Thus we know that:

$$\chi^\bullet \in (t_{0x_0} \ltimes (\sigma_0 + (x_0 := (u, \sigma)))^\bullet)\ \pi^\bullet$$

Notice that we were able to apply the induction hypothesis in both cases because the considered reduction was strictly shorter than the original one, as witnessed by the first and third transitions which ensure that we do at least one reduction step in each case.

From the reduction of the closure (t, σ), we also get immediately that

$$t \ltimes \sigma \equiv_\beta (\lambda x_0.\, t_0) \ltimes \sigma_0$$

by the soundness of the KAM itself. Some steps of rewriting eventually give us that

$$\begin{array}{rcl} t^\bullet \ltimes \sigma^\bullet & \equiv_\beta & (t \ltimes \sigma)^\bullet \\ & \equiv_\beta & ((\lambda x_0.\, t_0) \ltimes \sigma_0)^\bullet \\ & \equiv_\beta & (\lambda x_0.\, t_0)^\bullet \ltimes \sigma_0{}^\bullet \\ & \equiv_\beta & (\lambda x_0.\, t_0, \lambda \pi\, x_0.\, t_{0x_0}\ \pi) \ltimes \sigma_0{}^\bullet \end{array}$$

and in particular, if we look at the left-hand side of the reduction of the original application process, we obtain

$$\begin{array}{rcl} \mathrm{snd}\ (t^\bullet \ltimes \sigma^\bullet)\ \pi^\bullet\ (u^\bullet \ltimes \sigma^\bullet) & \equiv_\beta & ((t_{0x_0}[x_0 := (u^\bullet \ltimes \sigma^\bullet)]) \ltimes \sigma_0{}^\bullet)\ \pi^\bullet \\ & \equiv_\beta & (t_{0x_0} \ltimes (\sigma_0 + (x_0 := (u, \sigma)))^\bullet)\ \pi^\bullet \end{array}$$

so that, in the end,

$$\chi^\bullet \in \mathrm{snd}\ (t^\bullet \ltimes \sigma^\bullet)\ \pi^\bullet\ (u^\bullet \ltimes \sigma^\bullet)$$

By unfolding the $\in$ notation and using the distributivity properties of the $\gg\!=$ operator, we easily conclude that

$$\rho^\bullet \in \mathrm{snd}\ (t^\bullet \ltimes \sigma^\bullet)\ \pi^\bullet\ (u^\bullet \ltimes \sigma^\bullet) \gg\!= \lambda \rho.\, (u_x \ltimes \sigma^\bullet)\ \rho$$

and therefore

$$\rho^\bullet \in ((t\ u)_x \ltimes \sigma^\bullet)\ \pi^\bullet$$

which was what was to be proved.

□

9.3.3 When Krivine meets Gödel

The moment has come when we have the global picture of the dynamic contents of the Dialectica translation. The $(-)_x$ translation allows to capture the current stack each time the variable x reaches head position. As this may occur several times, we have to return a multiset instead of only one stack. In addition, we also need the current stack to somehow delimit our returned stacks. The access to the current stack may be essential, because it contains terms that may in turn reach head position, taking control over the whole machine. This gives a very higher-orderish flavour to the Dialectica translation.

The fact that beneath the Dialectica translation was waiting for so long an interpretation deeply relying on the properties of the KAM keeps us in awe. Who knew that the seemingly ad-hoc definitions of the original Dialectica were hiding side-effects akin to delimited continuations?

Let us now step back a moment and try to explain things we presented but with a new light.

Proposition 66 (Substitution lemma revisited)**.** *Let us look again at the substitution lemma on the reverse translations. Recall that we have the equation below.*

$$(t[x := r])_y \equiv_\beta \lambda\pi.\, (t_y[x := r^\bullet]\ \pi) \odot (t_x[x := r^\bullet]\ \pi \gg\!= \lambda\rho.\, r_y\ \rho)$$

It tells us no more than that the accesses to the variable y in the term $t[x := r]$ can be classified in two kinds of accesses.

- *The left-hand side of the union corresponds to the accesses to y in the term $t[x := r]$ itself. This was expected.*
- *The substitution may nonetheless have created new accesses to y: these are precisely described by the accesses to y in r when r is to be substituted by x when it comes in head position. But this is exactly what is constructed by the right-hand side term*

  $$t_x[x := r^\bullet]\ \pi \gg\!= \lambda\rho.\, r_y\ \rho$$

  Indeed, the left argument of the bind constructs all accesses to x in t, that are then passed to the accessors of y created by r, which is the right argument of the bind.

We can also revisit the historical presentation of the Dialectica translation, and compare it with our hybrid of Diller-Nahm and Curry-Howardesque translation. The main differences lies in the way the historical translation gets rid of the abstract multiset datastructure.

- Instead of returning an empty multiset, the historical translation choose a canonical placeholder materialized by the dummy term.
- When having to make the union of two multisets, the historical translation dynamically determines one the two terms to be picked, according to the decidable orthogonality.

The main issue with this process is that the selection occurring when picking one of the two terms is incorrect with respect to the operational semantics: it may choose a dummy term just because in the current context, it wins against the considered term, even though it is not a proper stack.

An intriguing fact is that even if our translation is call-by-name, in the sense that the stacks we operate on are the one of the KAM, the resulting calculus by itself is by no means call-by-name, i.e. the translation preserves *all* β-equivalences, not only the ones valid in a call-by-name reduction strategy. This translation is therefore call-by-name w.r.t. the notion of calculus, not to the one of strategies.

9.3.4 An unfortunate mismatch

While the simulation theorem is particularly informative, and can be seen as the uncovering of the dynamical nature of the Dialectica translation when seen as a KAM translation rather than a term one, there is still an itchy spot just under our fingers.

A reader granted with a keen eye may already have realized that there is a huge discrepancy between the Dialectica translation and its KAM interpretation. The simulation theorem relates indeed reduction sequences of the KAM with *multisets* of stacks. This is were there is a catch.

- The KAM is fully sequential. With the reduction rules we provided, there is even only one possible reduction path. This means that the stacks we observe in the accesses to variables ought to be ordered with respect to the sequential order by which they appear along the reduction.
- The Dialectica is not sequential. The use of multisets, as free commutative monoids by excellence, forgoes any hope to recover some order relating it to the KAM.

One could naively use the natural candidate for free non-commutative monoids, namely, lists. Unluckily, this is not possible.

Proposition 67. *If we implement abstract multisets by standard lists, disregarding the required equivalence axioms, then the Dialectica translation does not preserve β-equivalence anymore.*

Proof. We essentially use the commutativity axioms throughout the proof of the preservation, especially in the substitution lemma. Most of the rewriting rules would make no sense using lists.

□

This discrepancy of a list-using Dialectica can actually be explained itself through the KAM. We can indeed get such an intuition by looking at rules dealing with the $\odot$ operator, that corresponds to contraction from the point of view of linear logic. For instance, let us consider the reverse translation of the application rule.

$$(t\ u)_x \quad := \quad \lambda\pi.\,(\mathrm{snd}\ t^\bullet\ \pi\ u^\bullet \gg\!= \lambda\rho.\,u_x\ \rho) \odot (t_x\ (u^\bullet, \pi))$$

Assume we had a sequential simulation theorem, stating that for any term t, t_x produces the list of accessed stack in the order induced by the reduction of the KAM. Then the above translation does not comply with this interpretation. Indeed, in the KAM, the term $t\ u$ can make two types of accesses to x, either from t or from u. A quick analysis already done in the proof of the simulation theorem shows that a reduction for the term $t\ u$ is of the form

$$\begin{array}{lll}
\langle (t\ u, \sigma) \mid \pi \rangle & \longrightarrow & \langle (t, \sigma) \mid (u, \sigma) \cdot \pi \rangle \\
& \longrightarrow^* & \langle (\lambda x_0.\, t_0, \sigma_0) \mid (u, \sigma) \cdot \pi \rangle \quad (\star) \\
& \longrightarrow & \langle (t_0, \sigma_0 + (x_0 := (u, \sigma))) \mid \pi \rangle \\
& \longrightarrow^* & \langle (x_0, \sigma_0 + (x_0 := (u, \sigma))) \mid \chi \rangle \quad (\star) \\
& \longrightarrow & \langle (u, \sigma) \mid \chi \rangle \\
& \longrightarrow^* & \langle (x, \tau) \mid \rho \rangle \quad (\star) \\
& \ldots & \\
& \longrightarrow^* & \langle (x_0, \sigma_0 + (x_0 := (u, \sigma))) \mid \chi \rangle \quad (\star) \\
& \ldots &
\end{array}$$

where the $(\star)$ stands for reductions where a x from t may appear in head position and hence trigger a growth of the returned list as a side-effect.

One can immediately remark that there is something really wrong here with respect to the translation, when using lists instead of multisets. The accesses of x in $t\ u$ can be indeed interleaved between t and u: they can be realized before t itself evaluates to a function, then some part of that function may access x, or maybe x_0 is accessed before instead, and so on. This has nothing to do with the Dialectica interpretation which sharply separates the two sources of accesses: the left side of the union corresponds to terms from t and the right side to those from u. No such interleaving is present in the translation.

This is actually even worse. No such interleaving is *possible* in the Dialectica translation. To cope with this sequentialization, as witnessed by the above pattern of reduction, one would need to know the order of relative accesses to the free variables in the term (i.e. x vs. x_0 in our example). The Dialectica is totally oblivious of this issue, because all terms t_x are constructed in parallel for each variable x, regardless of the other variables. There is no further dependency on the free variables of the source term.

If we wished to overcome this defect, we would need a more complicated scheme of translation acknowledging the reduction order induced by the KAM. At the present time, we are not aware of such a translation. If it exists, we believe it to be both quite complicated and very higher-order. It would indeed require a way to speak of closures as first-class objects: while Dialectica treats stacks on par with terms, closures are the great absent of the translation, although they do exist in the KAM. To handle closures in a typed way, we believe that we need to resort to complex very functional objects hiding existential type.

We conjecture that this phenomenon is actually a consequence of the linear factorization of the Dialectica translation, rather than a true defect of the translation itself. As expressed by Proposition 13, linear logic is essentially a model of commutative monads. Requiring the translation to respect the sequentiality of the KAM would break this commutativity. That is why we also conjecture that any sequential Dialectica would *not* factor through linear logic, even on the type translation only.

We propose nonetheless translations cousins of the Dialectica translation at Section 12 that do arise from linear decompositions and that we hope to help us designing a sequential Dialectica.

9.3.5 A quantitative interpretation?

We would like to take a step backward, and look at the Dialectica translation from two very distinct point of views, that may turn out to be the same from our higher-level point of view. We allege that some bridge should be drawn between two alien communities, to the benefit of all.

The starting remark of this little digression that we would like to be enlightening is the simple fact that we can see the Dialectica translation as a way to *count* things. By counting, we mean measuring the complexity of a program for some precise metric.

Indeed, the simulation theorem teaches us that given a term t and a stack π, we can retrieve all accesses to a given variable x by t all along the reduction of the process $\langle t \mid \pi \rangle$. This is given as a multiset m by the translated term t_x. In particular, by just considering the length of this multiset, we can actually compute the number of accesses to x in the reduction of $\langle t \mid \pi \rangle$. This is, per se, a valuable complexity metric of our program. In the process, we lost the contents of the multiset, but if we are only interested by the counting, this is no matter.

This only would be already interesting, but there is more. It turns out that it is well-known that the Dialectica can count, but through two very different paths.

- First, quite an important part of Kohlenbach's book on proof mining [66] is dedicated to the study of majorizability and its implementation in the Dialectica translation, under the name of the *monotone functional interpretation*. The main idea consists in replacing the witness terms by higher-order arithmetical functions subject to a certain realizability condition. The multiset operations are in turn replaced by arithmetical operations. In particular, the multiset union is changed into a maximum.

  The purpose of this translation is to provide bounds on the growth of arithmetical functions, thus the inherent underlying counting approach.

  Even though we do not fully master this translation, we believe it to be losely based on an erasure of the multiset translation.

- The second instance of a *déjà-vu* counting Dialectica can be found in the trend of recent work on the so-called *quantitative semantics* (see for instance Breuvart and Pagani [23], Gaboardi et al. [25] and Laird et al. [73]).

In those works, the authors aim at extracting quantitative properties from the semantics, thanks to a clever use of linear logic. The paper [73] in particular designs such a semantics atop of a variation of coherence spaces.

It is folklore that coherence spaces are a degenerate instance of the double-glueing construction, which is actually itself the generalization of De Paiva categories, from which our Dialectica proceeds.

We believe those two families of objects to be essentially the same, albeit seen through the prism of two distinct communities' practices and history. We hope that the present work can close the gap between both. We also hope that reunifying both will help in ameliorating the overall design process coming from the two traditions, as well as fixing tickling defects present in one but not on the other, and conversely.

- The monotone functional interpretation is a purely logical tool. It only cares about numbers (arithmetic and analysis, primarily) and not about the underlying programs, as well as about richer proof-as-program types.
- The metrics used in quantitative semantics are purely first-order data, be it numbers or a structural semiring as in [25], thus preventing the typability of many perfectly valid more higher-order programs. The multiset-producing terms of the Dialectica translation are much richer than a mere integer.

There is therefore probably a lot to learn from the other side of the bank.

10 Variants of the Dialectica translation

> It is possible to get "as much mustard as wanted" from a mustard watch.
>
> *Yann-Joachim Ringard about logical extensions.*

In this part, we consider extensions and variations of the translation presented previously. Indeed, because we are climbing over the shoulders of linear logic, it is not difficult to tweak the considered call-by-name decomposition or to choose a completely different one to get different calling conventions for free.

Note that we will be more concise in the proofs of this chapter, because most of them are essentially the same as the previous chapter, or at least following the very same arguments.

10.1 Call-by-name positive connectives

Up to now, we only translated the purely negative fragment of the call-by-name λ-calculus, that is, our one and only type was the call-by-name arrow. Positive datatypes are often overlooked for various reasons, and we find it a pity. They feature indeed properties sharply contrasting with the negative fragment that deserve to be studied for their own sake. This is why we dedicate this section to their study in the renewed Dialectica translation. We will be focusing on the following extension of the simple types defined at Section 9.2.

$$A, B := \ldots \mid 1 \mid A \times B \mid 0 \mid A + B$$

We believe that one of the main reasons of their mistreatment is that they often raise issues in call-by-name. It is quite well-known indeed that the purely negative fragment is of the realms of the vast land of λ-calculi which is the most devoid of any issue. In general, problems tend to occur whenever one wishes to add inductive datatypes to a call-by-name system. Indeed, the introduction of what would correspond to positive datatypes from a polarized point of view often result in the scattered appearance of itchy spots.

Luckily, in our case, we already scratched them thoroughly enough, because we actually *already* introduced positive datatypes in the target calculus, to be able to construct stacks. It is indeed a standard fact that stacks corresponding to some terms have the opposite polarity, so that our all-negative terms must have all-positive stacks.

Therefore, adding positives should not be so painful. Yet, this is not totally uninteresting, because we still introduce alternations of polarity, which can give interesting insights into both polarized logic and the Dialectica translation.

10.1.1 Dynamics

The λ-terms associated to the positive datatypes are the same as for the $\lambda^{\times+}$-calculus. We will be taking the very same typing rules and derivations. It means that our extended translation can be seen as a translation from the $\lambda^{\times+}$-calculus into itself.

10.1.2 Extended KAM

As for the KAM, it is easy to extend it with positive connectives. The fact that we have coexisting polarities forces us to have stacks of negative polarity, that is, stacks that force the head of their term before going any further. Thus, the current closure is no more the sole responsible of the reduction, as the stack may also be in charge. This also implies that our stacks are now higher-order objects.

Definition 112 (Extended KAM stacks)**.** The stacks of the KAM are extended as follows.

$$\begin{array}{rll} \pi, \rho := & \ldots & \\ \mid & () \mapsto (t, \sigma) \cdot \pi & \\ \mid & (x, y) \mapsto (t, \sigma) \cdot \pi & \\ \mid & [\cdot] & \\ \mid & ([x \mapsto t_1 \mid y \mapsto t_2], \sigma) \cdot \pi & \end{array}$$

Each of these stacks correspond to the forcing of a term whose head must reduce to a given inductive constructor. We have four positive types, resulting in four distinct stacks constructions.

Definition 113 (Extended KAM rules)**.** The reduction rules of the KAM are then completed with the ones given below.

Forcing rules:

$$\begin{array}{rcl} \langle (\texttt{match}\ t\ \texttt{with}\ () \mapsto u, \sigma) \mid \pi \rangle & \longrightarrow & \langle (t, \sigma) \mid () \mapsto (u, \sigma) \cdot \pi \rangle \\ \langle (\texttt{match}\ t\ \texttt{with}\ (x, y) \mapsto u, \sigma) \mid \pi \rangle & \longrightarrow & \langle (t, \sigma) \mid (x, y) \mapsto (u, \sigma) \cdot \pi \rangle \\ \langle (\texttt{match}\ t\ \texttt{with}\ [x \mapsto u_1 \mid y \mapsto u_2], \sigma) \mid \pi \rangle & \longrightarrow & \langle (t, \sigma) \mid ([x \mapsto u_1 \mid y \mapsto u_2], \sigma) \cdot \pi \rangle \\ \langle (\texttt{match}\ t\ \texttt{with}\ [\cdot], \sigma) \mid \pi \rangle & \longrightarrow & \langle (t, \sigma) \mid [\cdot] \rangle \end{array}$$

Return rules:

$$\begin{array}{rcl} \langle ((), \sigma) \mid () \mapsto (u, \tau) \cdot \pi \rangle & \longrightarrow & \langle (u, \tau) \mid \pi \rangle \\ \langle ((t_1, t_2), \sigma) \mid (x, y) \mapsto (u, \tau) \cdot \pi \rangle & \longrightarrow & \\ & & \langle (u, \tau + (x := (t_1, \sigma)) + (y := (t_2, \sigma))) \mid \pi \rangle \\ \langle (\textbf{inl}\ t, \sigma) \mid ([x \mapsto u_1 \mid y \mapsto u_2], \tau) \cdot \pi \rangle & \longrightarrow & \langle (u_1, \tau + (x := (t, \sigma))) \mid \pi_1 \rangle \\ \langle (\textbf{inr}\ t, \sigma) \mid ([x \mapsto u_1 \mid y \mapsto u_2], \tau) \cdot \pi \rangle & \longrightarrow & \langle (u_2, \tau + (x := (t, \sigma))) \mid \pi_2 \rangle \end{array}$$

The first four rules encode the elimination of positive datatypes: whenever a `match` occurs in head position, its content should be forced, which was the purpose of the extended stacks. The matched term is then reduced until it evaluates to a partial value, that is, a term whose head node is an inductive constructor.

The four remaining rules are the return rules of the machine. Whenever a term evaluates to a partial value, this value is destructured by pattern-matching, and control is returned back to the caller, i.e. the matching branch of the pattern-matching that led to this machine state.

Note that there is no return reduction associated to the stack $[\cdot]$. Indeed, it corresponds to a proof of the empty type, which should not occur if our system is consistent.

To help the reader, we can provide a typing system for those extended stacks, inspired by the one provided at Section 9.3.1.

Definition 114 (Stack typing, extended)**.** Stacks for inductive connectives are given the following typing derivations.

$$\frac{\sigma \vdash \Gamma \qquad \Gamma \vdash u : C \qquad \vdash \pi : C}{\vdash () \mapsto (u, \sigma) \cdot \pi : 1} \qquad \frac{}{\vdash [\cdot] : 0}$$

$$\frac{\sigma \vdash \Gamma \qquad \Gamma, x : A, y : B \vdash u : C \qquad \vdash \pi : C}{\vdash (x, y) \mapsto (u, \sigma) \cdot \pi : A \times B}$$

$$\frac{\sigma \vdash \Gamma \qquad \Gamma, x : A \vdash u_1 : C \qquad \Gamma, y : B \vdash u_2 : C \qquad \vdash \pi : C}{\vdash ([x \mapsto u_1 \mid y \mapsto u_2], \sigma) \cdot \pi : A + B}$$

The subject reduction property stated at Section 9.3.1 is still valid, though we will not dwell on it.

10.1.3 Type translation

Recovering the type translation is only a matter of letting ourselves guide by the linear logic decomposition. The call-by-name linear decomposition is characterized by the fact we add an exponential at each change of polarity.

Especially, we need to interleave a ! connective under each positive connective. This corresponds to the fact that the terms under positive constructors are thunked, and therefore are not forced when pattern-matching over the resulting term.

Definition 115 (Call-by-name decomposition, extended)**.** We extend the call-by-name decomposition $[\![\cdot]\!]_{\mathrm{n}}$ into linear logic to positive datatypes as follows.

$$\begin{array}{lcl} [\![1]\!]_{\mathrm{n}} & := & 1 \\ [\![A \times B]\!]_{\mathrm{n}} & := & ![\![A]\!]_{\mathrm{n}} \otimes ![\![B]\!]_{\mathrm{n}} \\ [\![0]\!]_{\mathrm{n}} & := & 0 \\ [\![A + B]\!]_{\mathrm{n}} & := & ![\![A]\!]_{\mathrm{n}} \oplus ![\![B]\!]_{\mathrm{n}} \end{array}$$

We can then look at the resulting types once we apply the Dialectica translation on those linear types. For the easiness of manipulation of translated terms, as in the case of sequent translation, we will define those types only isomorphically to the types we would obtain from the raw translation.

Proposition 68 (Type translation)**.** *The witness translation has the following values.*

$$\begin{array}{lcl} \mathbb{W}([\![1]\!]_{\mathrm{n}}) & := & 1 \\ \mathbb{W}([\![A \times B]\!]_{\mathrm{n}}) & := & \mathbb{W}([\![A]\!]_{\mathrm{n}}) \times \mathbb{W}([\![B]\!]_{\mathrm{n}}) \\ \mathbb{W}([\![0]\!]_{\mathrm{n}}) & := & 0 \\ \mathbb{W}([\![A + B]\!]_{\mathrm{n}}) & := & \mathbb{W}([\![A]\!]_{\mathrm{n}}) + \mathbb{W}([\![B]\!]_{\mathrm{n}}) \end{array}$$

Yet, for counter types, we will tweak the definitions and take the isomorphisms below as proper definitions, regardless of the fact that we were seeing up to now the $\mathbb{C}([\![\cdot]\!]_{\mathrm{n}})$ *translation as a composition of two distinct translations.*

$$\begin{array}{lcl} \mathbb{C}([\![1]\!]_{\mathrm{n}}) & \cong & 1 \\ \mathbb{C}([\![A \times B]\!]_{\mathrm{n}}) & \cong & \left\{ \begin{array}{c} \mathbb{W}([\![A]\!]_{\mathrm{n}}) \times \mathbb{W}([\![B]\!]_{\mathrm{n}}) \to \mathfrak{M}\, \mathbb{C}([\![A]\!]_{\mathrm{n}}) \\ \times \\ \mathbb{W}([\![A]\!]_{\mathrm{n}}) \times \mathbb{W}([\![B]\!]_{\mathrm{n}}) \to \mathfrak{M}\, \mathbb{C}([\![B]\!]_{\mathrm{n}}) \end{array} \right. \\ \mathbb{C}([\![0]\!]_{\mathrm{n}}) & \cong & 1 \\ \mathbb{C}([\![A + B]\!]_{\mathrm{n}}) & \cong & (\mathbb{W}([\![A]\!]_{\mathrm{n}}) \to \mathfrak{M}\, \mathbb{C}([\![A]\!]_{\mathrm{n}})) \times (\mathbb{W}([\![B]\!]_{\mathrm{n}}) \to \mathfrak{M}\, \mathbb{C}([\![B]\!]_{\mathrm{n}})) \end{array}$$

As in the previous chapter, we will just write $\mathbb{W}(A)$ for $\mathbb{W}([\![A]\!]_{\mathrm{n}})$ in this section when the context makes clear that A is an intuitionistic type.

10.1.4 Term translation

In order to write out our extended translation, we should base ourselves both on the typing of terms as well as on the intuitions provided by the simulation theorem: t_x should indeed capture all accesses to x in t, so that we must not forget about any when designing the translation.

Definition 116 (Term translations, extended)**.** The term translations are mutually inductively extended as follows. The direct translation are all straightforward.

$$\begin{array}{lcl} ()^\bullet & := & () \\ (t, u)^\bullet & := & (t^\bullet, u^\bullet) \\ (\mathbf{inl}\ t)^\bullet & := & \mathbf{inl}\ t^\bullet \\ (\mathbf{inr}\ t)^\bullet & := & \mathbf{inr}\ t^\bullet \\ (\mathtt{match}\ t\ \mathtt{with}\ () \mapsto u)^\bullet & := & \mathtt{match}\ t^\bullet\ \mathtt{with}\ () \mapsto u^\bullet \\ (\mathtt{match}\ t\ \mathtt{with}\ (x, y) \mapsto u)^\bullet & := & \mathtt{match}\ t^\bullet\ \mathtt{with}\ (x, y) \mapsto u^\bullet \\ (\mathtt{match}\ t\ \mathtt{with}\ [\cdot])^\bullet & := & \mathtt{match}\ t^\bullet\ \mathtt{with}\ [\cdot] \\ (\mathtt{match}\ t\ \mathtt{with}\ [x \mapsto u_1 \mid y \mapsto u_2])^\bullet & := & \\ & & \mathtt{match}\ t^\bullet\ \mathtt{with}\ [x \mapsto u_1^\bullet \mid y \mapsto u_1^\bullet] \end{array}$$

The interest comes from the reverse translations, which clearly make appear the higher-order nature of stacks accepting positive types.

$$
\begin{array}{lcl}
()_z & := & \lambda\pi.\, \varnothing \\
(t,u)_z & := & \lambda(\varphi,\psi).\ (\varphi\ (t^\bullet, u^\bullet) \gg\!= \lambda\chi.\, t_z\ \chi) \odot \\
 & & \qquad (\psi\ (t^\bullet, u^\bullet) \gg\!= \lambda\chi.\, u_z\ \chi) \\
(\mathbf{inl}\ t)_z & := & \lambda(\varphi,\psi).\, \varphi\ t^\bullet \gg\!= \lambda\chi.\, t_z\ \chi \\
(\mathbf{inr}\ t)_z & := & \lambda(\varphi,\psi).\, \psi\ t^\bullet \gg\!= \lambda\chi.\, t_z\ \chi \\
(\mathtt{match}\ t\ \mathtt{with}\ () \mapsto u)_z & := & \lambda\pi.\ (\mathtt{match}\ t^\bullet\ \mathtt{with}\ () \mapsto u_z\ \pi) \odot \\
 & & \qquad (t_z\ ()) \\
(\mathtt{match}\ t\ \mathtt{with}\ (x,y) \mapsto u)_z & := & \lambda\pi.\ (\mathtt{match}\ t^\bullet\ \mathtt{with}\ (x,y) \mapsto u_z\ \pi) \odot \\
 & & \qquad (t_z\ ((\lambda(x,y).\, u_x\ \pi), (\lambda(x,y).\, u_y\ \pi))) \\
(\mathtt{match}\ t\ \mathtt{with}\ [\cdot])_z & := & \lambda\pi.\ (\mathtt{match}\ t^\bullet\ \mathtt{with}\ [\cdot]) \odot \\
 & & \qquad (t_z\ ()) \\
(\mathtt{match}\ t\ \mathtt{with}\ [x \mapsto u_1 \mid y \mapsto u_2])_z & := & \lambda\pi.\ (\mathtt{match}\ t^\bullet\ \mathtt{with}\ [x \mapsto u_{1z}\ \pi \mid y \mapsto u_{2z}\ \pi]) \odot \\
 & & \qquad (t_z\ ((\lambda x.\, u_{1x}\ \pi), (\lambda y.\, u_{2y}\ \pi)))
\end{array}
$$

We clearly see a pattern repeating in the reverse translations defined above: while introduction rules are essentially constructing a CPS translation of the future pattern-matching they will be applied to, elimination rules construct a two-part object. According to our variable-observing interpretation, the left part corresponds to the uses of z in the pattern branches, while the right part corresponds to the uses of z in the term being matched. Interestingly enough, we see that the right part is always built upon the same sort of CPS that was featured in the introduction rules.

This CPS-like translation can be intuitively explained as follows. Let us consider for instance the term (t,u) and let us look closely at the $(t,u)_z$ translation. This means we want to track when z is used in this pair. Unluckily, even in call-by-name, a pair is somehow a value, i.e. an object that would be inert if we put it the KAM together with an empty stack. Therefore, we need its surrounding context, which is the one deciding how to use the various components. And this context is necessarily a functional object, because it eventually boils down to some pattern-matching context of the form $\mathtt{match}\ \cdot\ \mathtt{with}\ (x,y) \mapsto r$.

But there is hope: the only information we need about this context is the way it uses x and y in r, that is, r_x and r_y. Looking at the type of those terms allows us to understand that this is exactly the purpose of the φ and ψ terms in the translation above.

We state the typing soundness below, and give a bit of detail of the proof to help the reader understand what is going on thanks to the typing of the considered terms.

Proposition 69 (Typing soundness, extended)**.** *If* $\Gamma \vdash t : A$*, then*

$$\mathbb{W}(\Gamma) \vdash t^\bullet : \mathbb{W}(A)$$
$$\mathbb{W}(\Gamma) \vdash t_z : \mathbb{C}(A) \to \mathfrak{M}\,\mathbb{C}(U)$$

when $(z : U) \in \Gamma$.

Proof. By induction on the typing derivation. The direct translations are... direct, so we only look at the reverse translations. We assume in all cases that $(z : U) \in \Gamma$.

- Case $()$. This is trivial.
- Case (t, u). We have $\Gamma \vdash (t, u) : A \times B$. By induction hypothesis, we have all of the following derivations.

  $$\mathbb{W}(\Gamma) \vdash t^\bullet : \mathbb{W}(A)$$
  $$\mathbb{W}(\Gamma) \vdash u^\bullet : \mathbb{W}(B)$$
  $$\mathbb{W}(\Gamma) \vdash t_z : \mathbb{C}(A) \to \mathfrak{M}\,\mathbb{C}(U)$$
  $$\mathbb{W}(\Gamma) \vdash u_z : \mathbb{C}(B) \to \mathfrak{M}\,\mathbb{C}(U)$$

  Recall that

  $$\mathbb{C}(A \times B) \quad := \quad \begin{cases} \mathbb{W}(A) \times \mathbb{W}(B) \to \mathfrak{M}\,\mathbb{C}(A) \\ \times \\ \mathbb{W}(A) \times \mathbb{W}(B) \to \mathfrak{M}\,\mathbb{C}(B) \end{cases}$$

  so that we only have to prove the two following derivations

  $$\mathbb{W}(\Gamma), \varphi : \mathbb{W}(A) \times \mathbb{W}(B) \to \mathfrak{M}\,\mathbb{C}(A) \vdash \varphi\ (t^\bullet, u^\bullet) \gg\!= \lambda\chi.\, t_z\ \chi : \mathfrak{M}\,\mathbb{C}(U)$$
  $$\mathbb{W}(\Gamma), \psi : \mathbb{W}(A) \times \mathbb{W}(B) \to \mathfrak{M}\,\mathbb{C}(B) \vdash \psi\ (t^\bullet, u^\bullet) \gg\!= \lambda\chi.\, u_z\ \chi : \mathfrak{M}\,\mathbb{C}(U)$$

  which is just a matter of plugging all the proofs recovered from the induction hypotheses.
- Case **inl** t (and its symmetric). We have

  $$\mathbb{C}(A + B) \quad := \quad \begin{cases} \mathbb{W}(A) \to \mathfrak{M}\,\mathbb{C}(A) \\ \times \\ \mathbb{W}(B) \to \mathfrak{M}\,\mathbb{C}(B) \end{cases}$$

  so that it is sufficient to prove that

  $$\mathbb{W}(\Gamma), \varphi : \mathbb{W}(A) \to \mathfrak{M}\,\mathbb{C}(A) \vdash \varphi\ t^\bullet \gg\!= \lambda\chi.\, t_z\ \chi : \mathfrak{M}\,\mathbb{C}(U)$$

  which is easily obtained by applying the induction hypotheses.

- Case $\texttt{match}\ t\ \texttt{with}\ () \mapsto u$. We have to prove that

$$\mathbb{W}(\Gamma), \pi : \mathbb{C}(C) \vdash \texttt{match}\ t^\bullet\ \texttt{with}\ () \mapsto u_z\ \pi : \mathfrak{M}\,\mathbb{C}(U)$$
$$\mathbb{W}(\Gamma) \vdash t_z\ () : \mathfrak{M}\,\mathbb{C}(U)$$

  which are easily proved by applying the induction hypotheses.

- The other pattern-matching are proved similarly, in two parts, corresponding to the two derivations to prove above. The upper derivation is always the same and comes from the typing of the term being eliminated, while the second one creates a stack for the term being eliminated by using the branches of the match.
  - For $A \times B$, $(\lambda(x,y).\, u_x\ \pi, \lambda(x,y).\, u_y\ \pi)$ is a term of type

$$\mathbb{C}(A \times B) := (\mathbb{W}(A) \times \mathbb{W}(B) \to \mathfrak{M}\,\mathbb{C}(A)) \times (\mathbb{W}(A) \times \mathbb{W}(B) \to \mathfrak{M}\,\mathbb{C}(B))$$

  - For 0, $()$ is a term of type

$$\mathbb{C}(0) := 1$$

  - For $A + B$, $(\lambda x.\, u_{1x}\ \pi, \lambda y.\, u_{2y}\ \pi)$ is a term of type

$$\mathbb{C}(A + B) := (\mathbb{W}(A) \to \mathfrak{M}\,\mathbb{C}(A)) \times (\mathbb{W}(B) \to \mathfrak{M}\,\mathbb{C}(B))$$

  Note that the variables introduced by these stacks are free in the terms coming from the various pattern branches, so that choosing the same name is important.

□

It is rather insightful to observe that in the proof above, we silently feed the free variables of the term of the branches thanks to both pattern-matching and stack construction. It is indeed a nice property that we do not need to deeply manipulate our terms to make them blend into the translation: everything is done effortlessly thanks to free variables.

10.1.5 Computational soundness

As in the case of the pure λ-calculus, we still have the substitution lemma, and thus the preservation of β-equivalence through the translation. We need to add the commutative cuts corresponding to the additives 0 and $A + B$ though.

Such commutative cuts are straightforwardly adapted from the multiplicative case, and are given below.

Definition 117 (Additive commutative cuts)**.** The additive commutative cuts are given by the following equalities, with the usual conditions on variable capture.

$$
\begin{array}{lll}
\texttt{match}\ t\ \texttt{with}\ [\cdot] & \equiv_\beta & \emptyset \\
\texttt{match}\ t\ \texttt{with}\ [\cdot] & \equiv_\beta & (\texttt{match}\ t\ \texttt{with}\ [\cdot]) \odot (\texttt{match}\ t\ \texttt{with}\ [\cdot]) \\
\texttt{match}\ t\ \texttt{with}\ [\cdot] & \equiv_\beta & \{\texttt{match}\ t\ \texttt{with}\ [\cdot]\} \\
\texttt{match}\ t\ \texttt{with}\ [\cdot] & \equiv_\beta & \texttt{match}\ t\ \texttt{with}\ [\cdot] \gg\!= f \\
\texttt{match}\ t\ \texttt{with}\ [x \mapsto \emptyset \mid y \mapsto \emptyset] & \equiv_\beta & \emptyset \\
\texttt{match}\ t\ \texttt{with}\ [x \mapsto u_1 \odot u_2 \mid y \mapsto u_1 \odot u_2] & \equiv_\beta & \\
\multicolumn{3}{r}{(\texttt{match}\ t\ \texttt{with}\ [x \mapsto u_1 \mid y \mapsto u_1]) \odot (\texttt{match}\ t\ \texttt{with}\ [x \mapsto u_2 \mid y \mapsto u_2])} \\
\texttt{match}\ t\ \texttt{with}\ [x \mapsto \{u_1\} \mid x \mapsto \{u_2\}] & \equiv_\beta & \{\texttt{match}\ t\ \texttt{with}\ [x \mapsto u_1 \mid y \mapsto u_2]\} \\
\texttt{match}\ t\ \texttt{with}\ [x \mapsto u_1 \gg\!= f \mid y \mapsto u_2 \gg\!= f] & \equiv_\beta & \texttt{match}\ t\ \texttt{with}\ [x \mapsto u_1 \mid y \mapsto u_2] \gg\!= f
\end{array}
$$

Observe that the rules for the elimination of the empty type are worrisome. Indeed, they may create ill-typed terms, as there are no restrictions on the way we use them. There is no reason that an elimination of the absurdity has type $\mathfrak{M}\,A$ for some A, and yet, $\texttt{match}\ t\ \texttt{with}\ [\cdot]$ is convertible to $\emptyset$. This is actually an issue related to the fact that we are in an a priori type-free setting, and these commutative cuts are unsound in general in this setting. This is a standard phenomenon, as empty types only make sense in an explicitly typed system.

We will pretend it is not a real issue, because we just want to show that we can work out the untyped reduction proof in a way similar to the previous chapter. As soon as we get back into the realm of typedness, we are a little safer: all the rewriting steps we are using in the computational soundness lemma are indeed type-safe. The Dialectica translation and its equivalence properties can be adapted to an explicitly typed system without much effort, but that would require a lengthy description that we cannot afford here. We will therefore ignore these issues by considering that we would eventually only consider typed terms, even if the translation is defined over untyped terms.

Proposition 70 (Emptiness lemma, extended)**.** *For any λ-term t and any variable x not free in t, we have the equivalence below.*

$$t_x \equiv_\beta \lambda\pi.\,\emptyset$$

Proof. By induction on terms, as usual. □

Proposition 71 (Substitution lemma, extended)**.** *The substitution lemma holds, i.e. for any terms t and r, any variables x, y s.t. $x \neq y$ and x is not free in r, we have the following equivalences.*

$$(t[x := r])^\bullet \equiv_\beta t^\bullet[x := r^\bullet]$$

$$(t[x := r])_y \equiv_\beta \lambda\pi.\,(t_y[x := r^\bullet]\ \pi) \odot (t_x[x := r^\bullet]\ \pi \gg\!= \lambda\rho.\,r_y\ \rho)$$

Proof. By induction on terms. We will only look at the case of the introduction and elimination of products. The other cases are treated alike. The forward translation commutes with everything, so it is trivial here. We are only left with the reverse translations.

- Let us show that

$$((t,u)[x := r])_z \equiv_\beta \lambda\pi.\,((t,u)_z[x := r^\bullet]\ \pi) \odot ((t,u)_x[x := r^\bullet]\ \pi \gg\!= \lambda\rho.\,r_z\ \rho)$$

  when x is not free in r and distinct of z. The left-hand side gives us

$$\begin{array}{lll}
 & ((t,u)[x := r])_z & \\
\equiv_\beta & (t[x := r], u[x := r])_z & \\
\equiv_\beta & \lambda(\varphi,\psi).\ & (\varphi\ ((t[x := r])^\bullet, (u[x := r])^\bullet) \gg\!= \lambda\chi.\,(t[x := r])_z\ \chi) \odot \\
 & & (\psi\ ((t[x := r])^\bullet, (u[x := r])^\bullet) \gg\!= \lambda\chi.\,(u[x := r])_z\ \chi) \\
\equiv_\beta & \lambda(\varphi,\psi).\ & (\varphi\ (t^\bullet[x := r^\bullet], u^\bullet[x := r^\bullet]) \gg\!= \lambda\chi.\,t_z[x := r^\bullet]\ \chi) \odot \\
 & & (\varphi\ (t^\bullet[x := r^\bullet], u^\bullet[x := r^\bullet]) \gg\!= \lambda\chi.\,t_x[x := r^\bullet]\ \chi \gg\!= \lambda\rho.\,r_z\ \rho) \odot \\
 & & (\psi\ (t^\bullet[x := r^\bullet], u^\bullet[x := r^\bullet]) \gg\!= \lambda\chi.\,u_z[x := r^\bullet]\ \chi) \odot \\
 & & (\psi\ (t^\bullet[x := r^\bullet], u^\bullet[x := r^\bullet]) \gg\!= \lambda\chi.\,u_x[x := r^\bullet]\ \chi \gg\!= \lambda\rho.\,r_z\ \rho)
\end{array}$$

  It is easy to see that the right-hand side evaluates to the same value up to rearrangement of the unions.

- We must show that

$$\begin{array}{c}
((\texttt{match}\ t\ \texttt{with}\ (x,y) \mapsto u)[w := r])_z \\
\equiv_\beta \\
\lambda\pi.\,(\texttt{match}\ t\ \texttt{with}\ (x,y) \mapsto u)_z[w := r^\bullet]\ \pi \\
\odot ((\texttt{match}\ t\ \texttt{with}\ (x,y) \mapsto u)_w[w := r^\bullet]\ \pi \gg\!= \lambda\rho.\,r_z\ \rho)
\end{array}$$

  when w is not free in r and distinct of z. This case is very similar to the case of λ-abstraction, viz. the trick consists in using the emptiness lemma to get rid of spurious accesses to x and y in r in the left-hand side where they are not free.

□

Corollary 2. *The translation preserves β-equivalence.*

10.1.6 Stack translation

As in the purely negative case, we can translate stacks of the extended KAM to terms of the right type through Dialectica. This is really not difficult at all, although it is a little more involved than in the λ-calculus fragment, because stacks are now higher-order objects. We define the translation just below.

Definition 118 (Extended stack translation)**.** The stack translation is inductively defined as follows.

$$\begin{array}{rcl}
((() \mapsto (t, \sigma) \cdot \pi)^\bullet & := & () \\
((x, y) \mapsto (t, \sigma) \cdot \pi)^\bullet & := & (\lambda(x, y).\, (t \ltimes \sigma)_x\ \pi^\bullet, \lambda(x, y).\, (t \ltimes \sigma)_y\ \pi^\bullet) \\
[\cdot]^\bullet & := & () \\
(([x \mapsto t \mid y \mapsto u], \sigma) \cdot \pi)^\bullet & := & (\lambda x.\, (t \ltimes \sigma)_x\ \pi^\bullet, \lambda y.\, (u \ltimes \sigma)_y\ \pi^\bullet)
\end{array}$$

We easily get the preservation of typing through the translation, as expected.

Proposition 72 (Stack typing soundness, extended)**.** *The typing soundness for stacks holds.*

Proof. By induction on the stacks, as usual. This is obvious for nullary connectives, and just a matter of unfolding the translation for binary connectives.

□

Remark 15. There is something rather important to be said about this translation. If we look at the translation of additives, it is rather obvious that what we would like to have is not a tuple of size zero and two for the types 0 and $A + B$ respectively, but rather an term starting with an elimination of the term being observed. Formally, we would like to have

$$\begin{array}{rcl}
[\cdot]^\bullet & := & \lambda p.\, \texttt{match}\ p\ \texttt{with}\ [\cdot] \\
(([x \mapsto t \mid y \mapsto u], \sigma) \cdot \pi)^\bullet & := & \lambda p.\, \texttt{match}\ p\ \texttt{with}\ [x \mapsto (t \ltimes \sigma)_x\ \pi^\bullet \mid y \mapsto (u \ltimes \sigma)_y\ \pi^\bullet]
\end{array}$$

While this is not a real problem in the case of the empty type, this is really troublesome for the sum type. Indeed, in absence of dependent elimination on this sum type, the translated stack is not typable in a simply-typed calculus. Remark that indeed each branch have a distinct type, namely the type of multisets of stacks corresponding to each side of the sum type.

Likewise, the stack translation for the unit type is a bit at odds both with the intuition we get from linear logic as well as the fact it totally forgets about the continuation of the machine. Indeed, we know from linear logic that $\mathbb{C}(1) := \bot$, so that we should use this continuation to do something useful, instead of just dropping it without further consideration. The translation should rather be of the form

$$((() \mapsto (t, \sigma) \cdot \pi)^\bullet := \lambda p.\, \texttt{match}\ p\ \texttt{with}\ () \mapsto k\ \pi$$

for some well-chosen term k of type $\mathbb{C}(C) \to \bot$ constructed from the term t. Alas, there is no information for such a bottom type coming from t in this precise translation.

These two points will be discussed in the barebone version of the Dialectica translation of Section 12.2, as well as its dependent version.

10.1.7 Extended KAM simulation

Thanks to the construction guided by the linear decomposition, we easily mimic the arguments from the purely negative case to recover the same property. That is, the extended Dialectica has the same relationship with the extended KAM as its purely negative counterpart with the usual KAM, i.e. the simulation theorem is provable.

Theorem 26 (Extended simulation theorem)**.** *The simulation theorem holds for the extended translation with respect to the extended KAM.*

Proof. Once again, it is sufficient to reason by induction on the length of the reduction and to do a case analysis on the first reduction. Let us assume that the process $\langle (t, \sigma) \mid \pi \rangle$ we are considering is reducing to some state of the form $\langle (z, \tau) \mid \rho \rangle$ where z is a variable. We make the usual freedom assumption on z w.r.t. σ and π and we would like to show that

$$\rho^\bullet \in (t_z \ltimes \sigma^\bullet)\ \pi^\bullet$$

- Suppose $\langle (\texttt{match}\ t\ \texttt{with}\ () \mapsto u, \sigma) \mid \pi \rangle \longrightarrow \langle (t, \sigma) \mid () \mapsto (u, \sigma) \cdot \pi \rangle$. There are two ways z may come in head position of the machine, either coming from t or from u. But recall that

  $$((\texttt{match}\ t\ \texttt{with}\ () \mapsto u)_z \ltimes \sigma^\bullet)\ \pi^\bullet \equiv_\beta ((t_z \ltimes \sigma^\bullet)\ ()) \odot ((u_z \ltimes \sigma^\bullet)\ \pi^\bullet)$$

  If z comes from a reduction of t, we know by induction hypothesis applied to the reduced state that $\rho^\bullet \in (t_z \ltimes \sigma^\bullet)\ ()$. Otherwise, we know that

  $$\begin{aligned} \langle (t, \sigma) \mid () \mapsto (u, \sigma) \cdot \pi \rangle &\longrightarrow^* \langle ((), \sigma_0) \mid () \mapsto (u, \sigma) \cdot \pi \rangle \\ &\longrightarrow \langle (u, \sigma) \mid \pi \rangle \\ &\longrightarrow^* \langle (z, \tau) \mid \rho \rangle \end{aligned}$$

  so that we conclude by applying the induction hypothesis to the process $\langle (u, \sigma) \mid \pi \rangle$.

- Suppose $\langle ((), \sigma) \mid () \mapsto (u, \sigma') \cdot \pi \rangle \longrightarrow \langle (u, \sigma') \mid \pi \rangle$. In this case, z cannot appear in the reduction path of the machine, because we supposed the current stack did not contain it. Therefore, this is vacuously true.

- Suppose $\langle (\texttt{match}\ t\ \texttt{with}\ (x, y) \mapsto u, \sigma) \mid \pi \rangle \longrightarrow \langle (t, \sigma) \mid (x, y) \mapsto (u, \sigma) \cdot \pi \rangle$. As usual, there are two ways that z appears in head position, either from t or from u. We can conclude similarly to the elimination of the unit type by just looking at the translation of the set of collected stacks.

  $$\begin{gathered} ((\texttt{match}\ t\ \texttt{with}\ (x, y) \mapsto u)_z \ltimes \sigma^\bullet)\ \pi^\bullet \\ \equiv_\beta \\ ((t_z \ltimes \sigma^\bullet)\ ((\lambda(x, y).\, u_x\ \pi^\bullet), (\lambda(x, y).\, u_y\ \pi^\bullet))) \odot \\ (\texttt{match}\ (t^\bullet \ltimes \sigma^\bullet)\ \texttt{with}\ (x, y) \mapsto (u_z \ltimes \sigma^\bullet)\ \pi^\bullet) \end{gathered}$$

All instances of z coming from t will be provided by the first component, while all instances of z coming from u will be given by the second component.

- Suppose $\langle((t,u),\sigma) \mid (x,y) \mapsto (r,\sigma') \cdot \pi\rangle \longrightarrow \langle(r,\sigma' + (x := (t,\sigma)) + (y := (u,\sigma))) \mid \pi\rangle$. The variable z may appear in head position only from t and u. This means that we have either a reduction of the form

  $$\langle(r,\sigma'+(x := (t,\sigma))+(y := (u,\sigma))) \mid \pi\rangle \longrightarrow \langle(x,\sigma'+(x := (t,\sigma))+(y := (u,\sigma))) \mid \chi\rangle$$

  or of the form

  $$\langle(r,\sigma'+(x := (t,\sigma))+(y := (u,\sigma))) \mid \pi\rangle \longrightarrow \langle(y,\sigma'+(x := (t,\sigma))+(y := (u,\sigma))) \mid \chi\rangle$$

  But unfolding the term encoding the collected set gives us

  $$\begin{gathered}((t,u)_z \ltimes \sigma^\bullet)\ ((x,y) \mapsto (r,\sigma') \cdot \pi)^\bullet \\ \equiv_\beta \\ ((r_x \ltimes (\sigma' + (x := (t,\sigma)) + (y := (u,\sigma)))^\bullet)\ \pi^\bullet \gg\!= \lambda\chi.\,(t_z \ltimes \sigma^\bullet)\ \chi) \odot \\ ((r_y \ltimes (\sigma' + (x := (t,\sigma)) + (y := (u,\sigma)))^\bullet)\ \pi^\bullet \gg\!= \lambda\chi.\,(u_z \ltimes \sigma^\bullet)\ \chi)\end{gathered}$$

  so that, by applying the same argument as for the λ-abstraction case, we conclude by taking the first (resp. second) component in the first (resp. second) reduction scenario.

- The additive cases are treated just the same, so we will not give the details.

□

There is actually nothing magical in the fact that everything is provable so seamlessly: this is due to the fact that our translation factors through a call-by-name linear decomposition, and that the (extended) KAM is the archetypal machine for call-by-name, up to the point that we can just see it appearing behind the Lafont-Reus-Streicher CPS [72], itself inspired by the linear decomposition.

10.1.8 Recursive types

The positive types we presented here are all non-recursive. This is not representative of most general positive inductive types which feature some form of recursion. A canonical example is the type of unary integers $\mathbb{N}$, which is generated by the inductive definition

$$\mathbb{N} := \mathbf{0} : \mathbb{N} \mid \mathbf{S} : \mathbb{N} \to \mathbb{N}$$

and whose elimination rules can be defined either through a recursor similar in content to the term representing induction principle from **HA**, or directly as a mix of a guarded fixpoint and a match construct.

While we do not provide an interpretation for this kind of recursive types, we assume it to be doable, extrapolating from the treatment of integers in the historical translation. One nice feature of the aforementioned translation can be materialized by the fact that the translation itself reduces a potentially infinite fixpoint into a finite functional datatype, akin to the impredicative encoding used for instance by System F [47]. We also believe that it would give interesting insights in the treatment of co-inductive objects in the KAM. This is left for future work.

10.2 A glimpse at the resulting logic

Thanks to our multiset-using reformulation of the Dialectica translation, as well as the computational intuition provided by the simulation theorem, we can revisit the interpretation of the two additional semi-classical axioms featured by the historical Dialectica translation.

10.2.1 Dialectica as a side-effect

The first thing we can do is adding the additional expressive power present in the target language as new constructions in the source language. This is a standard construction, and it corresponds from the programming point of view as considering the direct style arising from the translation, that is, adding native side-effects in the source language.

Morally, the one effect added by our translation could be summarized by the translation of the arrow, which is the main diverging point from the usual proof-as-program interpretation. That would require the definition of a type of stacks accepting A, say $\sim A$ [1], subject to certain conditions, together with a principle of the following form.

$$(A \to B) \to A \to \sim B \to \mathfrak{M}\,(\sim A)$$

Intuitively, one can consider that the type $\sim A$ will have the type translations below.

$$\begin{array}{rcl} \mathbb{W}(\sim A) & := & \mathbb{C}(A) \\ \mathbb{C}(\sim A) & := & \mathbb{W}(A) \end{array}$$

With such translations, it seems that the principle is reduced to a mere projection, because the arrow $A \to B$ already comes with the necessary structure to recover the remaining of the principle.

Even without further making explicit the properties of the $\sim(-)$ type constructor induced by the above translations, we are already bumping into various more or less subtle issues. First, there is an obvious problem: the source calculus does not feature multisets, and we did not provide a way to translate them in the target calculus. We can try to work around this by representing them with an impredicative encoding, even though that is not that easy.

Moreover, the fact that we are call-by-name raises an issue of its own: except for the arrow $A \to B$ given as an argument to the principle, the other arrows should not be

[1] This notation is inspired by the involutive negation described in [85], where it has a similar rôle.

considered as call-by-name arrows, but rather as pure meta-arrows. In particular, they should *not* be translated as $\mathbb{W}((-) \to (-))$. Otherwise, these arrows would have to come with a reverse component. More formally, assuming we discriminate between pure arrows $- \to -$ and effectful arrows $- \dot{\to} -$, our principle should therefore have type

$$(A \dot{\to} B) \to A \to {\sim} B \to \mathfrak{M}\,({\sim} A)$$

This means that we need to define this principle by means of a native combinator rather than as an axiomatic call-by-name function. This issue is akin to the fact that, in call-by-value, effectful constructions such as a `try-with` clause

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash u : A}{\Gamma \vdash \texttt{try}\ t\ \texttt{with}\ u : A}$$

cannot only be typed up to an encoding such as

$$\mathrm{trywith} : (1 \to A) \to (1 \to A) \to A$$

where the $1 \to A$ types indicates that there may be effects performed by the invocation of the argument, resulting in the equivalence below.

$$\texttt{try}\ t\ \texttt{with}\ u \sim \mathrm{trywith}\ (\lambda().\, t)\ (\lambda().\, u)$$

Here, we are in call-by-name, and thence there is no way to ensure the purity of our objects without more expressive polarization. We could resort to the continuation-passing style technique called storage operators by Krivine, but that would be heavyweight.

When trying to define the connective ${\sim}(-)$, similar problems arise. Indeed, the one equation we want on it is the property that

$${\sim}(A \to B) \equiv A \times ({\sim} B)$$

but we do not have products in the source language either, if we are only considering the negative fragment. If we want to retrieve pairs via the impredicative encoding, we will once again hit the fact that we cannot discriminate between pure and impure arrows. There is a simple (simplistic?) way to ensure that stacks are pure objects, which consists in breaking the symmetry of the type interpretation of stacks. Namely, we can pose

$$\begin{array}{rcl} \mathbb{W}({\sim} A) & := & \mathbb{C}(A) \\ \mathbb{C}({\sim} A) & := & 0 \end{array}$$

so that arrows manipulating stacks are interpreted as

$$\begin{array}{rcl} \mathbb{W}({\sim} A \to B) & \cong & \mathbb{C}(A) \to \mathbb{W}(B) \\ \mathbb{C}({\sim} A \to B) & \cong & \mathbb{C}(A) \times \mathbb{C}(B) \\ \mathbb{W}(A \to {\sim} B) & \cong & \mathbb{W}(A) \to \mathbb{C}(B) \\ \mathbb{C}(A \to {\sim} B) & \cong & 0 \end{array}$$

for any A and B. This simple trick forbids one to observe the accesses of variables by stacks by a mere typing argument.

We will not provide the corresponding constructions though, because that would require to solve the multiset issue, and as we will show in the section studying the implementation of Markov's, we need to reinstall orthogonality to make it work properly.

10.2.2 Markov's principle

There are several issues to consider in order to recover Markov's principle. For us to explain the details, we first need to agree on what we mean as Markov's principle in a setting where we do not have existential types at hand. We will stick to the propositional version of Markov's principle described in [55], that is, we are interested in the principle

$$\neg\neg P \to P$$

where P is a purely positive type, i.e. built over the following inductive grammar:

$$P, Q := 0 \mid 1 \mid P + Q \mid P \times Q$$

As explained in the above article, this is a generalization of Markov's principle. Indeed, existential statements also pertain to the purely positive types, insofar as they can be eliminated by pattern-matching, allowing to purge them from the effects they may contain.

To understand where things start to get involved in our translation, let us just start with the explicitation of the interpretation of the negation.

Proposition 73. *Let A be a type, then*

$$\begin{array}{rcl} \mathbb{W}(\llbracket \neg A \rrbracket_{\mathrm{n}}) & \cong & \left\{ \begin{array}{c} \mathbb{W}(\llbracket A \rrbracket_{\mathrm{n}}) \to 0 \\ \times \\ \mathbb{W}(\llbracket A \rrbracket_{\mathrm{n}}) \to \mathfrak{M}\, \mathbb{C}(\llbracket A \rrbracket_{\mathrm{n}}) \end{array} \right. \\ \mathbb{C}(\llbracket \neg A \rrbracket_{\mathrm{n}}) & \cong & \mathbb{W}(\llbracket A \rrbracket_{\mathrm{n}}) \end{array}$$

In the sequence-based Dialectica, the first component of the witness type was collapsed to the empty sequence, because the interpretation of 0 was empty. Here, we run into trouble as soon as we want to recover information from a negated type, because pushing the isomorphism further, we have indeed

$$\mathbb{W}(\llbracket \neg A \rrbracket_{\mathrm{n}}) \quad \cong \quad \mathbb{W}(\llbracket A \rrbracket_{\mathrm{n}}) \to 0$$

which is not likely to be ever fed with something if our target system is consistent. This is why we need to emulate what the historical Dialectica does through a clever workaround.

The main idea is that actually, in call-by-name, we do not really use the empty type to encode negation, but rather the *bottom* type representing the return type of the whole computation. A bit of linear logic will help, as it sharply contrasts the two following types:

$$\begin{array}{rcl} !A \multimap 0 & & \\ !A \multimap \bot & \cong & ?A^{\bot} \end{array}$$

Contrarily to the empty type 0, the bottom type $\bot$ does not represent the impossibility of something, but rather the mere fact that the computation handed back control to the caller. While it shares with 0 the property that it cannot be proved in an empty sequent, it otherwise does not allow one to derive anything from it. That is the key point of our trick.

We do need to have a way to talk about this type in the source language, so we simply add it in the grammar of intuitionistic types, as follows

$$A := \ldots \mid \bot$$

and we trivially interpret it through the linear decomposition as itself, i.e.

$$[\![\bot]\!]_{\mathrm{n}} := \bot$$

As the bottom type has no introduction rule in linear logic, we mirror this fact by *not* adding any rule for it in the intuitionistic calculus. It somehow acts as an existentially-quantified type whose actual content is unknown. The negation over this type (thereafter called *weak negation*) behaves much better now.

Proposition 74. *Let A be a type, then*

$$\begin{aligned} \mathbb{W}([\![A \to \bot]\!]_{\mathrm{n}}) &\cong \mathbb{W}([\![A]\!]_{\mathrm{n}}) \to \mathfrak{M}\,\mathbb{C}([\![A]\!]_{\mathrm{n}}) \\ \mathbb{C}([\![A \to \bot]\!]_{\mathrm{n}}) &\cong \mathbb{W}([\![A]\!]_{\mathrm{n}}) \end{aligned}$$

In particular, we can effectively recover information from a witness of a weak negation, without being in an inconsistent state. Let us look in particular at the translation of the double weak negation of a type, to guess what we can extract of it.

Proposition 75. *Let A be a type, then*

$$\begin{aligned} \mathbb{W}([\![(A \to \bot) \to \bot]\!]_{\mathrm{n}}) &\cong (\mathbb{W}([\![A]\!]_{\mathrm{n}}) \to \mathfrak{M}\,\mathbb{C}([\![A]\!]_{\mathrm{n}})) \to \mathfrak{M}\,\mathbb{W}([\![A]\!]_{\mathrm{n}}) \\ \mathbb{C}([\![(A \to \bot) \to \bot]\!]_{\mathrm{n}}) &\cong \mathbb{W}([\![A]\!]_{\mathrm{n}}) \to \mathfrak{M}\,\mathbb{C}([\![A]\!]_{\mathrm{n}}) \end{aligned}$$

We stumble on a new issue here. It is rather obvious that, in order to mimic what the historical Dialectica did, we need to recover a witness from the right-hand side of the witness arrow, which now returns a multiset instead of a plain type. And this is quite an issue, because this multiset can be empty, which would defeat the purpose of extracting anything from it.

To be able to recover anything at all from it, we need to reintroduce the orthogonality relation we nonchalantly threw away in Section 9.1.4. We thus assume that we live in a dependently-typed system similar to the one described at Section 8.4 from the remaining of this section, so that we can describe properties of our terms. The orthogonality is then defined just as in the type-theoretical Dialectica, except for the exponential modality which has to be adapted to the multiset presentation.

Definition 119 (Multiset quantification)**.** We assume that our dependently typed system features a connective allowing to quantify a proposition over an abstract multiset, i.e. the syntax of propositions from Section 8 is extended with

$$\mathbf{A}, \mathbf{B} := \ldots \mid \forall x \in t.\, \mathbf{A}$$

while the well-formedness relation is extended with the rule

$$\frac{\Sigma \vdash t : \mathfrak{M}\, A \qquad \Sigma, x : A \vdash_{\mathrm{wf}} \mathbf{C}}{\Sigma \vdash_{\mathrm{wf}} \forall x \in t.\, \mathbf{C}}$$

Furthermore, we assume that we have the following axioms.

- $(\forall x \in \emptyset.\, \mathbf{P}) \leftrightarrow \top$
- $(\forall x \in \{t\}.\, \mathbf{P}) \leftrightarrow \mathbf{P}[x := t]$
- $(\forall x \in t \odot u.\, \mathbf{P}) \leftrightarrow (\forall x \in t.\, \mathbf{P}) \wedge (\forall x \in u.\, \mathbf{P})$
- $(\forall y \in t \gg\!= f.\, \mathbf{Q}) \to (\forall x.\, (\forall y \in f\ x.\, \mathbf{Q}) \to \mathbf{P}) \to \forall x \in t.\, \mathbf{P}$

Note that the last axiom is not really expressible in the system used at Section 8 because it features a universal quantification, but this can be worked around by presenting it as a derivation rule rather than as an axiom, which we give below.

$$\frac{\begin{array}{l}\Sigma \vdash t : \mathfrak{M}\, A \\ \Sigma \vdash f : A \to \mathfrak{M}\, B\end{array} \qquad \Sigma \mid \Gamma \vdash \forall y \in t \gg\!= f.\, \mathbf{Q} \qquad \Sigma, x : A \mid \Gamma, \forall y \in f\ x.\, \mathbf{Q} \vdash \mathbf{P}}{\Sigma \mid \Gamma \vdash \forall x \in t.\, \mathbf{P}}$$

Now we can simply write the definition of orthogonality on linear types.

Definition 120 (Revised orthogonality)**.** For any linear type A, we define the orthogonality relation $A_D[-,-]$ as in section 8.4 except for the bang connective, where we set

$$(!A)_D[u, \varphi] := \forall \pi \in \varphi\ u.\, A_D[u, \pi]$$

One can show that with this notion of orthogonality, our multiset-based translation provides universal realizers.

Theorem 27. *If $\vdash t : A$, then we have a proof of*

$$\pi : \mathbb{C}(A) \mid \cdot \vdash A_D[t^\bullet, \pi]$$

We will not show this theorem here. It is actually a simple variant of the Diller-Nahm translation [40], where we abstracted away the fact we were working with multisets (or, more precisely, finite sets). Its proof is a simple induction over the typing derivation, generalized to cope with the free variables under the following form.

$$\vec{x} : \Gamma \vdash t : A \quad \text{implies} \quad \vec{x} : \mathbb{W}(\Gamma), \pi : \mathbb{C}(A) \mid \forall \rho_i \in t_{x_i}\ \pi.\, \Gamma_{i\,D}[x_i, \rho_i] \vdash A_D[t^\bullet, \pi].$$

What we will rather show is the interpretation of the orthogonality over the weak negation of a type.

Proposition 76. *By taking the isomorphisms of Proposition 74 to be definitions, we have the following unfolding.*

$$(A \to \bot)_D[\varphi, u] := \neg(\forall \pi \in \varphi\ u.\, A_D[u, \pi])$$

Assuming our abstract multisets are in practice implemented with *finite* multisets, we *know* by a meta-theoretical argument that one can extract from a proof of $\neg(\forall x \in t.\, \mathbf{P})$ a witness w that satisfies $\mathbf{P}[x := w]$ when $\mathbf{P}$ is decidable. This is simply done by iterating over the multiset t, checking for each element whether it satisfies $\mathbf{P}$. This search eventually terminates because t is finite by definition.

This mechanism is not observable in the historical presentation of the Dialectica translation, because instead of returning the whole multiset of observed stacks, the translation provides us with a term that has been already picked up from the multiset by translation itself, using the dynamic dispatch merge function. The historical interpretation of Markov's principle as the mere identity is therefore hiding conceptual complexity under the carpet.

It is also noteworthy to realize that, in this case, the orthogonality relation remains decidable on closed formulae[2]. Indeed, one can effectively check if all elements of a multiset are orthogonal to a given term or not. This is why the orthogonality interpretation of the weak negation is actually productive.

Therefore, from a realizer φ of $(A \to \bot) \to \bot$, where A is a closed formula, one can build a function

$$\varphi^{\mathrm{w}} : (\mathbb{W}(\llbracket A \rrbracket_{\mathrm{n}}) \to \mathfrak{M}\,\mathbb{C}(\llbracket A \rrbracket_{\mathrm{n}})) \to \mathbb{W}(\llbracket A \rrbracket_{\mathrm{n}})$$

satisfying the following specification

$$k : \mathbb{W}(\llbracket A \rrbracket_{\mathrm{n}}) \to \mathfrak{M}\,\mathbb{C}(\llbracket A \rrbracket_{\mathrm{n}}) \mid \cdot \vdash \neg(A \to \bot)_D[k, \varphi^{\mathrm{w}}\ k]$$

which is, by unfolding, equivalent to

$$k : \mathbb{W}(\llbracket A \rrbracket_{\mathrm{n}}) \to \mathfrak{M}\,\mathbb{C}(\llbracket A \rrbracket_{\mathrm{n}}) \mid \cdot \vdash \neg\neg\forall \pi \in k\ (\varphi^{\mathrm{w}}\ k).\, A_D[\varphi^{\mathrm{w}}, \pi]$$

One can conclude by applying the decidability of the orthogonality relation to prove that the extracted witness realizes the formula indeed.

10.2.3 Independence of premise

As for the interpretation of Markov's principle, we do not have first-order types in our calculus, so we need to describe an equivalent principle in a propositional setting. This is once again done relatively to positive types. From the abstract point of view, independence of premise is a dependent form of the following principle:

$$(A \to B \times C) \to B \times (A \to C)$$

[2] All our types are propositional. This would not hold if ever we had quantifiers in our source language.

where B cannot computationally depend on A. It can be seen as a special instance of a commutation of a positive connective (here, the conjunction) with a negative one (the arrow).

If we want to port this axiom from the historical presentation, there is already an obvious issue: we cannot define that a type is irrelevant when its interpretation is the empty sequence, because we do not have sequences anymore. The closest thing one can do to overcome this issue is to define it relatively to the singleton type.

Definition 121 (Revised irrelevance)**.** Given a type A, we say it is irrelevant whenever

$$\begin{array}{rcl} \mathbb{W}(A) & \cong & 1 \\ \mathbb{C}(A) & \cong & 1 \end{array}$$

Proposition 77. *Unluckily, almost all types are relevant in our presentation.*

- *The type* 1 *is irrelevant.*
- *The type* 0 *is relevant.*
- *Even if* A *and* B *are irrelevant, in general* $A \to B$ *and* $A \times B$ *are not irrelevant.*

This comes from the fact that our translation features multisets, and it is obvious that multisets of singleton types are relevant, because they are isomorphic to integers. The independence of premise loses therefore all of its usefulness, because it states no more than

$$(1 \to B \times C) \to B \times (1 \to C)$$

which is trivially true, and even an isomorphism.

Apparently, the fact that the original Dialectica interprets the axiom of independence of premise is more due to chance rather than to a deliberate design choice. Indeed, it essentially comes from the fact that a lot of type interpretations are collapsed into the empty sequence, and that the realizability emerging from them comes uniquely from the meta-level logic and not from their trivial realizers. Our proof-theoretical presentation is much more computationally aware, and many properties that came from the meta-level are now present in the types themselves. Therefore, it seems that the independence of premise is out of our reach.

Yet, a similar principle could be probably retrieved by adding a touch of Friedman translation to the multiset-using Dialectica translation. The interpretation of the elimination of falsity in the historical translation, which is the only rule that directly relies on the dummy value at the level of terms, is typical of a Friedman-like translation. The historical Dialectica translation does not make any difference between dummy terms that serve an empty multiset purpose and dummy terms that play a call-by-name exception rôle. The independence of premise actually uses the latter, not the former. Such a translation remains to be described formally, though.

10.3 Classical-by-name translation

In this section we are going to take a glance at the classical version of the call-by-name linear decomposition. Actually, the classical-by-name version of the linear decomposition is very close to its intuitionistic by-name variant, which allows a rather direct adaptation of what we have done up to now. As a quick side remark, the relationship between call-by-value and classical-by-value is much more involved.

It is well-known that the vanilla KAM is already well-suited to handle classical logic natively through the `callcc` operator, so it may be interesting to see what our Dialectica does in this setting. For the sake of atomicity, we will be translating $\lambda\mu$-calculus instead of a λ-calculus enriched with a `callcc` primitive.

10.3.1 The $\lambda\mu$-calculus

The $\lambda\mu$-calculus is an extension of λ-calculus designed by Parigot [94] as the programming language equivalent to the classical natural deduction proof system, thanks to its handling of continuation manipulation. We already sketched it quickly in Chapter 6, but we will present it more thoroughly in this section as we will rely on more involved features, including typing.

Definition 122 ($\lambda\mu$-terms)**.** The $\lambda\mu$-terms are the usual λ-terms extended with a new μ binder (hence the name) together with a new syntactic class c known as commands. The whole grammar is given below.

$$t, u := x \mid t\ u \mid \lambda x.\, t \mid x \mid \mu\alpha.\, c$$

$$c := [\alpha]\, t$$

The α variables introduced by μ binders are in a distinct class than the usual variables introduced by λ. We use Greek names to distinguish them.

Remark 16. The use of Greek names is coherent with our own use of Greek-lettered variables in the Dialectica translation, because in both cases, they stand for stacks. In particular, we will be naming them consistently *stack variables*.

Definition 123 ($\lambda\mu$-calculus reduction)**.** The $\lambda\mu$-calculus reductions in call-by-name are generated by the three rules below, together with context closure.

$$\begin{array}{lcl} (\lambda x.\, t)\ u & \to & t[x := u] \\ (\mu\alpha.\, t)\ u & \to & \mu\alpha.\, t[\alpha := u \cdot \alpha] \\ [\beta]\, \mu\alpha.\, c & \to & c[\alpha := \beta] \end{array}$$

The notation $t[\alpha := u \cdot \alpha]$ means that we push the u term as an argument of all terms t in commands $[\alpha]\, t$. More formally, we define it by induction on t below, with the usual freshness conditions on bound variables.

- $x[\alpha := r \cdot \alpha] := x$

- $(\lambda x.\, t)[\alpha := r \cdot \alpha] := \lambda x.\, t[\alpha := r \cdot \alpha]$
- $(t\ u)[\alpha := r \cdot \alpha] := t[\alpha := r \cdot \alpha]\ u[\alpha := r \cdot \alpha]$
- $(\mu\beta.\, c)[\alpha := r \cdot \alpha] := \mu\beta.\, c[\alpha := r \cdot \alpha]$
- $([\alpha]\, t)[\alpha := r \cdot \alpha] := [\alpha]\, t[\alpha := r \cdot \alpha]\ r$
- $([\beta]\, t)[\alpha := r \cdot \alpha] := [\beta]\, t[\alpha := r \cdot \alpha]$

This strange-looking reduction is actually better understood when thinking of it as substitution of stacks.

Definition 124 ($\lambda\mu$-calculus typing)**.** Types of the $\lambda\mu$-calculus are the usual simply-typed types with arrow as the only connective, recalled below.

$$A, B := \alpha \mid A \to B$$

Typing statements of the $\lambda\mu$-terms are of the form

$$\Gamma \vdash t : A \mid \Delta$$

where both Γ and Δ are lists of types. We also type commands c as

$$\Gamma \vdash c \mid \Delta$$

with the same syntactic classes. The derivation rules are given below.

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A \mid \Delta} \qquad \frac{\Gamma, x : A \vdash t : B \mid \Delta}{\Gamma \vdash \lambda x.\, t : A \to B \mid \Delta} \qquad \frac{\Gamma \vdash t : A \to B \mid \Delta \qquad \Gamma \vdash u : A \mid \Delta}{\Gamma \vdash t\ u : B \mid \Delta}$$

$$\frac{\Gamma \vdash t : A \mid \Delta \qquad \alpha : A \in \Delta}{\Gamma \vdash [\alpha]\, t \mid \Delta} \qquad \frac{\Gamma \vdash c \mid \Delta, \alpha : A}{\Gamma \vdash \mu\alpha.\, c : A \mid \Delta}$$

10.3.2 Classical KAM

The usual KAM is almost $\lambda\mu$-calculus ready. The one thing added by the $\lambda\mu$-calculus is actually the ability to manipulate stacks as first-class objects, akin to the introduction of stacks of the Dialectica translation.

Essentially, the μ binder captures stacks, while commands reinstate them. Because we have true stack variables, the definitions of the KAM have to be extended a little bit, and in particular enriching the stacks with variables.

Definition 125 (Classical KAM processes)**.** As in the previous case, KAM states are made of processes p. Yet, those processes are now constituted of pair of a closure c and an evaluable stack π. Those stacks may contain variables, which is the main difference. This also implies that environments σ may bind stacks. We give the inductive grammar of those objects below.

$$\begin{array}{rcl} p & := & \langle c \mid \pi \rangle \\ c & := & (t, \sigma) \\ t, u & := & \cdots \\ \pi, \rho & := & (\alpha, \sigma) \mid c \cdot \pi \\ \sigma, \tau & := & \cdot \mid \sigma + (x := t) \mid \sigma + (\alpha := \pi) \end{array}$$

Here the t, u syntactic class stands for $\lambda\mu$-terms.

There is no distinguished ε in this presentation, because it is a particular case of an unbound stack variable. When needed, we will simply assume that this is some fresh free variable with an empty environment.

The reduction rules adapted in a similar fashion. There is a critical pair, though, which arises from the choice we have to make about the precise moment at which we evaluate stack variables. In absence of additional side-effects (such as global state), such a choice is a mere matter of taste, because the difference is not observable. In our case, we stick to unfolding stack variables on a demand-driven basis, for it makes the simulation proof look more natural.

Definition 126 (Classical KAM rules)**.** The reduction rules are given below.

$$\begin{array}{rcl} \langle (x, \sigma + (x := c)) \mid \pi \rangle & \longrightarrow & \langle c \mid \pi \rangle \\ \langle (x, \sigma + (y := c)) \mid \pi \rangle & \longrightarrow & \langle (x, \sigma) \mid \pi \rangle \\ \langle (x, \sigma + (\alpha := \pi)) \mid \pi \rangle & \longrightarrow & \langle (x, \sigma) \mid \pi \rangle \\ \langle (t\ u, \sigma) \mid \pi \rangle & \longrightarrow & \langle (t, \sigma) \mid (u, \sigma) \cdot \pi \rangle \\ \langle (\lambda x.\, t, \sigma) \mid c \cdot \pi \rangle & \longrightarrow & \langle (t, \sigma + (x := c)) \mid \pi \rangle \\ \langle (\lambda x.\, t, \sigma) \mid (\alpha, \tau + (\alpha := \pi)) \rangle & \longrightarrow & \langle (\lambda x.\, t, \sigma) \mid \pi \rangle \\ \langle (\lambda x.\, t, \sigma) \mid (\alpha, \tau + (\beta := \pi)) \rangle & \longrightarrow & \langle (\lambda x.\, t, \sigma) \mid (\alpha, \tau) \rangle \\ \langle (\lambda x.\, t, \sigma) \mid (\alpha, \tau + (x := c)) \rangle & \longrightarrow & \langle (\lambda x.\, t, \sigma) \mid (\alpha, \tau) \rangle \\ \langle (\mu\alpha.\, [\beta]\, t, \sigma) \mid \pi \rangle & \longrightarrow & \langle (t, \sigma + (\alpha := \pi)) \mid (\beta, \sigma + (\alpha := \pi)) \rangle \end{array}$$

Note that most of the rules are about variable accesses, and could be summarised into only one reduction rule. We keep the expanded form for its simplicity.

10.3.3 Type translation

We describe here the type translation that arises from the classical-by-name linear decomposition through the Dialectica translation. It is very similar to the call-by-name translation.

Definition 127 (Classical-by-name linear decomposition)**.** The classical-by-name linear decomposition $[\![A]\!]_{\mathcal{C}\mathrm{n}}$ of an intuitionistic type A is inductively defined on A as follows.

$$\begin{array}{rcl} [\![\alpha]\!]_{\mathcal{C}\mathrm{n}} & := & \alpha \\ [\![A \to B]\!]_{\mathcal{C}\mathrm{n}} & := & !?[\![A]\!]_{\mathcal{C}\mathrm{n}} \multimap ?[\![B]\!]_{\mathcal{C}\mathrm{n}} \end{array}$$

Sequents of the $\lambda\mu$-calculus are translated as

$$\begin{aligned}\llbracket \Gamma \vdash A \mid \Delta \rrbracket_{\mathcal{C}\mathrm{n}} &:= !?\llbracket \Gamma \rrbracket_{\mathcal{C}\mathrm{n}} \vdash ?\llbracket A \rrbracket_{\mathcal{C}\mathrm{n}} \mid ?\llbracket \Delta \rrbracket_{\mathcal{C}\mathrm{n}} \\ \llbracket \Gamma \vdash \Delta \rrbracket_{\mathcal{C}\mathrm{n}} &:= !?\llbracket \Gamma \rrbracket_{\mathcal{C}\mathrm{n}} \vdash ?\llbracket \Delta \rrbracket_{\mathcal{C}\mathrm{n}}\end{aligned}$$

For the sake of readability, as we did before, we define the composition of $\llbracket - \rrbracket_{\mathcal{C}\mathrm{n}}$ with $\mathbb{W}(-)$ up to isomorphism, and just write $\mathbb{W}(-)$ for the composition when the context is clear.

Proposition 78 (Arrow translation)**.** *We take the following isomorphism to be a definition.*

$$\mathbb{W}(\llbracket A \to B \rrbracket_{\mathcal{C}\mathrm{n}}) \cong \begin{cases} (\mathbb{C}(\llbracket A \rrbracket_{\mathcal{C}\mathrm{n}}) \to \mathfrak{M}\,\mathbb{W}(\llbracket A \rrbracket_{\mathcal{C}\mathrm{n}})) \to \mathbb{C}(\llbracket B \rrbracket_{\mathcal{C}\mathrm{n}}) \to \mathfrak{M}\,\mathbb{W}(\llbracket B \rrbracket_{\mathcal{C}\mathrm{n}}) \\ \times \\ (\mathbb{C}(\llbracket A \rrbracket_{\mathcal{C}\mathrm{n}}) \to \mathfrak{M}\,\mathbb{W}(\llbracket A \rrbracket_{\mathcal{C}\mathrm{n}})) \to \mathbb{C}(\llbracket B \rrbracket_{\mathcal{C}\mathrm{n}}) \to \mathfrak{M}\,\mathbb{C}(\llbracket A \rrbracket_{\mathcal{C}\mathrm{n}}) \end{cases}$$

$$\mathbb{C}(\llbracket A \to B \rrbracket_{\mathcal{C}\mathrm{n}}) \cong (\mathbb{C}(\llbracket A \rrbracket_{\mathcal{C}\mathrm{n}}) \to \mathfrak{M}\,\mathbb{W}(\llbracket A \rrbracket_{\mathcal{C}\mathrm{n}})) \times \mathbb{C}(\llbracket B \rrbracket_{\mathcal{C}\mathrm{n}})$$

We now turn to look at this translation through the prism of Dialectica. As usual, we will rely on a handy isomorphism for sequents, allowing to easily manipulate free variables. Note that we now have a new type of free variables coming from Δ. Our translation should therefore take it into account.

Proposition 79 (Sequent isomorphism)**.** *We have the following isomorphisms, when* $\Gamma := \Gamma_1, \ldots, \Gamma_n$ *and* $\Delta := \Delta_1, \ldots, \Delta_m$.

$$\mathbb{W}(\llbracket \Gamma \vdash A \mid \Delta \rrbracket_{\mathcal{C}\mathrm{n}}) \cong \begin{cases} \mathbb{W}(?\llbracket \Gamma \rrbracket_{\mathcal{C}\mathrm{n}}) \to \mathbb{C}(\llbracket \Delta \rrbracket_{\mathcal{C}\mathrm{n}}) \to \mathbb{C}(\llbracket A \rrbracket_{\mathcal{C}\mathrm{n}}) \to \mathfrak{M}\,\mathbb{W}(\llbracket A \rrbracket_{\mathcal{C}\mathrm{n}}) \\ \times \\ \mathbb{W}(?\llbracket \Gamma \rrbracket_{\mathcal{C}\mathrm{n}}) \to \mathbb{C}(\llbracket \Delta \rrbracket_{\mathcal{C}\mathrm{n}}) \to \mathbb{C}(\llbracket A \rrbracket_{\mathcal{C}\mathrm{n}}) \to \mathfrak{M}\,\mathbb{C}(\llbracket \Gamma_1 \rrbracket_{\mathcal{C}\mathrm{n}}) \\ \times \\ \cdots \\ \times \\ \mathbb{W}(?\llbracket \Gamma \rrbracket_{\mathcal{C}\mathrm{n}}) \to \mathbb{C}(\llbracket \Delta \rrbracket_{\mathcal{C}\mathrm{n}}) \to \mathbb{C}(\llbracket A \rrbracket_{\mathcal{C}\mathrm{n}}) \to \mathfrak{M}\,\mathbb{C}(\llbracket \Gamma_n \rrbracket_{\mathcal{C}\mathrm{n}}) \\ \times \\ \mathbb{W}(?\llbracket \Gamma \rrbracket_{\mathcal{C}\mathrm{n}}) \to \mathbb{C}(\llbracket \Delta \rrbracket_{\mathcal{C}\mathrm{n}}) \to \mathbb{C}(\llbracket A \rrbracket_{\mathcal{C}\mathrm{n}}) \to \mathfrak{M}\,\mathbb{W}(\llbracket \Delta_1 \rrbracket_{\mathcal{C}\mathrm{n}}) \\ \times \\ \cdots \\ \times \\ \mathbb{W}(?\llbracket \Gamma \rrbracket_{\mathcal{C}\mathrm{n}}) \to \mathbb{C}(\llbracket \Delta \rrbracket_{\mathcal{C}\mathrm{n}}) \to \mathbb{C}(\llbracket A \rrbracket_{\mathcal{C}\mathrm{n}}) \to \mathfrak{M}\,\mathbb{W}(\llbracket \Delta_m \rrbracket_{\mathcal{C}\mathrm{n}}) \end{cases}$$

$$\mathbb{W}(\llbracket \Gamma \vdash \Delta \rrbracket_{\mathcal{C}\mathrm{n}}) \cong \begin{cases} \mathbb{W}(?\llbracket \Gamma \rrbracket_{\mathcal{C}\mathrm{n}}) \to \mathbb{C}(\llbracket \Delta \rrbracket_{\mathcal{C}\mathrm{n}}) \to \mathbb{C}(\llbracket A \rrbracket_{\mathcal{C}\mathrm{n}}) \to \mathfrak{M}\,\mathbb{C}(\llbracket \Gamma_1 \rrbracket_{\mathcal{C}\mathrm{n}}) \\ \times \\ \dots \\ \times \\ \mathbb{W}(?\llbracket \Gamma \rrbracket_{\mathcal{C}\mathrm{n}}) \to \mathbb{C}(\llbracket \Delta \rrbracket_{\mathcal{C}\mathrm{n}}) \to \mathbb{C}(\llbracket A \rrbracket_{\mathcal{C}\mathrm{n}}) \to \mathfrak{M}\,\mathbb{C}(\llbracket \Gamma_n \rrbracket_{\mathcal{C}\mathrm{n}}) \\ \times \\ \mathbb{W}(?\llbracket \Gamma \rrbracket_{\mathcal{C}\mathrm{n}}) \to \mathbb{C}(\llbracket \Delta \rrbracket_{\mathcal{C}\mathrm{n}}) \to \mathbb{C}(\llbracket A \rrbracket_{\mathcal{C}\mathrm{n}}) \to \mathfrak{M}\,\mathbb{W}(\llbracket \Delta_1 \rrbracket_{\mathcal{C}\mathrm{n}}) \\ \times \\ \dots \\ \times \\ \mathbb{W}(?\llbracket \Gamma \rrbracket_{\mathcal{C}\mathrm{n}}) \to \mathbb{C}(\llbracket \Delta \rrbracket_{\mathcal{C}\mathrm{n}}) \to \mathbb{C}(\llbracket A \rrbracket_{\mathcal{C}\mathrm{n}}) \to \mathfrak{M}\,\mathbb{W}(\llbracket \Delta_m \rrbracket_{\mathcal{C}\mathrm{n}}) \end{cases}$$

Proof. By induction on Γ and Δ. The remainder of the argument is the same as the intuitionistic case. □

Remark 17. Note that we use the ? notation for compactness, but it should be unfolded into the following.

$$\mathbb{W}(?A) := \mathbb{C}(A) \to \mathfrak{M}\,\mathbb{W}(A)$$

This isomorphism is coherent with the principle of management of free variables we exposed above. Indeed, the $\mathbb{W}(?\llbracket \Gamma \rrbracket_{\mathcal{C}\mathrm{n}})$ corresponds to the term variables found in Γ, while the $\mathbb{C}(\llbracket \Delta \rrbracket_{\mathcal{C}\mathrm{n}})$ corresponds to the stack variables found in Δ. In addition, we have the right number of produced objects: one for the term (the first component), one for each variable of Γ (the n following components) and one for each variable in Δ (the m remaining components). Hence the formal definition below.

Definition 128 (Sequent translation)**.** In the classical-by-name Dialectica, sequents of the form $\Gamma \vdash t : A \mid \Delta$ are translated into three types of objects.

- The direct translation $t^\bullet$:

$$\mathbb{W}(?\Gamma), \mathbb{C}(\Delta) \vdash t^\bullet : \mathbb{C}(A) \to \mathfrak{M}\,\mathbb{W}(A)$$

- For each variable $x : \Gamma_i \in \Gamma$ the reverse translation t_x:

$$\mathbb{W}(?\Gamma), \mathbb{C}(\Delta) \vdash t_x : \mathbb{C}(A) \to \mathfrak{M}\,\mathbb{C}(\Gamma_i)$$

- For each variable $\alpha : \Delta_j \in \Delta$, the stack reverse translation t_α:

$$\mathbb{W}(?\Gamma), \mathbb{C}(\Delta) \vdash t_\alpha : \mathbb{C}(A) \to \mathfrak{M}\,\mathbb{W}(\Delta_j)$$

Likewise, commands $\Gamma \vdash c \mid \Delta$ are just translated into two types of objects:

- For each variable $x : \Gamma_i \in \Gamma$ the reverse translation c_x:

$$\mathbb{W}(?\Gamma), \mathbb{C}(\Delta) \vdash c_x : \mathfrak{M}\,\mathbb{C}(\Gamma_i)$$

- For each variable $\alpha : \Delta_j \in \Delta$, the stack reverse translation c_α:

$$\mathbb{W}(?\Gamma), \mathbb{C}(\Delta) \vdash c_\alpha : \mathfrak{M}\,\mathbb{W}(\Delta_j)$$

Note that in both cases, we consider that Greek-lettered variables from the Δ context can be univocally considered as usual variables of the target calculus. We were already doing so in the call-by-name translation, so this should not be surprising to the reader by now.

It is rather interesting to witness that, in a classical formulation, terms are much more symmetric than their intuitionistic counterpart. Indeed, the forward translation is now essentially the same as the reverse translation, except that it produces witnesses instead of counters. Likewise, the stack reverse translation does not even deserve this nickname, because it has actually the same type as the direct translation, so it could be advantageously renamed the α-direct translation, or something similar. We nonetheless stick to the old naming scheme for the sake of uniformity.

In all cases, the terms are waiting for a stack of the type of the term being constructed. Together with the distinctive ? worn by the intuitionistic context, this is the only asymmetry of the translation.

10.3.4 Term translation

To write the term translation down, we use the intuition provided by the original simulation theorem, and extend it to the classical case. Namely, the multiset produced by the term t_α is made of the terms that were encountered by the variable α in head position during the reduction of the KAM. As $t^\bullet$ is no more than a particular case of t_α, for the distinguished unnamed current stack, its translation is built by symmetry.

Definition 129 (Term translation)**.** The term translation is mutually inductively defined below.

Direct translation:

$$\begin{aligned}
x^\bullet &:= \lambda\pi.\, x\ \pi \\
(\lambda x.\, t)^\bullet &:= \lambda_.\, \{(\lambda x\, \pi.\, t^\bullet\ \pi, \lambda x\, \pi.\, t_x\ \pi)\} \\
(t\ u)^\bullet &:= \lambda\pi.\, t^\bullet\ (u^\bullet, \pi) \gg\!= \lambda(f, _).\, f\ u^\bullet\ \pi \\
(\mu\alpha.\, c)^\bullet &:= \lambda\alpha.\, c_\alpha
\end{aligned}$$

Reverse translation:

$$
\begin{array}{lcl}
x_x & := & \lambda\pi.\,\{\pi\} \\
x_y & := & \lambda\pi.\,\emptyset \\
(\lambda y.\,t)_x & := & \lambda(y,\pi).\,t_x\ \pi \\
(t\ u)_x & := & \lambda\pi.\,(t^\bullet\ (u^\bullet,\pi) \gg\!= \lambda(\_,f).\,f\ u^\bullet\ \pi \gg\!= \lambda\rho.\,u_x\ \rho) \odot (t_x\ (u^\bullet,\pi)) \\
(\mu\alpha.\,c)_x & := & \lambda\alpha.\,c_x \\
([\alpha]\,t)_x & := & t_x\ \alpha
\end{array}
$$

Stack reverse translation:

$$
\begin{array}{lcl}
x_\alpha & := & \lambda\pi.\,\emptyset \\
(\lambda y.\,t)_\alpha & := & \lambda(y,\pi).\,t_\alpha\ \pi \\
(t\ u)_\alpha & := & \lambda\pi.\,(t^\bullet\ (u^\bullet,\pi) \gg\!= \lambda(\_,f).\,f\ u^\bullet\ \pi \gg\!= \lambda\rho.\,u_\alpha\ \rho) \odot (t_\alpha\ (u^\bullet,\pi)) \\
(\mu\beta.\,c)_\alpha & := & \lambda\beta.\,c_\alpha \\
([\alpha]\,t)_\alpha & := & (t^\bullet\ \alpha) \odot (t_\alpha\ \alpha) \\
([\beta]\,t)_\alpha & := & t_\alpha\ \beta
\end{array}
$$

As we can see, there is a little bit of redundancy: we duplicate some arguments in the application case. This is actually a defect of the type decomposition of the Dialectica translation. We will provide a better-behaved decomposition later on.

The symmetry between $(-)_x$ and $(-)_\alpha$ is fairly obvious in this presentation. This is not unexpected, as usual variables and stack variables have a dual rôle in the calculus. The only places where the two translations are distinct are precisely the variable-introducing cases, be it the simple variable case for intuitionistic variable or the command case for stack variables.

Proposition 80 (Typing soundness)**.** *As expected, the translation preserves the typing according to the translation of definition 129.*

Proof. By induction on the typing derivation, as usual. □

10.3.5 Computational soundness

The call-by-name translation and the classical-by-name translation share many common points, including in the type translation as well as in the term translation. This is therefore no surprise that most of the theorems about the computational content of the latter are very similar to the ones about the former. We give in this section the corresponding modified properties.

Proposition 81 (Emptiness lemma)**.** *If x is not free in t (resp. in c) then $t_x \equiv_\beta \lambda\pi.\,\emptyset$ (resp. $c_x \equiv_\beta \emptyset$). Likewise, if α is not free in t (resp. in c) then $t_\alpha \equiv_\beta \lambda\pi.\,\emptyset$ (resp. $c_\alpha \equiv_\beta \emptyset$).*

Proof. For the reverse translation, this is essentially the same as the intuitionistic case. For the stack reverse translation, one only has to observe that we make the t_α and c_α translations grow only when we encounter as a subterm a command of the form $[\alpha]\, u$. Because we assumed that α was not free in the considered term, this cannot happen. □

The original substitution holds, extended to the new translations.

Proposition 82 (Substitution lemma)**.** *For all terms t and r, and all command c, assume some variable x not free in r and some other variable $y \neq x$. Then the following equalities hold.*

$$\begin{array}{lcl}
(t[x := r])^\bullet & \equiv_\beta & \lambda\pi.\, t^\bullet[x := r^\bullet]\ \pi \\
(t[x := r])_y & \equiv_\beta & \lambda\pi.\, (t_y[x := r^\bullet]\ \pi) \odot (t_x[x := r^\bullet]\ \pi \gg\!= \lambda\rho.\, r_y\ \rho) \\
(t[x := r])_\alpha & \equiv_\beta & \lambda\pi.\, (t_\alpha[x := r^\bullet]\ \pi) \odot (t_x[x := r^\bullet]\ \pi \gg\!= \lambda\rho.\, r_\alpha\ \rho) \\
 & & \\
(c[x := r])_y & \equiv_\beta & c_y[x := r^\bullet] \odot (c_x[x := r^\bullet] \gg\!= \lambda\rho.\, r_y\ \rho) \\
(c[x := r])_\alpha & \equiv_\beta & c_\alpha[x := r^\bullet] \odot (c_x[x := r^\bullet] \gg\!= \lambda\rho.\, r_\alpha\ \rho)
\end{array}$$

Proof. By induction on t and c.

The first two equations are proved using the same arguments and almost the same rewriting steps as the intuitionistic case. Thus, there is no need to give more details.

The third equation is trivial, assuming the first one: the $(-)_\alpha$ translation does not involve any $(-)_x$ translation, so that everything commutes straightforwardly.

The second to last equation is likewise obvious, while the last one is done by case analysis and simple application of the induction hypothesis.

□

There is a more interesting notion of substitution in the $\lambda\mu$-calculus: the $t[\alpha := r \cdot \alpha]$ involves indeed stack variables, a fact we need to reflect in the equations derived from our translation. This is why we have a classical substitution lemma, stated and proved below.

Proposition 83 (Classical substitution lemma)**.** *For all terms t and r, and all command c, assume some stack variable α not free in r and some other variable $\beta \neq \alpha$. Then the following equalities hold.*

$$\begin{array}{lcl}
(t[\alpha := r \cdot \alpha])^\bullet & \equiv_\beta & \lambda\pi.\, t^\bullet[\alpha := (r^\bullet, \alpha)]\ \pi \\
(t[\alpha := r \cdot \alpha])_y & \equiv_\beta & \lambda\pi.\ \ (t_y[\alpha := (r^\bullet, \alpha)]\ \pi) \odot \\
 & & \qquad (t_\alpha[\alpha := (r^\bullet, \alpha)]\ \pi \gg\!= \lambda(_, f).\, f\ r^\bullet\ \alpha \gg\!= \lambda\rho.\, r_y\ \rho) \\
(t[\alpha := r \cdot \alpha])_\alpha & \equiv_\beta & \lambda\pi.\, t_\alpha[\alpha := (r^\bullet, \alpha)]\ \pi \gg\!= \lambda(_, f).\, f\ r^\bullet\ \alpha \\
(t[\alpha := r \cdot \alpha])_\beta & \equiv_\beta & \lambda\pi.\ \ (t_\beta[\alpha := (r^\bullet, \alpha)]\ \pi) \odot \\
 & & \qquad (t_\alpha[\alpha := (r^\bullet, \alpha)]\ \pi \gg\!= \lambda(_, f).\, f\ r^\bullet\ \alpha \gg\!= \lambda\rho.\, r_\beta\ \rho) \\
 & & \\
(c[\alpha := r \cdot \alpha])_y & \equiv_\beta & \quad c_y[\alpha := (r^\bullet, \alpha)] \odot \\
 & & \quad (c_\alpha[\alpha := (r^\bullet, \alpha)] \gg\!= \lambda(_, f).\, f\ r^\bullet\ \alpha \gg\!= \lambda\rho.\, r_y\ \rho) \\
(c[\alpha := r \cdot \alpha])_\alpha & \equiv_\beta & \quad c_\alpha[\alpha := (r^\bullet, \alpha)] \gg\!= \lambda(_, f).\, f\ r^\bullet\ \alpha \\
(c[\alpha := r \cdot \alpha])_\beta & \equiv_\beta & \quad c_\beta[\alpha := (r^\bullet, \alpha)] \odot \\
 & & \quad (c_\alpha[\alpha := (r^\bullet, \alpha)] \gg\!= \lambda(_, f).\, f\ r^\bullet\ \alpha \gg\!= \lambda\rho.\, r_\beta\ \rho)
\end{array}$$

Proof. By induction on t and c.

We will only detail some interesting cases, all the remaining ones are direct applications of the induction hypothesis together with rewriting of the multiset equalities.

- Let us consider the command $[\alpha]\, t$. This is the most interesting case, because it introduces a new use of the stack variable α in the term. We have:

$$\begin{aligned}
& (([\alpha]\, t)[\alpha := r \cdot \alpha])_y \\
\equiv_\beta \quad & ([\alpha]\, t[\alpha := r \cdot \alpha]\ r)_y \\
\equiv_\beta \quad & ((t[\alpha := r \cdot \alpha])_y\ (r^\bullet, \alpha)) \odot \\
& ((t[\alpha := r \cdot \alpha])^\bullet\ (r^\bullet, \alpha) \gg\!= \lambda(_, f).\, f\ r^\bullet\ \alpha \gg\!= \lambda\rho.\, r_y\ \rho) \\
\equiv_\beta \quad & (t_y[\alpha := (r^\bullet, \alpha)]\ (r^\bullet, \alpha)) \odot \\
& (t_\alpha[\alpha := (r^\bullet, \alpha)]\ (r^\bullet, \alpha) \gg\!= \lambda(_, f).\, f\ r^\bullet\ \alpha \gg\!= \lambda\rho.\, r_y\ \rho) \odot \\
& (t^\bullet[\alpha := (r^\bullet, \alpha)]\ (r^\bullet, \alpha) \gg\!= \lambda(_, f).\, f\ r^\bullet\ \alpha \gg\!= \lambda\rho.\, r_y\ \rho) \\
\equiv_\beta \quad & ([\alpha]\, t)_y[\alpha := (r^\bullet, \alpha)] \odot \\
& ([\alpha]\, t)_\alpha[\alpha := (r^\bullet, \alpha)] \gg\!= \lambda(_, f).\, f\ r^\bullet\ \alpha \gg\!= \lambda\rho.\, r_y\ \rho
\end{aligned}$$

  This was the equality we were looking for. We used the commutative cut rules to factor back the two last components of the union into the reverse stack translation.

  The $(-)_\beta$ case is treated exactly the same, by symmetry. We only have to look at the $(-)_\alpha$ case.

$$\begin{aligned}
& (([\alpha]\, t)[\alpha := r \cdot \alpha])_\alpha \\
\equiv_\beta \quad & ([\alpha]\, t[\alpha := r \cdot \alpha]\ r)_\alpha \\
\equiv_\beta \quad & ((t[\alpha := r \cdot \alpha])_\alpha\ (r^\bullet, \alpha)) \odot \\
& ((t[\alpha := r \cdot \alpha])^\bullet\ (r^\bullet, \alpha) \gg\!= \lambda(_, f).\, f\ r^\bullet\ \alpha \gg\!= \lambda\rho.\, r_\alpha\ \rho) \\
\equiv_\beta \quad & (t_\alpha[\alpha := (r^\bullet, \alpha)]\ (r^\bullet, \alpha) \gg\!= \lambda(_, f).\, f\ r^\bullet\ \alpha) \odot \\
& (t^\bullet[\alpha := (r^\bullet, \alpha)]\ (r^\bullet, \alpha) \gg\!= \lambda(_, f).\, f\ r^\bullet\ \alpha \gg\!= \lambda\rho.\, r_\alpha\ \rho) \\
\equiv_\beta \quad & t_\alpha[\alpha := (r^\bullet, \alpha)]\ (r^\bullet, \alpha) \gg\!= \lambda(_, f).\, f\ r^\bullet\ \alpha
\end{aligned}$$

  This is the required equality. Note that we crucially used the fact that α was not free in r to get rid of the second component of the union.

□

These equations should be rather understandable, when adapting the simulation theorem to stack variables. They acknowledge the fact that both usual and stack substitution may trigger new accesses to both usual and stack variables. Thus, we can still read them as indicating from where the various accesses come.

By using those lemmas, we can directly conclude about the preservation of β-equivalence in the $\lambda\mu$-calculus.

Theorem 28 (Computational soundness)**.** *If $t \equiv_\beta u$, then:*

- $t^\bullet \equiv_\beta u^\bullet$
- *for any variable x, $t_x \equiv_\beta u_x$*
- *for any stack variable α, $t_\alpha \equiv_\beta u_\alpha$*

Proof. This amounts to applying the various substitution lemmas. Reduction of $(\lambda x.\, t)\, u$ is handled by the usual substitution lemma, reduction of $(\mu\alpha.\, c)\, u$ is handled by the classical substitution lemma, and reduction of $[\alpha]\, \mu\beta.\, c$ is trivial. In the first two cases we use the fact that x (resp. α) is not free in u to be allowed to apply those lemmas. All remaining context-closure rules are straightforward.

□

10.3.6 KAM simulation

The KAM simulation theorem is readily adapted to the classical case. As stack variables may also appear in head position of the machine, we can observe the terms against which they are reducing. This will form the multisets returned by the stack reverse translation.

All KAM translations need to be adjusted to the classical case, but this is really straightforward, so we will not detail anything here.

Conjecture 2 (KAM simulation)**.** *Let t be a $\lambda\mu$-term, σ a KAM environment and π a KAM stack. Assume some variable x not free in σ nor in any closure of π.*
If $\langle (t, \sigma) \mid \pi \rangle \longrightarrow^* \langle (x, \tau) \mid \rho \rangle$ for some τ and ρ, then $\rho^\bullet \in (t_x \ltimes \sigma^\bullet)\ \pi^\bullet$.

Let t be a $\lambda\mu$-term, σ a KAM environment and π a KAM stack. Assume some stack variable α not free in σ nor in any closure of π.
If $\langle (t, \sigma) \mid \pi \rangle \longrightarrow^* \langle (u, \tau) \mid (\alpha, \sigma') \rangle$ for some σ' and τ, then $u^\bullet \ltimes \tau^\bullet \in (t_\alpha \ltimes \sigma^\bullet)\ \pi^\bullet$.

10.4 Call-by-value translation

Once again, thanks to the linear decomposition, we can have for free an adaptation of the Dialectica translation to the call-by-value case by taking the right decomposition. The call-by-value translation is a little more involved than the by-name variants, because of its salient feature, that is, the notion of values.

10.4.1 Call-by-value

We briefly recall here the definition of the call-by-value reduction semantics. The main difference with the call-by-name reduction is that we only substitute values in the usual β-reduction.

In the remainder of this section, we will only look at the negative fragment of the λ-calculus, but as we will also look at call-by-value semantics in more expressive settings, we give here the definition of call-by-value for a λ-calculus with inductive types.

Definition 130 (Values)**.** We define the syntactic notion of values v, w as the subset of usual λ-terms described by the following grammar.

$$v, w := x \mid \lambda x.\, t \mid () \mid (v_1, v_2) \mid \mathrm{inl}\ v \mid \mathrm{inr}\ v$$

Call-by-value has essentially the same redexes as call-by-name, except that we restrict objects being substituted to values.

Definition 131 (Call-by-value)**.** A term t reduces to r in call-by-value, written $t \to_{\beta v} r$, when the pair (t, r) is in the relation generated by the rules below, and closed by congruence.

$$\begin{array}{lll}
(\lambda x.\, t)\ v & \to_{\beta v} & t[x := v] \\
\texttt{match}\ ()\ \texttt{with}\ () \mapsto u & \to_{\beta v} & u \\
\texttt{match}\ (v, w)\ \texttt{with}\ (x, y) \mapsto u & \to_{\beta v} & u[x := v, y := w] \\
\texttt{match}\ \mathrm{inl}\ v\ \texttt{with}\ [x \mapsto u_1 \mid y \mapsto u_2] & \to_{\beta v} & u_1[x := v] \\
\texttt{match}\ \mathrm{inr}\ v\ \texttt{with}\ [x \mapsto u_1 \mid y \mapsto u_2] & \to_{\beta v} & u_2[y := v]
\end{array}$$

The equivalence generated by these rules is written $\equiv_{\beta v}$.

10.4.2 Type translation

The call-by-value linear decomposition $[\![A]\!]_{\mathrm{v}}$ of an intuitionistic type A in inductively defined on A as follows.

$$\begin{array}{lll}
[\![\alpha]\!]_{\mathrm{v}} & := & !\alpha \\
[\![A \to B]\!]_{\mathrm{v}} & := & !([\![A]\!]_{\mathrm{v}} \multimap [\![B]\!]_{\mathrm{v}})
\end{array}$$

We will depart from the usual sequent translation as described in section 3.4.2 in this section, for ease of translation. We are actually going to add spurious bang connectives in the environment, so that we recover a tractable sequent translation. Indeed, in the fragment we consider, we know that all the translated types start with a bang.

We will therefore add those bang connectives explicitly, and shift our point of view in the witness translation. The criterion that demonstrates that we are not writing nonsense comes from the fact we recover the preservation of call-by-value semantics at the end, and no other one.

First, we tweak the decomposition to extrude all bang connectives into the englobing arrow, by defining a decomposition $[\![-]\!]_{\hat{\mathrm{v}}}$ as follows.

Definition 132 (Extruded decomposition)**.** We define the extruded call-by-value decomposition $[\![-]\!]_{\hat{\mathrm{v}}}$ by induction on our source intuitionistic types as follows.

$$\begin{array}{lll}
[\![\alpha]\!]_{\hat{\mathrm{v}}} & := & \alpha \\
[\![A \to B]\!]_{\hat{\mathrm{v}}} & := & ![\![A]\!]_{\hat{\mathrm{v}}} \multimap ![\![B]\!]_{\hat{\mathrm{v}}}
\end{array}$$

As explained above, this is a mere change in the point of view of the translation, as we have the following property.

Proposition 84. *For any type A, $[\![A]\!]_{\mathrm{v}} = ![\![A]\!]_{\hat{\mathrm{v}}}$.*

Proof. By induction on A. □

Basing ourselves on this type translation, we translate sequents as

$$[\![\Gamma_1, \dots, \Gamma_n \vdash A]\!]_{\mathrm{v}} := ![\![\Gamma_1]\!]_{\hat{\mathrm{v}}}, \dots, ![\![\Gamma_n]\!]_{\hat{\mathrm{v}}} \vdash ![\![A]\!]_{\hat{\mathrm{v}}}$$

The equivalence lemma stated before ensures us that this is actually the same sequent translation we would have recovered, should we have taken the standard decomposition. The modified decomposition gives us the following type translation through the Dialectica interpretation.

Proposition 85. *We have the following type unfoldings*

$$\begin{aligned} \mathbb{W}([\![A \to B]\!]_{\hat{\mathrm{v}}}) &:= \begin{cases} \mathbb{W}([\![A]\!]_{\hat{\mathrm{v}}}) \to \mathbb{W}([\![B]\!]_{\hat{\mathrm{v}}}) \\ \times \\ \mathbb{K}([\![B]\!]_{\hat{\mathrm{v}}}) \to \mathbb{K}([\![A]\!]_{\hat{\mathrm{v}}}) \end{cases} \\ \mathbb{C}([\![A \to B]\!]_{\hat{\mathrm{v}}}) &:= \mathbb{W}([\![A]\!]_{\hat{\mathrm{v}}}) \times \mathbb{K}([\![B]\!]_{\hat{\mathrm{v}}}) \end{aligned}$$

where

$$\mathbb{K}(C) := \mathbb{W}(C) \to \mathfrak{M}\,\mathbb{C}(C)$$

We will be using the notation $\mathbb{K}(-)$ for compactness reasons, as it is rather pervasive in the call-by-value translation. As usual, we will look at the sequent translation through a simplifying isomorphism.

Proposition 86 (Sequent isomorphism)**.** *We have the following isomorphism.*

$$\mathbb{W}([\![\Gamma_1, \dots, \Gamma_n \vdash A]\!]_{\mathrm{v}}) \cong \begin{cases} \mathbb{W}([\![\Gamma]\!]_{\hat{\mathrm{v}}}) \to \mathbb{W}([\![A]\!]_{\hat{\mathrm{v}}}) \\ \times \\ \mathbb{W}([\![\Gamma]\!]_{\hat{\mathrm{v}}}) \to \mathbb{K}([\![A]\!]_{\hat{\mathrm{v}}}) \to \mathfrak{M}\,\mathbb{C}([\![\Gamma_1]\!]_{\hat{\mathrm{v}}}) \\ \times \\ \dots \\ \times \\ \mathbb{W}([\![\Gamma]\!]_{\hat{\mathrm{v}}}) \to \mathbb{K}([\![A]\!]_{\hat{\mathrm{v}}}) \to \mathfrak{M}\,\mathbb{C}([\![\Gamma_n]\!]_{\hat{\mathrm{v}}}) \end{cases}$$

As in the call-by-name case, for any term $\Gamma \vdash t : A$, we will therefore be producing two types of translations:

- A forward translation $\mathbb{W}(\llbracket\Gamma\rrbracket_{\hat{\mathrm{v}}}) \vdash t^\bullet : \mathbb{W}(\llbracket A\rrbracket_{\hat{\mathrm{v}}})$.
- For each free variable $x : R \in \Gamma$, a reverse translation $\mathbb{W}(\llbracket\Gamma\rrbracket_{\hat{\mathrm{v}}}) \vdash t_x : \mathbb{K}(\llbracket A\rrbracket_{\hat{\mathrm{v}}}) \to \mathfrak{M}\,\mathbb{C}(\llbracket R\rrbracket_{\hat{\mathrm{v}}})$.

Let us detail this translation here.

Definition 133 (Term translation)**.** Given a λ-term t and a variable x, we mutually define the translations $t^\bullet$ and t_x by induction on t below.

$$\begin{aligned}
x^\bullet &:= x \\
(\lambda x.\, t)^\bullet &:= (\lambda x.\, t^\bullet, \lambda\varphi\, x.\, t_x\ \varphi) \\
(t\ u)^\bullet &:= \mathrm{fst}\ t^\bullet\ u^\bullet \\
x_x &:= \lambda\varphi.\, \varphi\ x \\
x_y &:= \lambda\varphi.\, \emptyset \\
(\lambda y.\, t)_x &:= \lambda\varphi.\, \varphi\ (\lambda y.\, t)^\bullet \gg\!= \lambda(y, \psi).\, t_x\ \psi \\
(t\ u)_x &:= \lambda\varphi.\, (t_x\ (\lambda f.\, \{(u^\bullet, \varphi)\})) \odot (u_x\ (\lambda v.\, \mathrm{snd}\ t^\bullet\ \varphi\ v))
\end{aligned}$$

As one can witness, the forward translation is the same as in the call-by-name case, the main difference coming from the reverse one. This translation features a much more continuation-passing-style flavour than the call-by-name variant. Indeed, the reverse translation takes what is essentially a continuation expecting the eventual value of the term, and threads it all along until the translation steps across a value.

Proposition 87 (Typing soundness)**.** *The translation preserves typing, according to the sequent isomorphism given before.*

Proof. By induction on the typing derivation. □

In order to state properly the substitution lemma, we need to define an auxiliary translation that is only defined on values.

Definition 134 (Value translation)**.** For all value v and variable x, we define v^{v}_x by case analysis on the value v as follows.

$$\begin{aligned}
x^{\mathrm{v}}_x &:= \lambda\pi.\, \{\pi\} \\
y^{\mathrm{v}}_x &:= \lambda\pi.\, \emptyset \\
(\lambda y.\, t)^{\mathrm{v}}_x &:= \lambda(y, \psi).\, t_x\ \psi
\end{aligned}$$

Proposition 88 (Value typing soundness)**.** *If* $\Gamma \vdash v : A$*, then for all* $(x : R) \in \Gamma$*,*

$$\mathbb{W}(\llbracket\Gamma\rrbracket_{\hat{\mathrm{v}}}) \vdash v^{\mathrm{v}}_x : \mathbb{C}(\llbracket A\rrbracket_{\hat{\mathrm{v}}}) \to \mathfrak{M}\,\mathbb{C}(\llbracket R\rrbracket_{\hat{\mathrm{v}}})$$

Proof. Case analysis on the value. □

The value translation can be seen as the core of the reverse translation of values, as the following lemma testifies.

Proposition 89 (Value decomposition)**.** *For any value v and variable x, we have*

$$v_x \equiv_\beta \lambda\varphi.\, \varphi\ v^\bullet \gg\!= \lambda\pi.\, v^{\mathrm{v}}_x\ \pi$$

Proof. By case analysis on the value and multiset rewriting. □

This property corresponds to the fact that one needs to perform a dereliction in the linear decomposition when casting values into computations. The term provided is effectively the translation of the dereliction itself.

We can finally unroll all the necessary lemmas to make the computational soundness theorem go through.

Proposition 90 (Emptiness lemma)**.** *If x is not free in t, then $t_x \equiv_\beta \lambda\varphi.\, \emptyset$.*

Proof. By induction on t. We omit the details which are very similar to the call-by-name case. □

Proposition 91 (Substitution lemma)**.** *For any term t, any variables $x \neq y$, any value v s.t. x is not free in v, we have*

$$\begin{aligned}
(t[x := v])^\bullet &\equiv_\beta t^\bullet[x := v^\bullet] \\
(t[x := v])_y &\equiv_\beta \lambda\varphi.\, (t_y[x := v^\bullet]\ \varphi) \odot (t_x[x := v^\bullet]\ \varphi \gg\!= \lambda\pi.\, v^{\mathrm{v}}_y\ \pi)
\end{aligned}$$

Proof. The substitution lemma is, as one can observe, similar to its call-by-name counterpart, except that it is restricted to values. It is, once again, proven crawling through tedious rewriting steps. We will omit the details of the proof, although we would like to insist on an important point.

The substitution lemma does not hold when the term being substituted is not a value, and for a simple reason: in that case, the translation v^{v}_y is just undefined. One could argue, based on typing hindsights, that the following equivalence ought to be derivable for any term r.

$$(t[x := r])_y \equiv_\beta \lambda\varphi.\, (t_y[x := r^\bullet]\ \varphi) \odot (t_x[x := r^\bullet]\ \varphi \gg\!= \lambda\pi.\, r_y\ (\lambda\_.\, \{\pi\}))$$

While this equivalence preserves typing, it is not provable in general. Let us compare the simple case of the variable substitution on the rightful equivalence and the tentative one, i.e. assume $t := x$. The left-hand side gives immediately

$$(x[x := r])_y \equiv_\beta r_y$$

If r is a value, then evaluating the correct right-hand side gives

$$\begin{array}{ll} & \lambda\varphi.\,(x_y[x := r^\bullet]\ \varphi) \odot (x_x[x := r^\bullet]\ \varphi \gg\!= \lambda\pi.\,r_y^{\mathrm{v}}\ \pi) \\ \equiv_\beta & \lambda\varphi.\,\emptyset \odot (\varphi\ r^\bullet \gg\!= \lambda\pi.\,r_y^{\mathrm{v}}\ \pi) \\ \equiv_\beta & \lambda\varphi.\,\varphi\ r^\bullet \gg\!= \lambda\pi.\,r_y^{\mathrm{v}}\ \pi \end{array}$$

which is convertible to the left-hand side thanks to the value decomposition lemma. Meanwhile, if we try to do so with the tentative substitution lemma, we end up with the following.

$$\begin{array}{ll} & \lambda\varphi.\,(x_y[x := r^\bullet]\ \varphi) \odot (x_x[x := r^\bullet]\ \varphi \gg\!= \lambda\pi.\,r_y\ (\lambda\_.\,\{\pi\})) \\ \equiv_\beta & \lambda\varphi.\,\emptyset \odot (\varphi\ r^\bullet \gg\!= \lambda\pi.\,r_y\ (\lambda\_.\,\{\pi\})) \\ \equiv_\beta & \lambda\varphi.\,\varphi\ r^\bullet \gg\!= \lambda\pi.\,r_y\ (\lambda\_.\,\{\pi\}) \end{array}$$

The problem is that we have no way to relate r_y with such a term when r is not a value, because the continuation captured by r_y can be used in many ways that are incompatible with the fact we immediately apply it to a value and discard it afterwards in the value translation. Taking a well-chosen instance for r, say $r := y\ y$, shows that the two terms must be distinct. In that case we have indeed

$$r_y \equiv_\beta \lambda\varphi.\,\{(y, \varphi)\} \odot (\mathrm{snd}\ y\ \varphi\ y)$$

but applying this term and the tentative right-hand side written above to a well-chosen function, for instance $\varphi := \lambda\_.\,\emptyset$, leads to two different terms.

$$\begin{array}{lll} (r_y)\ \varphi & \equiv_\beta & \{(y, \varphi)\} \odot (\mathrm{snd}\ y\ \varphi\ y) \\ \varphi\ r^\bullet \gg\!= \lambda\pi.\,r_y\ (\lambda\_.\,\{\pi\}) & \equiv_\beta & \emptyset \end{array}$$

If our abstract multisets are implemented as true multisets, then the two terms above cannot agree, because the first one has at least one element while the second one is empty. □

Once we have the substitution lemma, it is easy to check that the translation transports βv-equivalence into plain β-equivalence.

Proposition 92 (Computational soundness)**.** *If $t \equiv_{\beta v} u$ then $t^\bullet \equiv_\beta u^\bullet$ and for any variable x, $t_x \equiv_\beta u_x$.*

Proof. Congruence rules are immediate, and it is sufficient to check that call-by-value β-redexes are transported, which is easily proved by applying the substitution lemma. □

Once again, and similarly to the call-by-name case, the translation does not capture call-by-value as a reduction strategy, but rather as a calculus. All contextual reduction rules are allowed as long as the fired redex only substitutes values. The translation only validates this calculus, though. The counter-example in the proof of the substitution can be easily worked out to obtain two terms which are β-convertible, but whose translation is not.

One could probably construct a call-by-value machine that would feature a simulation theorem similar to the call-by-name setting, but we will not study this option here. We will not study the classical-by-value translation arising from the linear decomposition either, but once again, everything comes almost for free thanks to the linear decomposition.

11 A dependently-typed Dialectica

> Dialectic, for the most part, can be constructed only *a posteriori*.
>
> *Schopenhauer about synthesis of program translations.*

This short chapter is dedicated to a proof-of-concept dependently-typed Dialectica translation. As we will see, our version of the Dialectica translation accommodates quite well within a dependent framework, at least for the purely negative fragment.

Things turns out to be frankly more complicated, if doable at all, when introducing dependent elimination. The issues raised by dependent elimination are described in section 11.5.

11.1 A simple framework: $\lambda\Pi_\omega$

The source system for our translation will be one of the many variants of a purely negative dependently-typed system, namely, $\lambda\Pi_\omega$. A rough description can be given, as a PTS [18] with an infinitely countable hierarchy of types. It is a close variant of Luo's **ECC** [77], except that it does not feature any impredicative universe nor dependent pairs. Alternatively, one can see it as the Calculus of Constructions **CC** [33] without impredicativity and extended with the aforementioned hierarchy of universes.

As this is no more than a proof of concept, we choose to use a stripped-down version of systems we could actually find implemented in the wild. In particular, we describe and use a Curry-style PTS, because it simplifies a lot the assumptions and the presentation of the translation, while allowing to reuse the proofs given at chapter 9.

This has various drawbacks. For instance, type inference and checking is undecidable in general with our presentation. Nonetheless, adapting the translation to a Church-style setting seems quite in reach, at the cost of a more intricate but essentially similar presentation. This is still work in progress.

The syntax of the terms is given below. As usual in dependent setting, there is no *a priori* syntactic difference between terms and types.

Definition 135 (Terms)**.** Terms of $\lambda\Pi_\omega$ are inductively generated by the following grammar.

$$M, N, A, B := \Box_{i\in\mathbb{N}} \mid x \mid M\ N \mid \lambda x.\, M \mid \Pi x : A.\, B$$

As usual, x is bound by the constructions $\lambda x.\, M$ and $\Pi x : A.\, B$.

Although there is no formal difference between types and terms, we will tend to use the letters M, N for normal terms and A, B for terms that should be understood as types.

Definition 136 (Reduction rules)**.** The β-equivalence $\equiv_\beta$ is the contextual closure of the rule

$$(\lambda x.\, M)\ N \equiv_\beta M[x := N]$$

Definition 137 (Environments)**.** Environments are defined as ordered lists associating variables to terms, as usual, i.e.

$$\Gamma, \Delta := \cdot \mid \Gamma, x : A$$

Definition 138 (Typing system)**.** We define here two statements mutually recursively. The statement $\vdash_{\text{wf}} \Gamma$ means that the environment Γ is well-founded, while $\Gamma \vdash M : A$ means that the term M has type A in environment Γ. The rules are given below.

$$\frac{}{\vdash_{\text{wf}} \cdot} \qquad \frac{\Gamma \vdash A : \square_i}{\vdash_{\text{wf}} \Gamma, x : A} \qquad \frac{\Gamma \vdash M : B \qquad \Gamma \vdash A : \square_i}{\Gamma, x : A \vdash M : B}$$

$$\frac{\Gamma \vdash A : \square_i}{\Gamma, x : A \vdash x : A} \qquad \frac{\vdash_{\text{wf}} \Gamma \qquad i < j}{\Gamma \vdash \square_i : \square_j} \qquad \frac{\Gamma \vdash A : \square_i \qquad \Gamma, x : A \vdash B : \square_j}{\Gamma \vdash \Pi x : A.\, B : \square_{\max(i,j)}}$$

$$\frac{\Gamma, x : A \vdash M : B \qquad \Gamma \vdash \Pi x : A.\, B : \square_i}{\Gamma \vdash \lambda x.\, M : \Pi x : A.\, B} \qquad \frac{\Gamma \vdash M : \Pi x : A.\, B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\ N : B[x := N]}$$

$$\frac{\Gamma \vdash M : B \qquad \Gamma \vdash A : \square_i \qquad A \equiv_\beta B}{\Gamma \vdash M : A}$$

This is, as one can remark, a very simple system. There is no cumulativity, no inductive types and thus no dependent elimination, and not any other fanciful stuff.

As usual, we will allow ourselves some immediate notational artifacts. For instance, we will code the non-dependent arrow $A \to B$ in a standard way, by encoding it as $\Pi x : A.\, B$ where x does not appear in B.

11.2 The target system

Exactly as the translation of chapter 9, the target language of our translation is slightly richer than its corresponding source calculus. Similarly to the simply-typed case, where we needed to add pairs in the target calculus to represent the famous first-class stacks, we now need to add their dependent counterpart, unsurprisingly named dependent pairs, also known as Σ-types.

Obviously, we also need to accommodate for the use of abstract multisets, so that we also need them in the target calculus. The consequences of the presence of such a structure will be discussed more in detail later on. While Σ-types are well-known (they are present in **ECC** for instance), the soundness of multisets is questionable. For now

we refrain from more feebleness, and we concentrate on the proof of concept currently being defined.

We will name the extended system $\lambda\Pi^{\times}_{\omega}$. It will be seen as an extension of the source system $\lambda\Pi_{\omega}$, so that we only present the extensions, not the core system already exposed at the previous section. The two extensions are presented separately, for the sake of comprehensiveness.

11.2.1 Dependent pairs

There exists the same relation of adjunction between regular arrows and pairs than between Π-types and Σ-types, which is better known as the currification. We recall here the usual definition of Σ-types.

Definition 139 (Terms)**.** We extend $\lambda\Pi_{\omega}$ terms as follows.

$$M, N, A, B := \ldots \mid \Sigma x : A.\, B \mid (M, N) \mid \texttt{let}\ (x, y) := M\ \texttt{in}\ N$$

The variable x is bound by $\Sigma x : A.\, B$, as well as x and y by $\texttt{let}\ (x, y) := M\ \texttt{in}\ N$.

Reduction rules are extended with the following generator.

Definition 140 (Reduction rules)**.** We add the following generator to $\lambda\Pi_{\omega}$ reductions.

$$(\texttt{let}\ (x, y) := (M, N)\ \texttt{in}\ P) \equiv_{\beta} P[x := M, y := N]$$

Typing rules are then a simple adaptation to the dependent case of the usual pair construction.

Definition 141 (Typing rules)**.** We add the following typing rules to $\lambda\Pi_{\omega}$.

$$\frac{\Gamma \vdash A : \square_i \qquad \Gamma, x : A \vdash B : \square_j}{\Gamma \vdash \Sigma x : A.\, B : \square_{\max(i,j)}}$$

$$\frac{\Gamma \vdash \Sigma x : A.\, B : \square_i \qquad \Gamma \vdash M : A \qquad \Gamma \vdash N : B[x := M]}{\Gamma \vdash (M, N) : \Sigma x : A.\, B}$$

$$\frac{\Gamma \vdash M : \Sigma x : A.\, B \qquad \Gamma, x : A, y : B \vdash N : C[z := (x, y)] \qquad z \text{ fresh}}{\Gamma \vdash \texttt{let}\ (x, y) := M\ \texttt{in}\ N : C[z := M]}$$

Note that we chose to use a let notation instead of the usual pattern-matching we used throughout our previous definitions because it is slightly less verbose in a dependent setting. Yet, there is no formal difference between both otherwise.

This system allows to encode non-dependent pairs in the same fashion $\lambda\Pi_{\omega}$ allows to encode non-dependent arrows.

Notation 11. We write $A \times B$ for the term $\Sigma x : A.\, B$ where x is not free in B.

Up to well-formation of types, this shorthand admits the same typing rules as the usual non-dependent pairs.

11.2.2 Multisets

As for the non-dependent translation, we also need to add multisets. While pairs needed to be made dependent, this is not the case for multisets. We simply adapt our definitions a tiny bit.

Definition 142 (Terms)**.** Terms are extended as follows.

$$M, N, A, B := \ldots \mid \mathfrak{M}\, A \mid \{M\} \mid M \gg\!= N \mid M \odot N \mid \emptyset$$

Note that we could have simply added the above term constructors as mere constants. This would not have changed the presentation much.

The reduction rules are likewise adapted.

Definition 143 (Reduction rules)**.** The reduction rules are extended with the obvious context closures induced by the above terms, together with the redexes below.

- Monadic laws:

$$\begin{array}{lll} \{M\} \gg\!= F & \equiv_\beta & F\ M \\ M \gg\!= \lambda x.\, \{x\} & \equiv_\beta & M \\ M \gg\!= F \gg\!= G & \equiv_\beta & M \gg\!= \lambda x.\, G\ (F\ x) \end{array}$$

- Monoidal laws:

$$\begin{array}{lll} \emptyset \odot M & \equiv_\beta & M \\ M \odot N & \equiv_\beta & N \odot M \\ M \odot (N \odot P) & \equiv_\beta & (M \odot N) \odot P \end{array}$$

- Distributivity laws:

$$\begin{array}{lll} \emptyset \gg\!= F & \equiv_\beta & \emptyset \\ M \odot N \gg\!= F & \equiv_\beta & (M \gg\!= F) \odot (N \gg\!= F) \\ M \gg\!= \lambda x.\, \emptyset & \equiv_\beta & \emptyset \\ M \gg\!= \lambda x.\, (F\ x) \odot (G\ x) & \equiv_\beta & (M \gg\!= F) \odot (N \gg\!= G) \end{array}$$

- Commutative cuts:

$$\begin{array}{lll} \texttt{let}\ (x, y) := M\ \texttt{in}\ \emptyset & \equiv_\beta & \emptyset \\ \texttt{let}\ (x, y) := M\ \texttt{in}\ N_1 \odot N_2 & \equiv_\beta & \\ \qquad (\texttt{let}\ (x, y) := M\ \texttt{in}\ N_1) \odot (\texttt{let}\ (x, y) := M\ \texttt{in}\ N_2) & & \\ \texttt{let}\ (x, y) := M\ \texttt{in}\ \{N\} & \equiv_\beta & \{\texttt{let}\ (x, y) := M\ \texttt{in}\ N\} \\ \texttt{let}\ (x, y) := M\ \texttt{in}\ (N \gg\!= F) & \equiv_\beta & \texttt{let}\ (x, y) := M\ \texttt{in}\ N \gg\!= F \end{array}$$

These are exactly the same rules as in the non-dependent case, except that we used a let notation instead of a match-based one. Likewise, typing rules are, once again, the same as the non-dependent case, except for a few additional bells and whistles.

Definition 144 (Typing rules)**.** We add the following rules to $\lambda\Pi_\omega$.

$$\frac{\Gamma \vdash A : \square_i}{\Gamma \vdash \mathfrak{M}\, A : \square_i} \qquad \frac{\Gamma \vdash A : \square_i}{\Gamma \vdash \varnothing : \mathfrak{M}\, A} \qquad \frac{\Gamma \vdash M : A}{\Gamma \vdash \{M\} : \mathfrak{M}\, A} \qquad \frac{\Gamma \vdash M : A \qquad \Gamma \vdash N : A}{\Gamma \vdash M \odot N : \mathfrak{M}\, A}$$

$$\frac{\Gamma \vdash M : \mathfrak{M}\, A \qquad \Gamma \vdash N : A \to \mathfrak{M}\, B \qquad \Gamma \vdash A \to \mathfrak{M}\, B : \square_i}{\Gamma \vdash M \gg\!= N : \mathfrak{M}\, B}$$

Strangely enough, we do not require the bind operator to be dependent in the translation: we only use it in a simply-typed fashion.

11.3 The dependent Dialectica translation

Actually, there is almost nothing to do to port the Dialectica translation to the purely negative dependent case. We have chiefly to adapt our results in the untyped case. Most of the trickery of the translation lies in the fact we allowed ourselves to use rich multiset reductions, further resulting in preservation of β-equivalence.

11.3.1 Rationale

We give here the principles underlying what a dependent Dialectica translation is. This can be summarized into the two main additions from the simply-typed λ-calculus into $\lambda\Pi_\omega$:

- The shift from a simple arrow $A \to B$ to a dependent product $\Pi x : A.\, B$;
- The need to handle properly types as terms, requiring for instance a term translation for $\square_i$ and $\Pi x : A.\, B$.

Each of these items will be addressed separately. Let us explain how we are to proceed.

The handling of the dependent arrow turns out to be almost immediate, actually. We recall that in the non-dependent version, we had the following type translations.

$$\begin{aligned} \mathbb{W}(A \to B) &:= \begin{cases} \mathbb{W}(A) \to \mathbb{W}(B) \\ \qquad \times \\ \mathbb{C}(B) \to \mathbb{W}(A) \to \mathfrak{M}\, \mathbb{C}(A) \end{cases} \\ \mathbb{C}(A \to B) &:= \mathbb{W}(A) \times \mathbb{C}(B) \end{aligned}$$

Now, the main difference comes from the fact that B may actually depend on a term of type A. But luckily, the translation provides us with such a term, up to a violation of the linear decomposition. Indeed, by reordering the two arguments in the reverse proof of the arrow, and making the arrows and the product over $\mathbb{W}(A)$ dependent, we can pose

$$\begin{array}{rcl} \mathbb{W}(\Pi x : A.\, B) & := & \left\{ \begin{array}{l} \Pi x : \mathbb{W}(A).\, \mathbb{W}(B) \\ \qquad \times \\ \Pi x : \mathbb{W}(A).\, \mathbb{C}(B) \to \mathfrak{M}\, \mathbb{C}(A) \end{array} \right. \\ \mathbb{C}(\Pi x : A.\, B) & := & \Sigma x : \mathbb{W}(A).\, \mathbb{C}(B) \end{array}$$

such that all translations of B have an $x : \mathbb{W}(A)$ in their scope. Note that this is not so innocuous from the point of view of linear logic: we broke the factorization through the linear decomposition without any remorse. Nonetheless, the translation is still inspired by a linear decomposition, so we are not that heterodox.

The translation of types as term may seem more difficult at first sight, but we are actually rather constrained. First, it is legitimate that in the above explanation, we ruthlessly gave the type translation in terms of $\mathbb{W}(-)$ and $\mathbb{C}(-)$ without explaining how they make sense in the current setting. The relevant question here is: how should we be translating types to accommodate the legacy translations $\mathbb{W}(-)$ and $\mathbb{C}(-)$?

There is a sensible answer. We can simply identify the translation of types seen as *terms* with the pair of their $\mathbb{W}(-)$ and $\mathbb{C}(-)$ type translations. This is actually fairly standard. When translating dependent type theories, one usually translates types so as to pack any additional information about this type into the type-as-term translation. For instance, the dependent forcing translation of [63] packs types with a proof that they are monotonous, something which is usually proven externally in simply-typed variants. Thus we are no exception.

Hence, the type translations can be reduced to macros, typically by enforcing

$$A^\bullet := (\mathbb{W}(A), \mathbb{C}(A))$$

for any type A. This allows to *define* $\mathbb{W}(-)$ and $\mathbb{C}(-)$ from $(-)^\bullet$ rather than as an external translation. Therefore, we simply pose

$$\begin{array}{rcl} \mathbb{W}(A) & := & \mathrm{fst}\ A^\bullet \\ \mathbb{C}(A) & := & \mathrm{snd}\ A^\bullet \end{array}$$

and we are done.

While we actually already defined $(\Pi x : A.\, B)^\bullet$ in the previous paragraph, one can wonder what the translation $\square_i{}^\bullet$ could be. From what we just said, looking at the typing rule $\square_i : \square_j$ for $i < j$, we have the constraints

$$\begin{array}{rcl} \square_i{}^\bullet & : & \mathrm{fst}\ \square_j{}^\bullet \\ \mathrm{snd}\ \square_i{}^\bullet & : & \square_k \end{array}$$

for some k. As we will see, the practical choice for $\mathrm{snd}\ \square_i{}^\bullet$ is essentially irrelevant, as long as it is a type. This is why we simply pose

$$\square_i{}^\bullet := (\square_i \times \square_i, \square_i)$$

which satisfies the constraints.

Finally, we did not talk at all about the reverse translation for types. There does not seem to be a lot of design space for it. The simplest choice to do is to deny any computational content to types, that is, we morally enforce

$$A_x := \lambda\pi.\, \emptyset$$

when A is a closed type.

11.3.2 The dependent Dialectica

We summarize all of our rationale given in the previous section in one definition. This is straightforward.

Definition 145 (Dependent Dialectica)**.** We mutually define the translations $(-)^\bullet$ and $(-)_x$ by induction below, where $\mathbb{W}(M) := \mathrm{fst}\ M^\bullet$ and $\mathbb{C}(M) := \mathrm{snd}\ M^\bullet$.

$$\begin{array}{lcl}
x^\bullet & := & x \\
(\lambda x.\, M)^\bullet & := & (\lambda x.\, M^\bullet, \lambda x\, \pi.\, M_x\ \pi) \\
(M\ N)^\bullet & := & \mathrm{fst}\ M^\bullet\ N^\bullet \\
(\Pi x : A.\, B)^\bullet & := & \left(\begin{array}{c} \left(\begin{array}{c} \Pi x : \mathbb{W}(A).\, \mathbb{W}(B) \\ \times \\ \Pi x : \mathbb{W}(A).\, \mathbb{C}(B) \to \mathfrak{M}\, \mathbb{C}(A) \end{array} \right), \\ \Sigma x : \mathbb{W}(A).\, \mathbb{C}(B) \end{array} \right) \\
(\Box_i)^\bullet & := & (\Box_i \times \Box_i, \Box_i) \\
x_x & := & \lambda\pi.\, \{\pi\} \\
x_y & := & \lambda\pi.\, \emptyset \\
(\lambda y.\, M)_x & := & \lambda(y, \pi).\, t_x\ \pi \\
(M\ N)_x & := & \lambda\pi.\, (\mathrm{snd}\ M^\bullet\ N^\bullet\ \pi \gg\!= \lambda\rho.\, N_x\ \rho) \odot (M_x\ (N^\bullet, \pi)) \\
(\Pi x : A.\, B)_x & := & \lambda\pi.\, \emptyset \\
(\Box_i)_x & := & \lambda\pi.\, \emptyset
\end{array}$$

This is almost the same as the simply-typed translation, except for the commutations explained before, and the special handling of types. As usual, we used generalized patterns in λ-abstraction, properly defined as an abstraction followed by a let-matching.

Theorem 29 (Computational soundness)**.** *If* $M \equiv_\beta N$ *then*

$$M^\bullet \equiv_\beta N^\bullet$$

$$M_x \equiv_\beta N_x$$

Proof. Essentially the same proof as the simply-typed case. All the auxiliary lemmas pass just through. We just need to have a look at the status of those lemmas on the two new syntactic types constructs $\square_i$ and $\Pi x : A.\, B$.

- Emptiness lemma is trivial on syntactic types, because they have already an empty translation.
- Substitution lemma is hardly more difficult to prove. Commutation with the forward translation is obvious, while commutation with the reverse translation always collapses to an empty term $\lambda\pi.\, \emptyset$.
- Adding types as terms does not add any supplementary redex, only the context rules. By construction there is nothing to prove.

□

The previous theorem finally allows us to state and prove the soundness of typing of our translation.

Theorem 30 (Typing soundness)**.** *Assume* $\Gamma \vdash M : A$*, then*

$$\mathbb{W}(\Gamma) \vdash M^\bullet : \mathbb{W}(A)$$

$$\mathbb{W}(\Gamma) \vdash M_x : \mathbb{C}(A) \to \mathfrak{M}\,\mathbb{C}(U)$$

when $(x : U) \in \Gamma$ *and where* $\mathbb{W}(\Gamma)$ *stands for the pointwise application of* $\mathbb{W}(-)$ *to* Γ*.*

Proof. By induction on the typing derivation. We also need to prove at the same time that the translation preserves well-formedness of contexts, but it is actually obvious. We focus on the most interesting cases.

- Assume $\Gamma \vdash \square_i : \square_j$. We first need to show that $\mathbb{W}(\Gamma) \vdash (\square_i)^\bullet : \mathbb{W}(\square_j)$ which is equivalent by unfolding and conversion to $\mathbb{W}(\Gamma) \vdash (\square_i \times \square_i, \square_i) : \square_j \times \square_j$. This is easily proven thanks to the induction hypotheses.

  We also need to prove that $\mathbb{W}(\Gamma) \vdash (\square_i)_x : \mathbb{C}(\square_j) \to \mathfrak{M}\,\mathbb{C}(U)$. This simply unfolds to $\mathbb{W}(\Gamma) \vdash \lambda\pi.\, \emptyset : \square_j \to \mathfrak{M}\,\mathbb{C}(U)$ which is once again easily typed, owing to the fact that U was present in the well-formed environment Γ, so that it is a type and the above arrow is well-typed.

- Assume $\Gamma \vdash \Pi x : A.\, B : \square_{\max(i,j)}$. We need to prove

$$\mathbb{W}(\Gamma) \vdash \left(\left(\begin{array}{c} \Pi x : \mathbb{W}(A).\, \mathbb{W}(B) \\ \times \\ \Pi x : \mathbb{W}(A).\, \mathbb{C}(B) \to \mathfrak{M}\,\mathbb{C}(A) \end{array} \right), \Sigma x : \mathbb{W}(A).\, \mathbb{C}(B) \right) : \mathbb{W}(\square_{\max(i,j)})$$

  which amounts to

$$\mathbb{W}(\Gamma) \vdash \left(\begin{array}{c} \Pi x : \mathbb{W}(A).\, \mathbb{W}(B) \\ \times \\ \Pi x : \mathbb{W}(A).\, \mathbb{C}(B) \to \mathfrak{M}\,\mathbb{C}(A) \end{array} \right) : \square_{\max(i,j)}$$

$$\mathbb{W}(\Gamma) \vdash \Sigma x : \mathbb{W}(A).\, \mathbb{C}(B) : \square_{\max(i,j)}$$

which are in turn easily derived from the hypotheses.

The reverse translation is handled just as in the previous case.

- Conversion rule is handled thanks to the computational soundness theorem.
- Other cases are similar to the simply-typed case, except for additional well-formedness of types and contexts.

□

There are several remarks to make here. As already mentioned before, the precise choice for $\mathbb{C}(\square_i)$ does not really matter, as long as it is a type. We could have chosen a unit type if $\lambda\Pi^{\times}_{\omega}$ was equipped with one. In any case, we never construct them whatsoever.

All the magic comes from the fact that our translation preserves β-equivalence, allowing us to make the conversion rule work unmodified. As we will see in the next section, this can represent an issue.

11.4 Practical feasibility

The translation provided in the previous section is explicitly branded as a proof-of-concept. If we wanted to implement it in an actual proof assistant such as Coq for instance, we would quickly stumble upon both practical and theoretical issues.

11.4.1 Church-style encoding

For decidability purposes, most of the proof systems based on the Curry-Howard correspondence use Church-style terms, that is, terms which carry type annotations. Typically, λ-abstractions are required to be annotated as $\lambda x : A.\, M$, and pattern matchings are annotated with the return type of their branches.

It seems our translation can be adapted to this setting, at the cost of a more technical definition. Indeed, we would need the whole typing derivation to provide the translation. Think for instance about the application case $\Gamma \vdash M\ N : B[x := M]$. Without the actual type of the application, we would not be able to write out the reverse translation

$$(M\ N)_x := \lambda\pi : \mathbb{C}(B)[x := M^{\bullet}]. \ldots$$

because we would need to annotate the λ. This is surely workable, but this requires a rather heavily annotated system. Annotation should also be put on the multiset-operating constructions to retrieve decidability.

Likewise, this would imply that the proof of preservation of β-equivalence and preservation of typing to be done mutually recursively, because only equality over well-typed terms, or at least annotated enough, would make sense.

This would have required a lot of technical details, and this is why we did not push the current attempt further, although we believe this particular point to be the easiest to overcome.

11.4.2 Actual multisets

The implementation of multisets is the most delicate detail of the issues of our proof of concept. First, being a quotient, it is difficult to encode it without losing canonicity of the representation. We could think of it as an opaque datatype with built-in equivalence, but that would require a dedicated system.

We actually do not even know if the equivalence of two well-typed terms according to the rules given at definition 143 is decidable. If this were not the case, it would put a threat upon the feasibility of our translation.

An alternative path that seems more reasonable is the conjoint use of higher inductive types [105] to represent multisets, together with encoding tricks *à la* Oury [91] to get rid of the necessity of built-in equivalence rules to make the conversion rule go through.

- Higher inductive types, also known as HITs, are a recent byproduct of the research program of homotopic type theory [105]. They provide an elegant way to build the long-awaited quotients in type theory. HITs can be seen as inductive types forced to obey additional equality rules, thus effectively producing a quotient. Our finite multisets naturally fit in this framework, as they are but lists up to commutation. This would not give us the definitional equations we hope for though, as they only allow for additional propositional equalities over the considered objects.
- To work around the definitional equality requirement, we can probably use extensionality encodings, as described in [91] and used in the dependent forcing encoding [63] for instance. The main idea of these encodings is that conversion rules in the term are made relevant, by replacing it with a rewriting over a propositional equality constructed from the fact that the source terms are β-equivalent. This is more involved than it sounds, because this equality percolates throughout the term. Nonetheless, our translation seems particularly fitted for this kind of tricks.

Alas, we do not have a perfectly working translation at hand, but we hope that the above ideas will allow us to transport the dependent Dialectica translation in an actual implementation of dependent type theory.

11.5 Towards dependent elimination

Even putting aside the issues of the translation, our source system $\lambda\Pi_\omega$ was totally devoid of any positive types, and therefore did not feature any type of dependent elimination. Dependent elimination is essential in the daily use of dependent type theory, so we ought to check whether it can be adapted to the dependent Dialectica translation.

The answer is disappointing: we probably cannot, at least as is. We hint in the remaining of this section at the root of this impossibility, by focusing on the archetypal sum type.

Definition 146 (Plain sum type)**.** We extend terms of $\lambda\Pi_\omega$ and $\lambda\Pi_\omega^\times$ with the sum type as follows.

- Terms are extended as follows.

$$M, N, A, B := \ldots \mid A + B \mid \mathrm{inl}\ M \mid \mathrm{inr}\ M \mid \mathtt{match}\ M\ \mathtt{with}\ [x \mapsto N_1 \mid y \mapsto N_2]$$

- The following typing rules are added.

$$\frac{\Gamma \vdash A : \square_i \qquad \Gamma \vdash B : \square_i}{\Gamma \vdash A + B : \square_i}$$

$$\frac{\Gamma \vdash A + B : \square_i \qquad \Gamma \vdash M : A}{\Gamma \vdash \mathrm{inl}\ M : A + B} \qquad \frac{\Gamma \vdash A + B : \square_i \qquad \Gamma \vdash M : B}{\Gamma \vdash \mathrm{inr}\ M : A + B}$$

$$\frac{\Gamma \vdash M : A + B \qquad \begin{array}{c}\Gamma, x : A \vdash N_1 : C \\ \Gamma, y : B \vdash N_2 : C\end{array} \qquad \Gamma \vdash C : \square_i}{\Gamma \vdash \mathtt{match}\ M\ \mathtt{with}\ [x \mapsto N_1 \mid y \mapsto N_2] : C}$$

- The usual reduction and congruence rules are added both to $\lambda\Pi_\omega$ and $\lambda\Pi^\times_\omega$ together with the commutative cuts for the sum type from definition 117 in $\lambda\Pi^\times_\omega$.

Note with attention how the elimination rule remains non-dependent.

We can indeed translate the above extension according to the simply-typed translation given at section 10.1.

Definition 147. We extend translation 145 with the following translations.

$$\begin{array}{lcl}
(A + B)^\bullet & := & \left(\begin{array}{c}\mathbb{W}(A) + \mathbb{W}(B), \\ (\mathbb{W}(A) \to \mathfrak{M}\,\mathbb{C}(A)) \times (\mathbb{W}(B) \to \mathfrak{M}\,\mathbb{C}(B))\end{array}\right) \\
(\mathbf{inl}\ M)^\bullet & := & \mathbf{inl}\ M^\bullet \\
(\mathbf{inr}\ M)^\bullet & := & \mathbf{inr}\ M^\bullet \\
(\mathtt{match}\ M\ \mathtt{with}\ [x \mapsto N_1 \mid y \mapsto N_2])^\bullet & := & \mathtt{match}\ M^\bullet\ \mathtt{with}\ [x \mapsto N_1{}^\bullet \mid y \mapsto N_1{}^\bullet] \\
(A + B)_z & := & \lambda\pi.\, \emptyset \\
(\mathbf{inl}\ M)_z & := & \lambda(\varphi, \psi).\, \varphi\ M^\bullet \gg\!= \lambda\chi.\, M_z\ \chi \\
(\mathbf{inr}\ M)_z & := & \lambda(\varphi, \psi).\, \psi\ M^\bullet \gg\!= \lambda\chi.\, M_z\ \chi \\
(\mathtt{match}\ M\ \mathtt{with}\ [x \mapsto N_1 \mid y \mapsto N_2])_z & := & \lambda\pi.\ (\mathtt{match}\ M^\bullet\ \mathtt{with}\ [x \mapsto N_{1z}\ \pi \mid y \mapsto N_{2z}\ \pi]) \\
 & & \qquad\qquad \odot \\
 & & \qquad (M_z\ ((\lambda x.\, N_{1x}\ \pi), (\lambda y.\, N_{2y}\ \pi)))
\end{array}$$

This is a direct adaptation of the principles we exposed before. Apart from the type $A + B$ which is translated according to the usual type scheme, the other terms keep the same translation as in the non-dependent case.

Proposition 93. *The soundness theorems hold in the system with sums.*

Proof. This is almost immediate for computational soundness, by applying the soundness result from the non-dependent setting. We only have to check that the translations for $A + B$ also respect it, which is, as in the negative fragment, merely a matter of staring at the translated terms (no additional redexes and trivial congruence closure).

Typing soundness goes through without much effort. The proof is similar to the simply-typed extended system, except for the following points.

- We have to check that the translations of $A + B$ have the right type, which is true by construction (a pair of types, and an empty term).
- Additional typability constraints not present in the simply-typed case come as usual from the hypothesis of well-formedness of the environment.

□

The issue is more stringent whenever trying to interpret dependent elimination, because it simply does not typecheck. Let us look at the example to grasp the nature of the failure. Assume we replace the elimination rule of the sum type both in $\lambda\Pi_\omega$ and $\lambda\Pi^\times_\omega$ with the one below

$$\frac{\Gamma \vdash M : A + B \qquad \begin{array}{c}\Gamma, x : A \vdash N_1 : C[z := \mathrm{inl}\ x] \\ \Gamma, y : B \vdash N_2 : C[z := \mathrm{inr}\ y]\end{array} \qquad \Gamma, z : A + B \vdash C : \square_i}{\Gamma \vdash \mathtt{match}\ M\ \mathtt{with}\ [x \mapsto N_1 \mid y \mapsto N_2] : C[z := M]}$$

where z is fresh. Then all the previous translated terms typecheck except for the reverse translation of the pattern-matching. Let us consider the translation of the above rule, and let us put one annotation in the resulting term to see why.

$$\lambda\pi : \mathbb{W}(C[z := M^\bullet]).\ (\mathtt{match}\ M^\bullet\ \mathtt{with}\ [x \mapsto N_{1z}\ \pi \mid y \mapsto N_{2z}\ \pi]) \odot (M_z\ ((\lambda x.\, N_{1x}\ \pi), (\lambda y.\, N_{2y}\ \pi)))$$

The stack π has to have type $\mathbb{W}(C[z := M])$ to conform with the soundness theorem. The problem stems from the fact that N_1 and N_2 are in turn expecting a term of type $\mathbb{W}(C[z := \mathrm{inl}\ x])$ and $\mathbb{W}(C[z := \mathrm{inr}\ y])$ respectively, so that the terms $N_{iz}\ \pi$, $N_{1x}\ \pi$ and $N_{2y}\ \pi$ are ill-typed. This is not much of an issue in the left side of the union, because one can recover the typability by making explicit a commutative cut in the translation, resulting in the following term.

$$\lambda\pi.\ ((\mathtt{match}\ M^\bullet\ \mathtt{with}\ [x \mapsto N_{1z} \mid y \mapsto N_{2z}])\ \pi) \odot (M_z\ ((\lambda x.\, N_{1x}\ \pi), (\lambda y.\, N_{2y}\ \pi)))$$

This is a standard technique of dependently-typed programming language, nicknamed the *convoy pattern* by Chlipala [28]. The real problem comes from the second component. Indeed, in the term $M_z\ ((\lambda x.\, N_{1x}\ \pi), (\lambda y.\, N_{2y}\ \pi))$ there is no way to relate the type of

π to the sum contained in M. This is because M_z actually implements internally an elimination by means of a CPS-like construction, as witnessed by the reverse translation of the corresponding values:

$$\begin{aligned}(\mathbf{inl}\ M)_z &:= \lambda(\varphi,\psi).\,\varphi\ M^\bullet \gg\!= \lambda\chi.\,M_z\ \chi\\ (\mathbf{inr}\ M)_z &:= \lambda(\varphi,\psi).\,\psi\ M^\bullet \gg\!= \lambda\chi.\,M_z\ \chi\end{aligned}$$

Here, the pair (φ,ψ) can be seen as the current continuation of the sum. The core issue is that the type of this continuation is totally unrelated to the actual value of the corresponding term, as we have

$$\mathbb{C}(A+B) := (\mathbb{W}(A) \to \mathfrak{M}\,\mathbb{C}(A)) \times (\mathbb{W}(B) \to \mathfrak{M}\,\mathbb{C}(B))$$

while we would rather like this product to look like an elimination on the current hole, intuitively of the form

$$\mathbb{C}(A+B) := \texttt{match}\ (\cdot)\ \texttt{with}\ [x \mapsto \mathfrak{M}\,\mathbb{C}(A) \mid y \mapsto \mathfrak{M}\,\mathbb{C}(B)]$$

where $(\cdot)$ is the value currently being built. The presence of this pattern-matching in the type should allow us to keep track of the internal elimination performed by M. We still do not know how to fix our translation though, as making the type translation depend on the term being built is a real can of worms that resists our understanding. Chapter 12 explains in more details to which extent the above translation is actually a CPS, and provides more hints at how to cope with the dependent elimination.

That is why the current state of the translation is disappointing. It allows to somehow translate the negative fragment of dependent type theory, as well as positive connectives, but does not interpret the dependent elimination that one would expect in this setting.

12 Decomposing Dialectica: Forcing, CPS and the rest

> On appelle passoires du premier ordre les passoires qui ne laissent passer ni les nouilles, ni l'eau.
>
> *Prof. Shadoko about zoology.*

This chapter is dedicated to the study of some linear decompositions, particularly arising from commutative monads, and their relation with the Dialectica translation. This will provide us with a way to construct program translations compatible with dependently typed systems.

We will hint at a way to reconstruct the Dialectica translation by successive refinements of seemingly lesser importance. The computational intuition of the KAM simulation theorem will be a great support in the design of such translations. Indeed, this will prove of greater help than just reasoning with types.

12.1 Overview

The study of the Dialectica translation in this thesis has been particularly influenced by the work of Krivine [71] and Miquel [81] on forcing in the realm of classical realizability, as well as its application to dependent type theory due to Jaber et al. [63].

The forcing translation was invented by Cohen [30] to build a model of ZFC disproving the Continuum Hypothesis, a longstanding question that had been formalized by Hilbert himself as the first problem for the twentieth century.

Seen as a program translation, it is essentially the same as the Kripke model construction [67] that was designed as a complete model for (a modal extension of) intuitionistic propositional logic. We will use interchangeably the names forcing and Kripke model in the remainder of this chapter.

At the type level, it can be thought of as an indexed translation, relating types with so-called *worlds*, that are equipped with a particular order. Whenever a world w and a type A are related, we say that w *forces* A, or equivalently that A is *valid* at w. The trick is that while provable formulae are valid at any world, some unprovable formulae may be valid from a certain world onwards. When at the right world, this allows to prove more formulae, and, in particular, in the historical case, to negate the Continuum Hypothesis.

As shown by Krivine and Miquel, the underlying translation at the level of terms is rather dull. Indeed, it is no more than the usual reader monad described at Section 3.3. The readable cell of the translation contains the current world, and may be updated

almost like the usual reader monad. The subtlety dwells in the fact that this cell can only be updated monotonously w.r.t. the order of worlds.

The forcing translation is interesting because, in addition to disproving the Continuum Hypothesis as well as being a complete model of intuitionistic logic, it also allows interesting new features, such as step-indexing, as well as providing a natural way to compute normalization-by-evaluation [20, 62, 1].

It is also folklore that the Kripke models factor through a linear decomposition. This is another common point with the Dialectica translation. While the linear decomposition of Kripke models is a useful heuristic tool, it seems not to have been studied in detail in the literature. We will recall it in this chapter.

Those two aspects, namely, the computational explanation through the KAM and the linear logic decomposition explain why it is natural to study the relationship between the Dialectica translation and the forcing translation.

We will show that there is a strong kinship linking forcing, CPS and the Dialectica translation, as soon as one considers first-class stacks.

12.2 The simplest forcing: the reader monad

We recall that given a fixed type R, the reader monad over R defined as $TA := R \to A$ is a commutative monad, so it naturally gives rise to a linear decomposition. It is actually a degenerated case of the Kripke model construction, as we will see in the next section. The reader monad can be seen as adding an additional essentially read-only memory cell of type R.

For now, we quickly give the reader monad translation both in call-by-name and in call-by-value. In the following, we assume given some fixed type R. We also assume a variable ω not appearing in the source terms.

12.2.1 Pseudo-linear translation

Without additional structure on the type R, there is no true linear translation, in the sense that we do not interpret full classical linear logic. We can nonetheless interpret the infamous intuitionistic fragment of linear logic.

These intuitionistic linear types are precisely the ones generated by the grammar given below.

$$A, B := \alpha \mid A \multimap B \mid 1 \mid A \otimes B \mid 0 \mid A \oplus B \mid !A$$

The translation we provide does not have much to do with linear logic, actually. There is no such thing as real negative types, and the dual construction is just absent.

It is rather about using a language with a built-in monad. As we define it below, the ! connective marks where we introduce the monad constructor. This may seem rather silly, but with incremental refinement, this construction is going to start to make sense in a linear fashion.

Definition 148 (Pseudo-linear translation)**.** Given a linear type A, we define the intuitionistic type $\mathbb{W}(A)$ by induction over A as follows.

- $\mathbb{W}(\alpha) := \alpha$
- $\mathbb{W}(0) := 0$
- $\mathbb{W}(A \oplus B) := \mathbb{W}(A) + \mathbb{W}(B)$
- $\mathbb{W}(1) := 1$
- $\mathbb{W}(A \otimes B) := \mathbb{W}(A) \times \mathbb{W}(B)$
- $\mathbb{W}(A \multimap B) := \mathbb{W}(A) \to \mathbb{W}(B)$
- $\mathbb{W}(!A) := R \to \mathbb{W}(A)$

For now, there is no such thing as a counter type. We will be introducing them later on.

We did not specify the rules of the intuitionistic system, so there is no soundness theorem to state. There is no term in sight either, as we only gave the type translation.

The main use of this type translation is as a heuristic tool, plugged with a given decomposition of intuitionistic logic into linear logic. This is the topic of the next sections.

12.2.2 Call-by-name reader translation

In this section, we look at the composition of the call-by-name linear decomposition coupled with the pseudo-linear translation of the reader monad. We unfold this composition in the definition given below.

Definition 149 (Type translation)**.** The translation $\mathbb{W}(\llbracket - \rrbracket_{\mathrm{n}})$ has the following unfoldings.

$$\begin{array}{lcl}
\mathbb{W}(\llbracket A \to B \rrbracket_{\mathrm{n}}) & := & (R \to \mathbb{W}(\llbracket A \rrbracket_{\mathrm{n}})) \to \mathbb{W}(\llbracket B \rrbracket_{\mathrm{n}}) \\
\mathbb{W}(\llbracket 1 \rrbracket_{\mathrm{n}}) & := & 1 \\
\mathbb{W}(\llbracket A \times B \rrbracket_{\mathrm{n}}) & := & (R \to \mathbb{W}(\llbracket A \rrbracket_{\mathrm{n}})) \times (R \to \mathbb{W}(\llbracket B \rrbracket_{\mathrm{n}})) \\
\mathbb{W}(\llbracket 0 \rrbracket_{\mathrm{n}}) & := & 0 \\
\mathbb{W}(\llbracket A + B \rrbracket_{\mathrm{n}}) & := & (R \to \mathbb{W}(\llbracket A \rrbracket_{\mathrm{n}})) + (R \to \mathbb{W}(\llbracket B \rrbracket_{\mathrm{n}}))
\end{array}$$

Basing ourselves on this type translation allows to write out a term translation easily.

Definition 150 (Term translation)**.** Let ω be a reserved term variable. We define the translation $(-)^{\bullet}$ by induction on λ-terms.

$$
\begin{array}{lcl}
x^\bullet & := & x\ \omega \\
(\lambda x.\, t)^\bullet & := & \lambda x.\, t^\bullet \\
(t\ u)^\bullet & := & t^\bullet\ (\lambda\omega.\, u^\bullet)
\end{array}
\qquad
\begin{array}{lcl}
()^\bullet & := & () \\
(t, u)^\bullet & := & (\lambda\omega.\, t^\bullet, \lambda\omega.\, u^\bullet) \\
(\mathrm{inl}\ t)^\bullet & := & \mathrm{inl}\ (\lambda\omega.\, t^\bullet) \\
(\mathrm{inr}\ t)^\bullet & := & \mathrm{inr}\ (\lambda\omega.\, t^\bullet)
\end{array}
$$

$$
\begin{array}{lcl}
(\mathtt{match}\ t\ \mathtt{with}\ () \mapsto u)^\bullet & := & \mathtt{match}\ t^\bullet\ \mathtt{with}\ () \mapsto u^\bullet \\
(\mathtt{match}\ t\ \mathtt{with}\ (x, y) \mapsto u)^\bullet & := & \mathtt{match}\ t^\bullet\ \mathtt{with}\ (x, y) \mapsto u^\bullet \\
(\mathtt{match}\ t\ \mathtt{with}\ [\cdot])^\bullet & := & \mathtt{match}\ t^\bullet\ \mathtt{with}\ [\cdot] \\
(\mathtt{match}\ t\ \mathtt{with}\ [x \mapsto u_1 \mid y \mapsto u_2])^\bullet & := & \mathtt{match}\ t^\bullet\ \mathtt{with}\ [x \mapsto u_1^\bullet \mid y \mapsto u_2^\bullet]
\end{array}
$$

We can dissert a bit about this translation. First, all rules that would correspond to a promotion rule in the source linear calculus are matched with a corresponding closure over the variable ω. This thunking property allows to freeze computations waiting for the current cell so that they can be given this cell later on. This phenomenon appears at changes of polarity, i.e. in the application rule as well as the introduction of constructors. This acknowledges the fact that in call-by-name, arguments of functions as well as subcomponents of positive constructors are never forced[1] unless they appear in head position.

The variable translation is the only one to feature this forcing, because it is the precise moment when the thunked terms appear in a needed position, so that they need the content of the reader cell.

The other cases are plain commuting with the translation, because the reader cell does not interfere with them.

The preservation of typing by this translation is almost trivial, having been designed precisely by following the typing intuition.

Proposition 94 (Typing soundness)**.** *Assume* $\Gamma_1, \ldots, \Gamma_n \vdash t : A$*. Then*

$$R \to \mathbb{W}(\llbracket \Gamma_1 \rrbracket_{\mathrm{n}}), \ldots, R \to \mathbb{W}(\llbracket \Gamma_n \rrbracket_{\mathrm{n}}), \omega : R \vdash t^\bullet : \mathbb{W}(\llbracket A \rrbracket_{\mathrm{n}})$$

Proof. By induction on the typing derivation. □

More interestingly, we also recover the preservation of β-equivalence through the translation. The most technical thing to prove is yet-another substitution lemma.

Proposition 95 (Substitution lemma)**.** *Assume* t *and* r *two* λ*-terms. Then*

$$(t[x := r])^\bullet \equiv_\beta t^\bullet[x := \lambda\omega.\, r^\bullet]$$

[1] In the computational meaning, a term which is deplorably ambiguous with the forcing translation described in the next section.

Proof. By induction on t. We crucially need the substituted term in the right-hand side to be closed w.r.t. ω for this to hold. Luckily, we prepended the translated term with a λ-abstraction on ω. We detail the cases of the pure λ-calculus fragments, as other cases are treated alike.

- Case x. The left-hand side gives immediately

$$(x[x := r])^\bullet \equiv_\beta r^\bullet$$

  while the right-hand side is

$$x^\bullet[x := \lambda\omega.\, r^\bullet] \equiv_\beta (x\ \omega)[x := \lambda\omega.\, r^\bullet] \equiv_\beta (\lambda\omega.\, r^\bullet)\ \omega \equiv_\beta r^\bullet$$

- Case $\lambda y.\, t$. We have

$$((\lambda y.\, t)[x := r])^\bullet \equiv_\beta (\lambda y.\, t[x := r])^\bullet \equiv_\beta \lambda y.\, (t[x := r])^\bullet \equiv_\beta \lambda y.\, t^\bullet[x := \lambda\omega.\, r^\bullet]$$

  from which we conclude immediately.

- Case $t\ u$. This case is the one requiring the closure of the term $\lambda\omega.\, r^\bullet$ w.r.t. to ω. We have for the left-hand side

$$\begin{array}{lll} ((t\ u)[x := r])^\bullet & \equiv_\beta & (t[x := r]\ u[x := r])^\bullet \\ & \equiv_\beta & (t[x := r])^\bullet\ (\lambda\omega.\, (u[x := r])^\bullet) \\ & \equiv_\beta & t^\bullet[x := \lambda\omega.\, r^\bullet]\ (\lambda\omega.\, u^\bullet[x := \lambda\omega.\, r^\bullet]) \\ & \equiv_\beta & (t^\bullet\ (\lambda\omega.\, u^\bullet))[x := \lambda\omega.\, r^\bullet] \end{array}$$

  from which we easily conclude. Note that the transition from the penultimate line to the last one is only valid because the bound ω present in $u^\bullet$ does not appear free in $\lambda\omega.\, r^\bullet$. This was the point of closing the substituend w.r.t. ω.

□

Preservation of β-equivalence comes for free by applying the above lemma.

Theorem 31 (Computational soundness)**.** *If $t \equiv_\beta u$, then $t^\bullet \equiv_\beta u^\bullet$.*

Proof. For redexes, it is sufficient to unfold the definitions and apply the substitution lemma, while for contextual rules, this is immediate. □

12.2.3 Call-by-value reader translation

The call-by-value translation is likewise obtained by choosing the $[\![-]\!]_{\mathrm{v}}$ decomposition over the $[\![-]\!]_{\mathrm{n}}$ one.

Definition 151 (Type translation)**.** The translation $\mathbb{W}([\![-]\!]_{\mathrm{v}})$ has the following unfoldings.

$$\begin{aligned}
\mathbb{W}([\![A \to B]\!]_{\mathrm{v}}) &:= R \to \mathbb{W}([\![A]\!]_{\mathrm{v}}) \to \mathbb{W}([\![B]\!]_{\mathrm{v}}) \\
\mathbb{W}([\![1]\!]_{\mathrm{v}}) &:= 1 \\
\mathbb{W}([\![A \times B]\!]_{\mathrm{v}}) &:= \mathbb{W}([\![A]\!]_{\mathrm{v}}) \times \mathbb{W}([\![B]\!]_{\mathrm{v}}) \\
\mathbb{W}([\![0]\!]_{\mathrm{v}}) &:= 0 \\
\mathbb{W}([\![A + B]\!]_{\mathrm{v}}) &:= \mathbb{W}([\![A]\!]_{\mathrm{v}}) + \mathbb{W}([\![B]\!]_{\mathrm{v}})
\end{aligned}$$

Term translation is almost as easy as in the call-by-name case, if not easier.

Definition 152 (Term translation)**.** Let ω be a reserved term variable. We define the translation $(-)^\bullet$ by induction on λ-terms.

$$\begin{aligned}
x^\bullet &:= x \\
(\lambda x.\, t)^\bullet &:= \lambda \omega\, x.\, t^\bullet \\
(t\ u)^\bullet &:= t^\bullet\ \omega\ u^\bullet
\end{aligned}
\qquad
\begin{aligned}
()^\bullet &:= () \\
(t, u)^\bullet &:= (t^\bullet, u^\bullet) \\
(\mathrm{inl}\ t)^\bullet &:= \mathrm{inl}\ t^\bullet \\
(\mathrm{inr}\ t)^\bullet &:= \mathrm{inr}\ t^\bullet
\end{aligned}$$

$$\begin{aligned}
(\texttt{match}\ t\ \texttt{with}\ () \mapsto u)^\bullet &:= \texttt{match}\ t^\bullet\ \texttt{with}\ () \mapsto u^\bullet \\
(\texttt{match}\ t\ \texttt{with}\ (x, y) \mapsto u)^\bullet &:= \texttt{match}\ t^\bullet\ \texttt{with}\ (x, y) \mapsto u^\bullet \\
(\texttt{match}\ t\ \texttt{with}\ [\cdot])^\bullet &:= \texttt{match}\ t^\bullet\ \texttt{with}\ [\cdot] \\
(\texttt{match}\ t\ \texttt{with}\ [x \mapsto u_1 \mid y \mapsto u_2])^\bullet &:= \texttt{match}\ t^\bullet\ \texttt{with}\ [x \mapsto u_1{}^\bullet \mid y \mapsto u_2{}^\bullet]
\end{aligned}$$

As usual when studying call-by-value, the translation verifies some nice properties when applied to values.

Proposition 96. *If v is a value, then ω is not free in $v^\bullet$.*

Proof. Straightforward induction. □

Once again, typing preservation by this translation is immediate.

Proposition 97 (Typing soundness)**.** *Assume $\Gamma_1, \ldots, \Gamma_n \vdash t : A$. Then*

$$\mathbb{W}([\![\Gamma_1]\!]_{\mathrm{v}}), \ldots, \mathbb{W}([\![\Gamma_n]\!]_{\mathrm{v}}), \omega : R \vdash t^\bullet : R \to \mathbb{W}([\![A]\!]_{\mathrm{v}})$$

Likewise, for any value v such that $\Gamma_1, \ldots, \Gamma_n \vdash v : A$, we have

$$\mathbb{W}([\![\Gamma_1]\!]_{\mathrm{v}}), \ldots, \mathbb{W}([\![\Gamma_n]\!]_{\mathrm{v}}) \vdash v^\bullet : \mathbb{W}([\![A]\!]_{\mathrm{v}})$$

The substitution lemma holds, adapted to the call-by-value case, that is, when only substituting values.

Proposition 98 (Substitution lemma)**.** *Let t be a term and v a value. Then*

$$(t[x := v])^\bullet \equiv_\beta t^\bullet[x := v^\bullet]$$

Proof. Direct induction on the term t. The essential point here is that translated values do not contain the free variable ω, as in the call-by-name translation, so that we can push them under ω-bindings without any concern. □

Theorem 32 (Computational soundness)**.** *If $t \equiv_{\beta v} u$ then $t^\bullet \equiv_\beta u^\bullet$.*

Proof. Essentially an application of the substitution lemma. □

12.3 Forcing in more detail

As explained by Krivine [71] and Miquel [81], forcing is no more than a refinement of the reader monad, where the type R is given more structure. Namely, it is enriched with an order, and all updates of the cell must be monotonous w.r.t. this order. They rather present it as a dedicated slot on the stack, because they are working in a classical setting where the relative order of the reader and continuation effect handlers must be made explicit, but there is no such choice in an intuitionistic setting.

12.3.1 Linear translation

This is better explained by making explicit the linear type translation. In order to be able to talk about the order over R, we need a first-order typing system[2], though we do not really bother about it in this section. Indeed, we only want the reader to get a taste of the translation without dwelling too much on details. Thus we assume we are now in a dependently typed system, featuring a relation

$$\leq : R \to R \to \texttt{Prop}$$

which is reflexive and transitive, i.e. there exist two terms with the following types.

$$\begin{array}{rcl} \text{refl} & : & \forall \omega : R.\, \omega \leq \omega \\ \text{trans} & : & \forall \omega_1\, \omega_2\, \omega_3 : R.\, \omega_1 \leq \omega_2 \to \omega_2 \leq \omega_3 \to \omega_1 \leq \omega_3 \end{array}$$

Because of this dependency, types may now mention the ambient cell of type R being read. As in the case of the simple reader, we name this ambient cell using the dedicated ω variable.

Definition 153. The linear translation for the forcing translation is defined inductively below. We insist on the fact that ω is in general free in this translation.

- $\mathbb{W}(\alpha) := \alpha$

[2] We do not make explicit the system here, but any extension of the simply-typed λ-calculus with a dependent arrow should be enough. We stick to a Curry-style calculus for its simplicity, and the fact that the computational content of the term is not buried under type annotations.

- $\mathbb{W}(0) := 0$
- $\mathbb{W}(A \oplus B) := \mathbb{W}(A) + \mathbb{W}(B)$
- $\mathbb{W}(1) := 1$
- $\mathbb{W}(A \otimes B) := \mathbb{W}(A) \times \mathbb{W}(B)$
- $\mathbb{W}(A \multimap B) := \mathbb{W}(A) \to \mathbb{W}(B)$
- $\mathbb{W}(!A) := \forall \omega' : R. \omega' \leq \omega \to (\mathbb{W}(A)[\omega := \omega'])$

As one can see, this translation is essentially the same one as the one given by the reader. The only change takes place at the level of the ! connective. There, we just made the $R \to -$ dependent and stuffed an additional lesser-than condition inside it. As we will manipulate quite often the free ω variable of the type translation, we use a proper notation.

Notation 12. We will write $\mathbb{W}_{\omega'}(A)$ for $\mathbb{W}(A)[\omega := \omega']$ for the sake of conciseness.

In the remaining of this section, we will quickly look at the term translation in call-by-name and call-by-value. The treatment of the dependency in the call-by-value setting has been thoroughly described in [63], as it raises issues of its own. In particular, lots of tricks have to be considered to handle properly the translation, especially carrying a lot of equalities around to emulate convertibility.

As this is only an overview, and we are actually only concerced about the computational content, we will just give the shape of the terms without further care of design flaws that would happen in a fully dependent setting.

For the sake of readability, we will write the dependently-typed terms refl and trans with implicit arguments, that is, we will omit the leading applications of arguments of type R when writing terms. We will reserve the ω letter for variables of type R, and we will use the convention that terms of type $\omega_1 \leq \omega_2$ for some ω_1 and ω_2 are written in the fraktur typeface $\mathfrak{p}, \mathfrak{q}$, etc.

12.3.2 Call-by-name decomposition

As done before, we first make explicit the translation on types.

Definition 154 (Type translation)**.** The translation $\mathbb{W}(\llbracket - \rrbracket_{\mathrm{n}})$ has the following unfoldings.

$$
\begin{aligned}
\mathbb{W}_\omega(\llbracket \alpha \rrbracket_{\mathrm{n}}) &:= \alpha \\
\mathbb{W}_\omega(\llbracket A \to B \rrbracket_{\mathrm{n}}) &:= (\forall \omega'. \omega' \leq \omega \to \mathbb{W}_{\omega'}(\llbracket A \rrbracket_{\mathrm{n}})) \to \mathbb{W}_\omega(\llbracket B \rrbracket_{\mathrm{n}}) \\
\mathbb{W}_\omega(\llbracket 1 \rrbracket_{\mathrm{n}}) &:= 1 \\
\mathbb{W}_\omega(\llbracket A \times B \rrbracket_{\mathrm{n}}) &:= (\forall \omega'. \omega' \leq \omega \to \mathbb{W}_{\omega'}(\llbracket A \rrbracket_{\mathrm{n}})) \times (\forall \omega'. \omega' \leq \omega \to \mathbb{W}_{\omega'}(\llbracket B \rrbracket_{\mathrm{n}})) \\
\mathbb{W}_\omega(\llbracket 0 \rrbracket_{\mathrm{n}}) &:= 0 \\
\mathbb{W}_\omega(\llbracket A + B \rrbracket_{\mathrm{n}}) &:= (\forall \omega'. \omega' \leq \omega \to \mathbb{W}_{\omega'}(\llbracket A \rrbracket_{\mathrm{n}})) + (\forall \omega'. \omega' \leq \omega \to \mathbb{W}_{\omega'}(\llbracket B \rrbracket_{\mathrm{n}}))
\end{aligned}
$$

The term translation is more involved than in the plain reader case, because it requires substituting all free variables of the translated term when applying what would come from a promotion rule in the corresponding linear sequent. There are various ways to present this operation. One could use a substitution threaded along the translation, but we rather choose to stick to a global operation done at boxing time.

Definition 155 (Lift)**.** Let t be a λ-term and ω some variable. Assume that the free variables of t range over $\omega, x_1, \dots, x_n$. Assume another term $\mathfrak{p}$. We write $\uparrow^{\mathfrak{p}}_{\omega} t$ for

$$\uparrow^{\mathfrak{p}}_{\omega} t := t[x_i := \lambda\omega'\, \mathfrak{q}.\, x_i\ \omega'\ (\mathrm{trans}\ \mathfrak{q}\ \mathfrak{p})]$$

for some fresh variables ω' and $\mathfrak{q}$.

We now turn to the term translation itself.

Definition 156 (Term translation)**.** The translation $t^\bullet$ is defined by induction on t as follows.

$$\begin{array}{lcl} x^\bullet & := & x\ \omega\ \mathrm{refl} \\ (\lambda x.\, t)^\bullet & := & \lambda x.\, t^\bullet \\ (t\ u)^\bullet & := & t^\bullet\ (\lambda\omega\, \mathfrak{p}.\, \uparrow^{\mathfrak{p}}_{\omega} u^\bullet) \end{array} \qquad \begin{array}{lcl} ()^\bullet & := & () \\ (t, u)^\bullet & := & (\lambda\omega\, \mathfrak{p}.\, \uparrow^{\mathfrak{p}}_{\omega} t^\bullet, \lambda\omega\, \mathfrak{p}.\, \uparrow^{\mathfrak{p}}_{\omega} u^\bullet) \\ (\mathrm{inl}\ t)^\bullet & := & \mathrm{inl}\ (\lambda\omega\, \mathfrak{p}.\, \uparrow^{\mathfrak{p}}_{\omega} t^\bullet) \\ (\mathrm{inr}\ t)^\bullet & := & \mathrm{inr}\ (\lambda\omega\, \mathfrak{p}.\, \uparrow^{\mathfrak{p}}_{\omega} t^\bullet) \end{array}$$

$$\begin{array}{lcl} (\mathtt{match}\ t\ \mathtt{with}\ () \mapsto u)^\bullet & := & \mathtt{match}\ t^\bullet\ \mathtt{with}\ () \mapsto u^\bullet \\ (\mathtt{match}\ t\ \mathtt{with}\ (x, y) \mapsto u)^\bullet & := & \mathtt{match}\ t^\bullet\ \mathtt{with}\ (x, y) \mapsto u^\bullet \\ (\mathtt{match}\ t\ \mathtt{with}\ [\cdot])^\bullet & := & \mathtt{match}\ t^\bullet\ \mathtt{with}\ [\cdot] \\ (\mathtt{match}\ t\ \mathtt{with}\ [x \mapsto u_1 \mid y \mapsto u_2])^\bullet & := & \mathtt{match}\ t^\bullet\ \mathtt{with}\ [x \mapsto {u_1}^\bullet \mid y \mapsto {u_2}^\bullet] \end{array}$$

Note that this definition by itself is actually badly specified, as the toplevel ω actually appears in the boxed terms of the form $\lambda\omega\, \mathfrak{p}.\, \uparrow^{\mathfrak{p}}_{\omega} t^\bullet$ because the type of $\mathfrak{p}$ involves it, so that the trans operator mentions it as an implicit argument. This is actually not a real issue because the actual type of the terms $\mathfrak{p}$ are enough to desambiguate this collision. We could have given a presentation with De Bruijn indices, wich would have resolved all the ambiguities, but this would not give any valuable intuition of what is going on in the skeleton of the translation.

As for the type translation, if we wanted to be totally out of any such issue, we should parameterize the translation by the name of the dedicated variable ω, so that we would not have to rename this variable explicitly on the fly.

If we look at this translation from a higher level, we see that this is really an enriched version of the reader monad. The main difference lies in the fact that we have to keep track of the monotonous conditions imposed on the readable cell. From the

linear logic side, the distinction appears when playing with exponential rules, that is, in call-by-name, in the variable case (which features weakening and dereliction) and in the application case (which features contraction and promotion). Only the two latter rules are observable in the forcing translation: dereliction corresponds to reflexivity of the relation, while promotion is implemented by the lift operation which is no more that a refined transitivity.

As we did not fully specify the system we were working on, hence any results about typing soundness is actually somewhat void. Nonetheless, for any sensible instance of such a system, the results would hold. Recall that this section does not aim at providing a faithful account of this translation, but rather a rough description that will ultimately justify our journey through the Dialectica decomposition.

In order to show the typing soundness for the whole translation, we first need to show that the lifting operation preserves the typing through the translation.

Proposition 99. *Let t be a term of type*

$$\omega : R, \vec{x}_i : (\forall \omega' : R.\, \omega' \leq \omega \to \Gamma_i) \vdash t : A$$

then we have

$$\omega : R, \vec{x}_i : (\forall \omega' : R.\, \omega' \leq \omega \to \Gamma_i) \vdash \lambda \omega\, \mathfrak{p}.\, (\uparrow^{\mathfrak{p}}_{\omega} t) : \forall \hat{\omega} : R.\, \hat{\omega} \leq \omega \to A[\omega := \hat{\omega}]$$

Proof. By induction on the term t. The only interesting case is the variable case x_i, where we have

$$\uparrow^{\mathfrak{p}}_{\omega} x_i \equiv \lambda \omega'\, \mathfrak{q}.\, x_i\ \omega'\ (\mathrm{trans}\ \mathfrak{q}\ \mathfrak{p})$$

so we conclude by typing of the trans term, as $\mathfrak{p} : \hat{\omega} \leq \omega$ and $\mathfrak{q} : \omega' \leq \hat{\omega}$.

□

The typing soundness is then a mere corollary of this proposition.

Proposition 100 (Typing soundness)**.** *If $\Gamma \vdash t : A$, then*

$$\omega : R, \vec{x} : \mathbb{W}_{\omega}(!\llbracket \Gamma \rrbracket_{\mathrm{n}}) \vdash t^{\bullet} : \mathbb{W}_{\omega}(\llbracket A \rrbracket_{\mathrm{n}})$$

Proof. By induction on the typing derivation. The cases which are not trivial are either the variable case, or the cases involving lifting. The former amounts to observe that in the translation, variables from the environment have type

$$x_i : \mathbb{W}_{\omega}(!\llbracket \Gamma_i \rrbracket_{\mathrm{n}}) \equiv \forall \omega' : R.\, \omega' \leq \omega \to \mathbb{W}_{\omega'}(\llbracket \Gamma_i \rrbracket_{\mathrm{n}})$$

so that by applying them to the current cell value and reflexivity one obtains the correct type. The latter ones are handled thanks to the previous lemma. □

Assuming enough reduction properties on the refl and trans terms, we can also recover preservation of β-equivalence by the translation. The required equations are actually expected. We want those terms to be compatible with the semantic of the underlying order, namely

$$\begin{array}{rcl} \mathrm{trans}\ \mathrm{refl}\ \mathfrak{p} & \equiv_\beta & \mathfrak{p} \\ \mathrm{trans}\ \mathfrak{p}\ \mathrm{refl} & \equiv_\beta & \mathfrak{p} \\ \mathrm{trans}\ (\mathrm{trans}\ \mathfrak{p}\ \mathfrak{q})\ \mathfrak{r} & \equiv_\beta & \mathrm{trans}\ \mathfrak{p}\ (\mathrm{trans}\ \mathfrak{q}\ \mathfrak{r}) \end{array}$$

This is similar to the properties of multisets in the Dialectica translation, where refl would have the rôle of $\{\cdot\}$ and trans the one of $\gg=$. Indeed, the first pair is induced by the translation of the dereliction, while the second is by promotion. Moreover, both structures can be thought of as a monad over the ambient category of types and terms. We will push this idea further in the next sections.

Proposition 101 (Lifting)**.** *Let t be a term. Then the following equalities hold.*

$$\uparrow_\omega^{\mathrm{refl}} t^\bullet \equiv_\beta t^\bullet$$

$$\uparrow_\omega^{(\mathrm{trans}\ \mathfrak{p}\ \mathfrak{q})} t^\bullet \equiv_\beta \uparrow_\omega^{\mathfrak{q}} \uparrow_\omega^{\mathfrak{p}} t^\bullet$$

Proof. As usual, because of the definition of lifting, it is sufficient to look at the case of free variables of $t^\bullet$ (and hence of t). We know that all free variables x in $t^\bullet$ distinct from ω are in an application of the form $x\ \omega_0\ \mathfrak{r}$ (where ω_0 may be ω). Hence, through the lifting of refl we get

$$(\lambda\omega'\,\mathfrak{q}.\, x\ \omega'\ (\mathrm{trans}\ \mathfrak{q}\ \mathrm{refl}))\ \omega_0\ \mathfrak{r} \equiv_\beta x\ \omega_0\ (\mathrm{trans}\ \mathfrak{r}\ \mathrm{refl}) \equiv_\beta x\ \omega_0\ \mathfrak{r}$$

so that, in the end, the lifted term is unchanged. The same arguments allows to conclude for the lifting of trans.

□

Proposition 102 (Substitution lemma)**.** *Let t and r be λ-terms. Then the following equality holds.*

$$(t[x := r])^\bullet \equiv_\beta t^\bullet[x := \lambda\omega\,\mathfrak{p}.\uparrow_\omega^{\mathfrak{p}} r^\bullet]$$

Proof. By induction on the term t. The arguments are essentially the same as for the reader translation.

□

The reduction lemma follows immediately.

Proposition 103. *If $t \equiv_\beta u$ then $t^\bullet \equiv_\beta u^\bullet$.*

Proof. As usual. □

This concludes the presentation of the call-by-name variant of the forcing translation. Miquel's presentation [81] in the framework of classical realizability is also call-by-name, although it is defined in a classical setting. The linear decomposition can be tweaked to accommodate this fact and recover this particular variant. We advocate the call-by-name variant over the call-by-value one, because, as we will see in the next section, it is lighter to use, at least for the negative fragment, a recurring phenomenon in translations arising from linear decompositions. In particular, we would like this section to offer a little more visibility to the call-by-name forcing, which is the oft-forgotten flavour of forcing in the literature.

12.3.3 Call-by-value decomposition

The call-by-value presentation is the standard one for Kripke models. This is unfortunate, because as we will see, this implies more technical apparatus at the level of terms than for the call-by-name translation. Conversely, the type translation is more straightforward.

We first head to give the expanded type translation.

Definition 157 (Type translation)**.** The translation $\mathbb{W}([\![-]\!]_{\mathrm{v}})$ has the following unfoldings, using the already mentioned indexed notation.

$$\begin{aligned}
\mathbb{W}_\omega([\![\alpha]\!]_{\mathrm{v}}) &:= \alpha \\
\mathbb{W}_\omega([\![A \to B]\!]_{\mathrm{v}}) &:= \forall \omega'.\, \omega' \leq \omega \to \mathbb{W}_{\omega'}([\![A]\!]_{\mathrm{v}}) \to \mathbb{W}_{\omega'}([\![B]\!]_{\mathrm{v}}) \\
\mathbb{W}_\omega([\![1]\!]_{\mathrm{v}}) &:= 1 \\
\mathbb{W}_\omega([\![A \times B]\!]_{\mathrm{v}}) &:= \mathbb{W}_\omega([\![A]\!]_{\mathrm{v}}) \times \mathbb{W}_\omega([\![B]\!]_{\mathrm{v}}) \\
\mathbb{W}_\omega([\![0]\!]_{\mathrm{v}}) &:= 0 \\
\mathbb{W}_\omega([\![A + B]\!]_{\mathrm{v}}) &:= \mathbb{W}_\omega([\![A]\!]_{\mathrm{v}}) + \mathbb{W}_\omega([\![B]\!]_{\mathrm{v}})
\end{aligned}$$

The traditional presentation of the term translation relies on the following monotonicity lemma. This lemma is useless in the call-by-name case, where its equivalent is embodied by the lifting operation.

Proposition 104 (Monotonicity)**.** *For all type A, there is a term run_A with the following type.*

$$\vdash \forall \omega_1\, \omega_2.\, \omega_2 \leq \omega_1 \to \mathbb{W}_{\omega_1}([\![A]\!]_{\mathrm{v}}) \to \mathbb{W}_{\omega_2}([\![A]\!]_{\mathrm{v}})$$

Proof. By induction on A. The only non-trivial case is the arrow case, so we detail it. We pose

$$\begin{aligned}
\mathrm{run}_{A\to B} := \;& \lambda \omega_1\, \omega_2\, (\mathfrak{p} : \omega_2 \leq \omega_1). \\
& \lambda f : \mathbb{W}_{\omega_1}([\![A \to B]\!]_{\mathrm{v}}). \\
& \lambda \omega_3\, (\mathfrak{q} : \omega_3 \leq \omega_2)\, (x : \mathbb{W}_{\omega_3}([\![A]\!]_{\mathrm{v}})).\, f\ \omega_3\ (\mathrm{trans}\ \mathfrak{q}\ \mathfrak{p})\ x
\end{aligned}$$

and it is easy to see that the above term has the right type.

□

The linear decomposition sheds an interesting light upon the meaning of this lemma. Indeed, this theorem is, up to a reordering of universal quantifications, no more than the evaluation property from Section 3.4.2. When writing it alternatively as

$$\vdash \forall \omega_1.\, \mathbb{W}_{\omega_1}([\![A]\!]_{\mathrm{v}}) \to (\forall \omega_2.\, \omega_2 \leq \omega_1 \to \mathbb{W}_{\omega_2}([\![A]\!]_{\mathrm{v}}))$$

then we clearly recognize that this is the translated formulation of the sequent

$$\vdash [\![A]\!]_{\mathrm{v}} \multimap\, !\,[\![A]\!]_{\mathrm{v}}$$

This is no mere coincidence: the uses of the monotonicity lemma in the historical presentation correspond precisely to the uses of the evaluation property in the soundness proof of the call-by-value linear decomposition. There is a remarkable issue raised by the monotonicity lemma, though. As easily observed, it breaks the parametricity of the translation, because run_A is only defined for a given type A that we will need to have at hand to be able to define the translation. This defect is absent in the call-by-name presentation. Such an itch is similar to the need of typing in the historical Dialectica translation to create merging and dummy terms.

To cope with this limitation, we assume in the remaining of this section that the variables of the source terms are annotated with their simple type, i.e. we are considering simply-typed terms in Church-style. Annotations will be left implicit whenever they are not useful, and we will consider that fresh variables appearing in the target of the translation are not wearing any annotation. This will simplify the forecoming definitions, even though we could present them in a purely formal way.

Using these typing annotations, we can construct a lifting operation similar to the call-by-name $\uparrow^{\mathfrak{p}}_{\omega}(-)$, but requiring the typing annotations.

Definition 158 (Lift)**.** We define the call-by-value lift $\Uparrow^{\mathfrak{p}}_{\omega'} t$ as follows:

$$\Uparrow^{\mathfrak{p}}_{\omega'} t := t[{x_i}^X := \mathrm{run}_X\ \omega\ \omega'\ \mathfrak{p}\ x_i]$$

where x_i ranges over the annotated free variables of type X of the term t.

As usual regarding the translation of whole sequents, there is a choice to be made on the exact placement of exponential modalities, already described at Section 3.4.2. The presentation given by Jaber et al. is based on the alternative decomposition, but we will stick to the standard one, because we find it to be somehow easier to define w.r.t. the definitions we already gave in the call-by-value reader and in the call-by-name forcing cases.

Definition 159 (Term translation)**.** The translation $t^\bullet$ is defined by induction on t as follows.

$$\begin{array}{lcl} x^\bullet & := & x \\ (\lambda x.\, t)^\bullet & := & \lambda\omega'\,\mathfrak{p}.\Uparrow^{\mathfrak{p}}_{\omega'}(\lambda x.\, t^{\bullet'}) \\ (t\ u)^\bullet & := & t^\bullet\ \omega\ \mathrm{refl}\ u^\bullet \end{array} \qquad \begin{array}{lcl} ()^\bullet & := & () \\ (t,u)^\bullet & := & (t^\bullet, u^\bullet) \\ (\mathrm{inl}\ t)^\bullet & := & \mathrm{inl}\ t^\bullet \\ (\mathrm{inr}\ t)^\bullet & := & \mathrm{inr}\ t^\bullet \end{array}$$

$$\begin{array}{lcl} (\mathtt{match}\ t\ \mathtt{with}\ () \mapsto u)^\bullet & := & \mathtt{match}\ t^\bullet\ \mathtt{with}\ () \mapsto u^\bullet \\ (\mathtt{match}\ t\ \mathtt{with}\ (x,y) \mapsto u)^\bullet & := & \mathtt{match}\ t^\bullet\ \mathtt{with}\ (x,y) \mapsto u^\bullet \\ (\mathtt{match}\ t\ \mathtt{with}\ [\cdot])^\bullet & := & \mathtt{match}\ t^\bullet\ \mathtt{with}\ [\cdot] \\ (\mathtt{match}\ t\ \mathtt{with}\ [x \mapsto u_1 \mid y \mapsto u_2])^\bullet & := & \mathtt{match}\ t^\bullet\ \mathtt{with}\ [x \mapsto u_1{}^\bullet \mid y \mapsto u_2{}^\bullet] \end{array}$$

Here, $t^{\bullet'}$ stands for $t^\bullet[\omega' := \omega]$.

Proposition 105 (Typing soundness)**.** *If* $\Gamma \vdash t : A$*, then*

$$\omega : R, \vec{x} : \mathbb{W}_\omega([\![\Gamma]\!]_{\mathrm{v}}) \vdash t^\bullet : \mathbb{W}_\omega([\![A]\!]_{\mathrm{v}})$$

We could formulate a computational soundness theorem, but we believe this particular case to be well-known enough not to deserve further careful study. We only wanted to point out at the details of implementation arising from the call-by-value linear decomposition.

The call-by-value presentation, albeit the most used one, is not without issues. In particular, it imposes that all translated types must respect the monotonicity conditions. In complicated settings, like the translation acting on CIC described by Jaber et al. [63], this forces the term translation to embed a lot of additional information, including the fact that all types are monotone. We believe that the call-by-name presentation is better behaved and the resulting translation would probably be slightly lighter and more amenable.

12.3.4 Forcing you to repeat: a computational stuttering

As discovered by Krivine and Miquel, the call-by-name forcing translation can be seen as a program transformation that adds a reserved cell atop the KAM stack. A translated program $t^\bullet$ would then somehow ignore this stack as a meaningful argument, but would still modify it on the fly. Indeed, the simulation theorem for the forcing translation states that assuming some reduction r of the KAM

$$\langle (t, \sigma) \mid \pi \rangle \xrightarrow{r} \langle (u, \tau) \mid \rho \rangle$$

then the forcing-translated process now performs a reduction of the form

$$\langle (t^\bullet, \sigma^\bullet) \mid \omega \cdot \pi^\bullet \rangle \xrightarrow{r^\bullet}{}^* \langle (u^\bullet, \tau^\bullet) \mid (r \star \omega) \cdot \rho^\bullet \rangle$$

where $r \star \omega$ stands for a modification of ω depending on the nature of the reduction rule r considered. The very nature of the cell and the particular implementation of the $r \star \cdot$ operation is left open, as long as it satisfies some well-behavedness axioms. One can use it for logical purpose (proof of the independence of CH), computational purpose (NBE) or quantitative purpose (as in monitoring algebras [24]).

In its utmost embodiment, one can think of this cell as a reification of the current state of the machine. Indeed, if we forgot about the typing of this cell, we could just consider that it contains a process, and that the $r \star \cdot$ operator is no more than the mere performance of this reduction rule to the process to which it is applied. In this case, if the cell is initially filled with the process itself, then it will always contain a copy of the original process at the current point of execution. That is, all processes would be of the form

$$\langle (t^\bullet, \sigma^\bullet) \mid [\![\langle (t, \sigma) \mid \pi \rangle]\!] \cdot \pi^\bullet \rangle$$

for some t, σ and π where $[\![\cdot]\!]$ stands for some syntactic reification of the process as a λ-term. The monotonous constraint on the use of the cell is then reduced to the inverse reduction relation, i.e.

$$[\![\langle (t,\sigma) \mid \pi\rangle]\!] \leq [\![\langle (u,\tau) \mid \rho\rangle]\!] := \langle (u,\tau) \mid \rho\rangle \longrightarrow^* \langle (t,\sigma) \mid \pi\rangle$$

and thus the constraint on the variable is a promise that it will be fed with a latter stage of the computation.

In other words, the essence of the forcing translation is computational stuttering, i.e. duplicating the computation: to a meta-level reduction of the machine corresponds an object-level reified reduction of the original process.

12.4 A proto-Dialectica: the silly stack reader

We will not push the forcing translation to its actual limit, which is explained in Section 12.3.4. This would be rather subtle to type correctly, and we would surely need some form of dependent typing. Anyway, we will not prove so extremists in our use, for we will not reify the whole process, but rather *the current stack* only.

Indeed, we can consider that the ability to access the current stack of the process is the quintessential component of the Dialectica translation seen in its rawest form. Once we can grab it, it is sufficient to have a way to store it somewhere in order to write the reverse translations $(-)_x$. While the storing part can be worked around easily (using a global mutable memory for instance), the stack access part is fairly more involved. We propose in this section a naive approach to this problem, stemming in the forcing translation.

12.4.1 A first step into linearity

Contrarily to the usual forcing translation which has a fixed type for the cell, we will take inspiration from the stuttering intuition, and decide that it contains a stack of the (dual) type of the term currently in head position.

To this end, and in a fashion similar to the Dialectica translation, we need to introduce a new family of types $\mathbb{C}(-)$ standing for stacks of a given type. Now is the precise point where we fall into the realm of linear logic, and not just a fancy notation for a monadic decomposition. It is indeed one of the strength of linear logic to realize the existence of proper dual types, not hidden under some unnatural encoding disguise.

The leading idea is that we are going to adapt the interpretation of the ! connective, which is the one introducing accesses to the readable cell, so that the type of the cell is the dual of the current term. Formally, we go from the forcing interpretation

$$\mathbb{W}_\omega(!A) := \forall \omega' : R.\, \omega' \leq \omega \to \mathbb{W}_{\omega'}(A)$$

to the stack reader interpretation

$$\mathbb{W}(!A) := \mathbb{C}(A) \to \mathbb{W}(A)$$

The monotonicity constraint has vanished in the process, for the same reason the orthogonality did in the case of the Dialectica translation, that is, namely because it is useless when only interested in typing and computational soundness. The $\mathbb{C}(A)$ is to be fed with the current stack of the process when a dereliction of the exponential occurs.

We will concentrate first on the purely negative fragment, as the design choices arising from positive connectives will appear to be tricky. The interpretation of the sufficient fragment of (intuitionistic) linear types is given below.

Definition 160 (Type interpretation)**.** We define the types $\mathbb{W}(A)$ and $\mathbb{C}(A)$ by mutual induction on A as follows.

$$\begin{array}{lcllcl} \mathbb{W}(\alpha) & := & \alpha & \mathbb{C}(\alpha) & := & \alpha^{\perp} \\ \mathbb{W}(A \multimap B) & := & \mathbb{W}(A) \to \mathbb{W}(B) & \mathbb{C}(A \multimap B) & := & \mathbb{W}(A) \times \mathbb{C}(B) \\ \mathbb{W}(!A) & := & \mathbb{C}(A) \to \mathbb{W}(A) & \mathbb{C}(!A) & := & \mathbb{C}(A) \end{array}$$

Here, $\alpha^{\perp}$ stands for an arbitrary type fixed for each base type α.

The interpretation of witnesses is, up to the change in the interpretation of ! described above, the same as above. The counter interpretation is self-explanatory for the arrow, as we really want to have first-class stacks.

12.4.2 Call-by-name translation

Let us give a closer look at the result of this translation when composed with the call-by-name decomposition. This is just a subtle variant of the reader monad actually.

Proposition 106. *The translation* $\mathbb{W}(\llbracket - \rrbracket_{\mathrm{n}})$ *has the following unfolding.*

$$\begin{array}{lcl} \mathbb{W}(\llbracket A \to B \rrbracket_{\mathrm{n}}) & := & (\mathbb{C}(\llbracket A \rrbracket_{\mathrm{n}}) \to \mathbb{W}(\llbracket A \rrbracket_{\mathrm{n}})) \to \mathbb{W}(\llbracket B \rrbracket_{\mathrm{n}}) \\ \mathbb{C}(\llbracket A \to B \rrbracket_{\mathrm{n}}) & := & (\mathbb{C}(\llbracket A \rrbracket_{\mathrm{n}}) \to \mathbb{W}(\llbracket A \rrbracket_{\mathrm{n}})) \times \mathbb{C}(\llbracket B \rrbracket_{\mathrm{n}}) \end{array}$$

Sequents $\Gamma \vdash t : A$ *are translated to*

$$(\mathbb{C}(\llbracket \Gamma \rrbracket_{\mathrm{n}}) \to \mathbb{W}(\llbracket \Gamma \rrbracket_{\mathrm{n}})), \omega : \mathbb{C}(\llbracket A \rrbracket_{\mathrm{n}}) \vdash t^{\bullet} : \mathbb{W}(\llbracket A \rrbracket_{\mathrm{n}})$$

The ω variable in the sequent translation stands for the current content of the cell, that is, the current stack of the process, hence its type. While the above type translation is rather close to the reader monad, the term translation is more refined, because we need to modify the content of the cell so that it always matches the current stack of the process. To this end, we will keep in mind the reduction rules of the KAM. We give the translation below, and explain it afterwards.

Definition 161 (Term translation)**.** We define the $(-)^{\bullet}$ translation on λ-terms by induction below.

$$\begin{array}{lcl} x^{\bullet} & := & x\ \omega \\ (\lambda x.\, t)^{\bullet} & := & \lambda x.\, (\lambda \omega.\, t^{\bullet})\ (\mathrm{snd}\ \omega) \\ (t\ u)^{\bullet} & := & (\lambda \omega.\, t^{\bullet})\ ((\lambda \omega.\, u^{\bullet}), \omega)\ (\lambda \omega.\, u^{\bullet}) \end{array}$$

The term translation is justified as follows. In call-by-name, a variable contains a thunked object, so that we need to feed it with the current cell. This is exactly as in the usual reader monad, which explains the variable translation.

The translation of the λ-abstraction is justified by the KAM rule

$$\langle (\lambda x.\, t, \sigma) \mid c \cdot \pi \rangle \longrightarrow \langle (t, \sigma + (x := c)) \mid \pi \rangle$$

as the stack against which the t term will be cut in $\lambda x.\, t$ is no more than the second component of the current stack. The translation features a redundancy, because we would like to identify x with $\mathrm{fst}\ \omega$. This little defect will be the source of a later amelioration.

The most intricate case is, as usual in call-by-name, the application case. We recall that the associated KAM rule is of the form

$$\langle (t\ u, \sigma) \mid \pi \rangle \longrightarrow \langle (t, \sigma) \mid (u, \sigma) \cdot \pi \rangle$$

As in the case of the reader and forcing translation, the argument passed to the function is thunked by the abstraction over ω. Yet, we need to change the current stack of the t term, as witnessed by the KAM rule. Its argument in the translation is then no more than the stack corresponding to $u \cdot \pi$, where u is once again thunked.

As expected, the translation preserves typing.

Proposition 107. *If $\Gamma \vdash t : A$, then*

$$(\mathbb{C}(\llbracket \Gamma \rrbracket_{\mathrm{n}}) \to \mathbb{W}(\llbracket \Gamma \rrbracket_{\mathrm{n}})), \omega : \mathbb{C}(\llbracket A \rrbracket_{\mathrm{n}}) \vdash t^{\bullet} : \mathbb{W}(\llbracket A \rrbracket_{\mathrm{n}})$$

Proof. By induction on the typing derivation. □

The substitution lemma from the reader translation also holds.

Proposition 108. *Assume t and r two λ-terms. Then*

$$(t[x := r])^{\bullet} \equiv_{\beta} t^{\bullet}[x := \lambda\omega.\, r^{\bullet}]$$

Proof. Essentially the same proof as the reader translation. We have a little more ω-boxing occurring, but this does not matter, as the term substituted is still ω-closed. □

The preservation of β-equivalence follows directly.

Proposition 109. *If $t \equiv_{\beta} u$ then $t^{\bullet} \equiv_{\beta} u^{\bullet}$.*

Proof. We only look at the case of the reduction β-redex, which is as usual the only interesting one. We have

$$\begin{array}{rll} ((\lambda x.\, t)\ u)^{\bullet} & \equiv_{\beta} & (\lambda\omega\, x.\, (\lambda\omega.\, t^{\bullet})\ (\mathrm{snd}\ \omega))\ ((\lambda\omega.\, u^{\bullet}), \omega)\ (\lambda\omega.\, u^{\bullet}) \\ & \equiv_{\beta} & (\lambda x.\, (\lambda\omega.\, t^{\bullet})\ \omega)\ (\lambda\omega.\, u^{\bullet}) \\ & \equiv_{\beta} & (\lambda x.\, t^{\bullet})\ (\lambda\omega.\, u^{\bullet}) \\ & \equiv_{\beta} & t^{\bullet}[x := \lambda\omega.\, u^{\bullet}] \\ & \equiv_{\beta} & (t[x := u])^{\bullet} \end{array}$$

which concludes the proof.

□

12.4.3 Reading the stacks

Let us see what we gained through this translation. As advertised at the beginning of this chapter, the goal of this series of refined translation is to recover a way to observe the current stack of the process. We can indeed add in the source λ-calculus such an operator that will be easily interpreted through the stack reader translation.

As in the Dialectica translation, we need to give a way to talk about the type of stacks in the source language. We will still use the $\sim A$ notation for the stacks of dual type A.

Definition 162 (λ_ω-calculus)**.** We define the λ_ω-calculus as the usual simply-typed λ-calculus extended on types and terms as follows.

$$A, B := \ldots \mid \sim A$$

$$t, u := \ldots \mid \texttt{read}\ x\ \texttt{in}\ t$$

The typing rules are extended with the following inference rule.

$$\frac{\Gamma, x : \sim A \vdash t : A}{\Gamma \vdash \texttt{read}\ x\ \texttt{in}\ t : A}$$

We also need to inspect the stacks generated by our new construct. There is no pairs in the source calculus, so that we need to resort to an impredicative encoding to destruct arrow stacks. This is why we also assume a constant

$$\mathrm{observe}_{\to,A,B,R} : \sim(A \to B) \to (A \to \sim B \to R) \to R$$

for all A, B and R.

In the following we omit the indexes of the observe term for readability.

The semantics of this extended language is the one defined by its translation through the stack reader transformation.

Definition 163. The λ_ω-calculus is translated into the λ-calculus with pairs as follows.

- For the λ-calculus fragment, see the previous translation.
- The translation of type $\sim A$ is morally given by its linear decomposition, i.e. we would like to have
$$[\![\sim A]\!]_{\mathrm{n}} := [\![A]\!]_{\mathrm{n}}{}^{\perp}$$
and derive the interpretation by composition with $\mathbb{W}(-)$ and $\mathbb{C}(-)$. This does not work because we need to assert that stacks are values in call-by-name, and therefore they do not require the access to the current stack themselves. There is no way to overcome this defect without further polarizing our source calculus, so we work around this by postulating the type of stacks of stacks to be trivial, i.e.
$$\begin{array}{lcl} \mathbb{W}([\![\sim A]\!]_{\mathrm{n}}) & := & \mathbb{C}([\![A]\!]_{\mathrm{n}}) \\ \mathbb{C}([\![\sim A]\!]_{\mathrm{n}}) & := & 1 \end{array}$$

which breaks the linear decomposition and forces to see the $\mathbb{W}([\![-]\!]_{\mathrm{n}})$ and $\mathbb{C}([\![-]\!]_{\mathrm{n}})$ translations as a whole.

- As for term translation, we pose

$$\begin{aligned}(\texttt{read}\ x\ \texttt{in}\ t)^{\bullet} &:= t^{\bullet}[x := \lambda().\,\omega] \\ (\mathrm{observe}_{\to})^{\bullet} &:= \lambda\alpha\, k.\,\texttt{match}\ \alpha\ ()\ \texttt{with}\ (x,\alpha) \mapsto k\ (x,((\lambda().\,\alpha),\omega))\ x\ (\lambda().\,\alpha)\end{aligned}$$

Luckily, the extended translation is respectful of the type translation.

Proposition 110. *The extended constructs above have the right type through the translation, i.e. the rules described above are admissible in the source system.*

Proof. By construction. □

The actual logical expressiveness provided by these operators is actually quite disappointing. Indeed, if look at the typing rule for the term $\texttt{read}\ x\ \texttt{in}\ t$ together with the elimination over stacks of arrow type, one can easily observe that it amounts to the following generalized rule

$$\frac{\Gamma, A_1, \dots, A_n, \sim B \vdash B}{\Gamma \vdash A_1 \to \dots \to A_n \to B}$$

as the only connective of our language is the arrow. This means that reading the stack does not provide us with anything else that what we could achieve through repeated λ-abstraction. Thus we did not get anything logically. This was somehow expected. In some sense, we have recovered the read-only part of the μ binder from the $\lambda\mu$-calculus, acting as an infinitarily expanded λ-abstraction, but we lack the way to reinstate stack types as the current continuation. Consequently, stack types are not very useful.

12.4.4 Handling positive connectives

Positive connectives raise their own issues, as it is the norm in call-by-name. It is fairly obvious that we want to translate witness types of positive by themselves, up to introduction of bang connectives, as in the Dialectica translation.

The central question is rather what content we should be giving to the types $\mathbb{C}(P)$ where P is a positive type.

We can try to understand it by looking at the shape of positive stacks in the KAM. As explained in Section 10.1.2, positive stacks are functional objects rather than first-order ones. They are indeed generated by the grammar

$$\begin{aligned}\pi, \rho := \quad & \dots \\ \mid\ & () \mapsto (t, \sigma) \cdot \pi \\ \mid\ & (x, y) \mapsto (t, \sigma) \cdot \pi \\ \mid\ & [\cdot] \\ \mid\ & ([x \mapsto t_1 \mid y \mapsto t_2], \sigma) \cdot \pi\end{aligned}$$

where the variables are bound in the body of the pattern-matching branches. The problem is that the typing rules of those objects embed existential types. For instance, the pair eliminator had type

$$\frac{\sigma \vdash \Gamma \qquad \Gamma, x : A, y : B \vdash u : C \qquad \vdash \pi : C}{\vdash (x, y) \mapsto (u, \sigma) \cdot \pi : A \times B}$$

where the type C of u is absent from the resulting type for the stack (forgetting about closures, that were already collapsed in the translation). That means that, assuming our target language features some form of existential types, we would like to have morally

$$\mathbb{C}(A \times B) := \exists C.\, \mathbb{W}(A) \times \mathbb{W}(B) \to \mathbb{W}(C) \times \mathbb{C}(C)$$

but this is not possible, because C quantifies over source types, not over actual types of the target language. One could work around this by setting

$$\mathbb{C}(A \times B) := \exists \alpha.\, \exists \beta.\, \mathbb{W}(A) \times \mathbb{W}(B) \to \alpha \times \beta$$

but that would mean loosing the knowledge that α and β share a duality relation. To the best of our understanding, there is no way to ensure this fact in a simply-typed system.

In any case, the information that is retrievable from such a type is meager: the existential quantification makes the return value of the function uninformative. This is why we adopt here an especially brutal point of view: we *pose* the counter types of positive to be uninformative.

Definition 164 (Positive type translation)**.** We extend the type translation of the previous section as follows.

$$\begin{array}{lcl@{\qquad}lcl}
\mathbb{W}(\llbracket 1 \rrbracket_{\mathrm{n}}) & := & 1 & \mathbb{C}(\llbracket 1 \rrbracket_{\mathrm{n}}) & := & 1 \\
\mathbb{W}(\llbracket 0 \rrbracket_{\mathrm{n}}) & := & 0 & \mathbb{C}(\llbracket 0 \rrbracket_{\mathrm{n}}) & := & 1 \\
\mathbb{W}(\llbracket A + B \rrbracket_{\mathrm{n}}) & := & (\mathbb{C}(\llbracket A \rrbracket_{\mathrm{n}}) \to \mathbb{W}(\llbracket A \rrbracket_{\mathrm{n}})) + (\mathbb{C}(\llbracket B \rrbracket_{\mathrm{n}}) \to \mathbb{W}(\llbracket B \rrbracket_{\mathrm{n}})) & \mathbb{C}(\llbracket A + B \rrbracket_{\mathrm{n}}) & := & 1 \\
\mathbb{W}(\llbracket A \times B \rrbracket_{\mathrm{n}}) & := & (\mathbb{C}(\llbracket A \rrbracket_{\mathrm{n}}) \to \mathbb{W}(\llbracket A \rrbracket_{\mathrm{n}})) \times (\mathbb{C}(\llbracket B \rrbracket_{\mathrm{n}}) \to \mathbb{W}(\llbracket B \rrbracket_{\mathrm{n}})) & \mathbb{C}(\llbracket A \times B \rrbracket_{\mathrm{n}}) & := & 1
\end{array}$$

Strangely enough, this allows the trivialized translation to go through just fine.

Definition 165 (Extended term translation)**.** We extend the translation of the previous section with the following.

$$\begin{array}{lcl}
()^\bullet & := & () \\
(t, u)^\bullet & := & (\lambda \omega.\, t^\bullet, \lambda \omega.\, u^\bullet) \\
(\mathrm{inl}\ t)^\bullet & := & \mathrm{inl}\ (\lambda \omega.\, t^\bullet) \\
(\mathrm{inr}\ t)^\bullet & := & \mathrm{inr}\ (\lambda \omega.\, t^\bullet) \\
(\mathtt{match}\ t\ \mathtt{with}\ () \mapsto u)^\bullet & := & \mathtt{match}\ (\lambda \omega.\, t^\bullet)\ ()\ \mathtt{with}\ () \mapsto u^\bullet \\
(\mathtt{match}\ t\ \mathtt{with}\ (x, y) \mapsto u)^\bullet & := & \mathtt{match}\ (\lambda \omega.\, t^\bullet)\ ()\ \mathtt{with}\ (x, y) \mapsto u^\bullet \\
(\mathtt{match}\ t\ \mathtt{with}\ [\cdot])^\bullet & := & \mathtt{match}\ (\lambda \omega.\, t^\bullet)\ ()\ \mathtt{with}\ [\cdot] \\
(\mathtt{match}\ t\ \mathtt{with}\ [x \mapsto u_1 \mid y \mapsto u_2])^\bullet & := & \mathtt{match}\ (\lambda \omega.\, t^\bullet)\ ()\ \mathtt{with}\ [x \mapsto u_1^\bullet \mid y \mapsto u_2^\bullet]
\end{array}$$

Remark in the translation how positive values, by definition inert, drop the current stack, as all of their subterms are closed over ω. Dually, pattern-matching provides terms being eliminated with a dummy stack under the form of a empty tuple.

All the expected theorems are still true in this extended version, which is not that surprising, because of the trivial way we intepret counters of positives.

Proposition 111. *The following lemmas hold:*

- *Preservation of typing*
- *Substitution lemma*
- *Preservation of β-equivalence*

Proof. Essentially as for the base case.

We were guided by a linear decomposition, so this prevents us from many mistakes. In particular, the fact that we close terms w.r.t. ω at polarity changes is a key point for the validity of the substitution lemma (and thus for preservation of β-equivalence).

□

There is no real interest in providing an operator in the base language to intepret stacks of positive type, because they are, as justified before, irrelevant in our current formulation. It is reasonable to believe that this system is, by itself, quite useless. Indeed stacks are lost each time the current term is in evaluation position, that is, when we cross a pattern-matching in call-by-name: the current continuation of a term being evaluated to a value collapses to the unit type.

12.4.5 An attempt at call-by-value

If we try to apply the same principles to an equivalent call-by-value translation, we rapidly hit a wall. There are two ways to try to adapt the translation, and they are all defective in some sense.

The first, sensible, way to extend it is to consider the call-by-value linear decomposition and to see what we recover from it. This is immediate, and we get the following interpretations for the witness and counter types.

$$\begin{array}{lcl} \mathbb{W}([\![A \to B]\!]_{\hat{\mathrm{v}}}) & := & (\mathbb{C}([\![A]\!]_{\hat{\mathrm{v}}}) \to \mathbb{W}([\![A]\!]_{\hat{\mathrm{v}}})) \to \mathbb{C}([\![B]\!]_{\hat{\mathrm{v}}}) \to \mathbb{W}([\![B]\!]_{\hat{\mathrm{v}}}) \\ \mathbb{C}([\![A \to B]\!]_{\hat{\mathrm{v}}}) & := & (\mathbb{C}([\![A]\!]_{\hat{\mathrm{v}}}) \to \mathbb{W}([\![A]\!]_{\hat{\mathrm{v}}})) \times \mathbb{C}([\![B]\!]_{\hat{\mathrm{v}}}) \end{array}$$

Sequents $\Gamma \vdash t : A$ are likewise translated as follows.

$$(\mathbb{C}([\![\Gamma]\!]_{\hat{\mathrm{v}}}) \to \mathbb{W}([\![\Gamma]\!]_{\hat{\mathrm{v}}})), \omega : \mathbb{C}([\![A]\!]_{\hat{\mathrm{v}}}) \vdash t^{\bullet} : \mathbb{W}([\![A]\!]_{\hat{\mathrm{v}}})$$

If we try to formulate the corresponding term translation, what we get is a bit surprising. We give such a translation below.

Definition 166 (Term translation)**.** The tentative call-by-value term translation is defined by induction as follows.

$$\begin{array}{lcl} x^\bullet & := & x\ \omega \\ (\lambda x.\, t)^\bullet & := & \lambda x\, \omega.\, t^\bullet \\ (t\ u)^\bullet & := & (\lambda \omega.\, t^\bullet)\ ((\lambda \omega.\, u^\bullet), \omega)\ (\lambda \omega.\, u^\bullet)\ \omega \end{array}$$

This translation is so close to the call-by-name one that it even starts to smell fishy. There is nothing similar to what our intuition would have told us to expect to find in a call-by-value stack reader translation.

- There is no appearance of negative stacks featuring some form of higher-order terms.
- Stacks are essentially threaded along and never actually built. The application case does make the stack grow, but there is no phenomenon indicating that the terms are forced in sight.
- Values do not correspond to our intuition either. They are waiting for a stack, because they are always banged by the extruding translation. This conflicts with our point of view of values as inert object.
- For all these reasons, the abstract machine suggested by this translation is way too close to the KAM.

We can formulate soundness theorems for this interpretation, but we do not consider it to be worthwhile, for it does not feature the properties we would like to arise in a call-by-value setting.

There is another way to tackle a call-by-value stack reader. Rather than basing ourselves on the linear decomposition, we can try to guess what the call-by-value stacks look like. As hinted by the Dialectica translation, and by duality considerations, call-by-value stacks are functional objects waiting for the current value to return.

If we go this way, the issue of interpretation of positive types arise immediately, and we can simply repeat the impediments described in Section 12.4.4, namely that the only information we get back when observing a stack is some uninformative existential type. So it seems this is a no-go.

Thus both approaches lead to a failure. The problem seems to stem from the fact that we do not enforce sufficient duality in the type translation, so that continuations that happen to be actual functions are bound to be degenerate. To overcome this issue, we need something that looks less intuitionistic.

12.5 From forcing to CPS

We present in this section a translation inspired by the previous one, but trying to fix the defects of the latter. Surprisingly enough, while it is inspired by a mix between Kripke model style translation and stack observation, it turns out it is actually a close variant of the Lafont-Reus-Streicher CPS [72], and that it is a linear simplification of the Kripke translation found in [61] where the Kripke frame of discourse is the algebra of types.

12.5.1 Summary of the issues

The previous stack reader translation suffers from several defects.

- First, on the negative fragment, it is redundant. The translation of the λ-abstraction discards the top of the stack ω

$$(\lambda x.\, t)^\bullet := \lambda x.\, (\lambda \omega.\, t^\bullet)\ (\mathrm{snd}\ \omega)$$

  because it is actually the same as the argument being provided to the function, if we think of it in the KAM.

  This is mirrored in the translation of the application

$$(t\ u)^\bullet := (\lambda \omega.\, t^\bullet)\ ((\lambda \omega.\, u^\bullet), \omega)\ (\lambda \omega.\, u^\bullet)$$

  that duplicates its argument u both as an argument of the function and the top of the stack.

- Second, the translation of positive types is not satisfactory. Stacks of positive types are just erased into the unit type. This is not what we would like to have.

The solution to this problem comes from a careful scrutiny of the type of stacks. Why did we choose to erase positive stacks? Because they hide an existential type that we do not know how to handle well. Instead of trying to bury it under an existential, let us rather expose it it in the type of the stack being constructed. Namely, let us consider a type variable $\perp\!\!\!\perp$ that will act as the final return type of our stack. For instance, the empty stack will naturally be given this type, as

$$\vdash \varepsilon : \perp\!\!\!\perp$$

and likewise, the return type of positive stacks will be set to $\perp\!\!\!\perp$.

We wish to apply a nifty trick though: paralleling the quantification over later worlds at each bang connective in the Kripke models, we are going to quantify over such $\perp\!\!\!\perp$ return type whenever linear logic tells us to do so. Meanwhile, the redundancy of the negative fragment is going to be conflated by a mere polarization argument: negative terms are the ones that may access the current stack, and in particular they actually do not need the argument of the function that they can retrieve on the stack.

The global result is that we recover in both cases a complete model for intuitionistic logic, where the completeness proof performs a kind of internal normalization-by-evaluation. Though not totally unexpected, we believe this is an interesting result. It seems that, even if a lot of close variants exist in the folklore, to the best of our knowledge this particular translation was unknown in the literature. Amongst similar results in the current zeitgeist, see for instance Dagand and Scherer [100].

Definition 167 (Target language)**.** We need a target language featuring second-order quantification. The language we use for the target is the extension of the λ-calculus

with inductive types we have been using for a while together with implicit second-order quantifications, that is, we keep terms as is, and we extend types as

$$A, B := \ldots \mid \forall \alpha.\, A$$

and typing rules with the two following rules.

$$\frac{\Gamma \vdash t : A \qquad \alpha \text{ fresh in } \Gamma}{\Gamma \vdash t : \forall \alpha.\, A} \qquad \frac{\Gamma \vdash t : \forall \alpha.\, A}{\Gamma \vdash t : A[\alpha := T]}$$

The fact second order quantification is transparent allows to keep terms uncluttered from typing details, in practice displaying their proximity to the stack reader translation.

12.5.2 Call-by-name

Definition 168 (Type translation)**.** We define mutually recursively two translations $\mathbb{W}(!A)$ and $\mathbb{C}_{\perp\!\!\!\perp}(A)$, where $\mathbb{C}_{\perp\!\!\!\perp}(A)$ has $\perp\!\!\!\perp$ as a free type variable.

$$\begin{aligned}
\mathbb{W}(!A) &:= \forall \perp\!\!\!\perp.\, \mathbb{C}_{\perp\!\!\!\perp}(A) \to \perp\!\!\!\perp \\
\mathbb{C}_{\perp\!\!\!\perp}(\alpha) &:= \alpha \to \perp\!\!\!\perp \\
\mathbb{C}_{\perp\!\!\!\perp}(A \to B) &:= \mathbb{W}(!A) \times \mathbb{C}_{\perp\!\!\!\perp}(B) \\
\mathbb{C}_{\perp\!\!\!\perp}(1) &:= 1 \to \perp\!\!\!\perp \\
\mathbb{C}_{\perp\!\!\!\perp}(A \times B) &:= \mathbb{W}(!A) \times \mathbb{W}(!B) \to \perp\!\!\!\perp \\
\mathbb{C}_{\perp\!\!\!\perp}(0) &:= 0 \to \perp\!\!\!\perp \\
\mathbb{C}_{\perp\!\!\!\perp}(A + B) &:= \mathbb{W}(!A) + \mathbb{W}(!B) \to \perp\!\!\!\perp
\end{aligned}$$

As one can witness, for the negative fragment, this is a small variation on the call-by-name stack reader, where we changed the interpretation of the bang connective. Instead of

$$\mathbb{W}(!A) := \mathbb{C}(A) \to \mathbb{W}(A)$$

we now have

$$\mathbb{W}(!A) := \forall \perp\!\!\!\perp.\, \mathbb{C}_{\perp\!\!\!\perp}(A) \to \perp\!\!\!\perp$$

with a quantification inspired by Kripke models.

Definition 169 (Term translation)**.** As in the reader case, we assume a reserved variable ω, and we define the translation on terms $(-)^\bullet$ by induction on the term.

$$
\begin{array}{lcl}
x^\bullet & := & x\ \omega \\
(\lambda x.\, t)^\bullet & := & \texttt{match}\ \omega\ \texttt{with}\ (x, \omega) \mapsto t^\bullet \\
(t\ u)^\bullet & := & (\lambda\omega.\, t^\bullet)\ ((\lambda\omega.\, u^\bullet), \omega) \\
()^\bullet & := & \omega\ () \\
(t, u)^\bullet & := & \omega\ ((\lambda\omega.\, t^\bullet), (\lambda\omega.\, u^\bullet)) \\
(\text{inl}\ t)^\bullet & := & \omega\ (\text{inl}\ (\lambda\omega.\, t^\bullet)) \\
(\text{inr}\ u)^\bullet & := & \omega\ (\text{inr}\ (\lambda\omega.\, u^\bullet)) \\
(\texttt{match}\ t\ \texttt{with}\ () \mapsto u)^\bullet & := & (\lambda\omega.\, t^\bullet)\ (\lambda().\, u^\bullet) \\
(\texttt{match}\ t\ \texttt{with}\ (x, y) \mapsto u)^\bullet & := & (\lambda\omega.\, t^\bullet)\ (\lambda(x, y).\, u^\bullet) \\
(\texttt{match}\ t\ \texttt{with}\ [\cdot])^\bullet & := & (\lambda\omega.\, t^\bullet)\ (\lambda[\cdot]) \\
(\texttt{match}\ t\ \texttt{with}\ [x \mapsto u_1 \mid y \mapsto u_2])^\bullet & := & (\lambda\omega.\, t^\bullet)\ (\lambda[x \mapsto {u_1}^\bullet \mid y \mapsto {u_2}^\bullet])
\end{array}
$$

This translation keeps all the nice properties of its close relative, the stack reader translation. We write them down below.

Proposition 112 (Typing soundness)**.** *If* $\Gamma \vdash t : A$ *then for all* $\perp\!\!\!\perp$

$$\mathbb{W}(!\Gamma), \omega : \mathbb{C}_{\perp\!\!\!\perp}(A) \vdash t^\bullet : \perp\!\!\!\perp$$

Proof. By induction on the typing derivation. We detail a bit the interesting details on the second-order operations, for they give good insights on the relation between this translation and the Kripke one.

- Variable case: similarly to call-by-name Kripke models that use reflexivity of the relation on this case, we use elimination of universal quantification.
- λ-abstraction: no modification of the current $\perp\!\!\!\perp$ type (no exponential operation on this rule).
- Application: universal quantification over the argument of the application, similarly to the lift from Kripke models and promotion in linear decomposition.
- Introduction of positives: universal quantification over the subcomponents, for the same reason as application.
- Elimination of positives: same situation as the λ-abstraction.

□

Proposition 113 (Substitution lemma)**.** *For all terms* t *and* r*, we have*

$$(t[x := r])^\bullet \equiv_\beta t^\bullet[x := \lambda\omega.\, r^\bullet]$$

Proof. By induction on t. As in the reader translation, everything goes through owing to the fact that $\lambda\omega.\, r^\bullet$ is closed w.r.t. ω, allowing to make the substitution commute with the surrounding context.

□

Proposition 114 (Computational soundness)**.** *If* $t \equiv_\beta u$ *then* $t^\bullet \equiv_\beta u^\bullet$.

Proof. Almost direct application of the substitution lemma. □

The really interesting fact about this forcing-like CPS comes from the fact it is actually complete for intuitionistic logic.

Theorem 33 (Typing completeness)**.** *For all term t s.t.*

$$\mathbb{W}(!\Gamma) \vdash t : \mathbb{W}(!A)$$

there exists a term t_0 s.t.

$$\Gamma \vdash t_0 : A$$

Actually, we will be proving a more precise formulation of this theorem, which is very similar to normalization-by-evaluation theorems. We define indeed two translations $\downarrow_A^\Gamma(-)$ and $\uparrow_A^\Gamma(-)$ by induction on an environment Γ and a type A. For the sake of simplicity, we identify the source language as a subset of the target language, and we see these translations as acting from and to the target language.

Definition 170 (Reflect-Reify)**.** Assuming an environment Γ and a simple type A, we define the translation $\downarrow_A^\Gamma(-)$ and $\uparrow_A^\Gamma(-)$ by mutual induction on Γ and A.

$$\begin{array}{lcl}
\downarrow_\alpha^\Gamma t & := & [t]_\Gamma^\downarrow\ (\lambda x.\, x) \\
\downarrow_{A\to B}^\Gamma t & := & \lambda x.\, \downarrow_B^{\Gamma,x:A}(\lambda\omega.\, t\ (x,\omega)) \\
\downarrow_1^\Gamma t & := & [t]_\Gamma^\downarrow\ (\lambda().\, ()) \\
\downarrow_{A\times B}^\Gamma t & := & [t]_\Gamma^\downarrow\ (\lambda(x,y).\, (\downarrow_A x, \downarrow_B y)) \\
\downarrow_0^\Gamma t & := & [t]_\Gamma^\downarrow\ (\lambda[\cdot]) \\
\downarrow_{A+B}^\Gamma t & := & [t]_\Gamma^\downarrow\ (\lambda[x \mapsto \mathrm{inl}\ (\downarrow_A x) \mid y \mapsto \mathrm{inr}\ (\downarrow_B y)])
\end{array}$$

$$\begin{array}{lcl}
\uparrow_\alpha^\Gamma t & := & \lambda\omega.\, \omega\ [t]_\Gamma^\uparrow \\
\uparrow_{A\to B}^\Gamma t & := & \lambda(x,\omega).\, (\uparrow_B^{\Gamma,x:A}(t\ x))\ \omega \\
\uparrow_1^\Gamma t & := & \lambda\omega.\, \mathtt{match}\ [t]_\Gamma^\uparrow\ \mathtt{with}\ () \mapsto \omega\ () \\
\uparrow_{A\times B}^\Gamma t & := & \lambda\omega.\, \mathtt{match}\ [t]_\Gamma^\uparrow\ \mathtt{with}\ (x,y) \mapsto \omega\ (\uparrow_A x, \uparrow_B y) \\
\uparrow_0^\Gamma t & := & \lambda\omega.\, \mathtt{match}\ [t]_\Gamma^\uparrow\ \mathtt{with}\ [\cdot] \\
\uparrow_{A+B}^\Gamma t & := & \lambda\omega.\, \mathtt{match}\ [t]_\Gamma^\uparrow\ \mathtt{with}\ [x \mapsto \omega\ (\mathrm{inl}\ (\uparrow_A x)) \mid y \mapsto \omega\ (\mathrm{inr}\ (\uparrow_B y))]
\end{array}$$

$$\begin{array}{lcl}
[t]_\Gamma^\downarrow & := & t[x_i := \uparrow_{\Gamma_i} x_i] \\
[t]_\Gamma^\uparrow & := & t[x_i := \downarrow_{\Gamma_i} x_i]
\end{array}$$

This recursive definition is well-founded, because even if the current environment can grow, it only does so with strict subtypes of the type under focus. This parallels the fact we are actually building a cut-free proof, which therefore enjoys the subformula property. Note that all variables introduced in the above definition are fresh except for the dedicated ω variable.

Proposition 115. *If* $\Gamma, \Delta \vdash t : A$ *then*

$$\mathbb{W}(!\Gamma), \Delta \vdash \uparrow_A^\Gamma t : \mathbb{W}(!A)$$

and dually if $\mathbb{W}(!\Gamma), \Delta \vdash t : \mathbb{W}(!A)$ *then*

$$\Gamma, \Delta \vdash \downarrow_A^\Gamma t : A$$

Proof. By mutual induction on A and Γ. Let us treat in detail the arrow case, which is the most complicated one because it features a modification of the environment.

- Assume $\Gamma, \Delta \vdash t : A \to B$. We must show that

  $$\mathbb{W}(!\Gamma), \Delta \vdash \lambda(x, \omega).\, (\uparrow_B^{\Gamma, x:A}(t\ x))\ \omega : \forall \perp\!\!\!\perp.\, \mathbb{W}(!A) \times \mathbb{C}_{\perp\!\!\!\perp}(B) \to \perp\!\!\!\perp$$

  which amounts to proving that

  $$\mathbb{W}(!\Gamma), x : \mathbb{W}(!A), \Delta \vdash \uparrow_B^{\Gamma, x:A}(t\ x) : \mathbb{W}(!B)$$

  By induction hypothesis on B, it is sufficient to prove

  $$\Gamma, x : A, \Delta \vdash t\ x : B$$

  which is in turn easily derived from the leading assumption because x is not free in t.

- Assume $\mathbb{W}(!\Gamma), \Delta \vdash t : \mathbb{W}(!(A \to B))$. We must show that

  $$\Gamma, \Delta \vdash \lambda x.\, \downarrow_B^{\Gamma, x:A}(\lambda\omega.\, t\ (x, \omega)) : A \to B$$

  which is equivalent to

  $$\Gamma, x : A, \Delta \vdash \downarrow_B^{\Gamma, x:A}(\lambda\omega.\, t\ (x, \omega)) : B$$

  Applying the induction hypothesis on B, we now have to prove

  $$\mathbb{W}(!\Gamma), x : \mathbb{W}(!A), \Delta \vdash \lambda\omega.\, t\ (x, \omega) : \mathbb{W}(!B)$$

  which is once again a straightforward derivation using the starting typing hypothesis on t and the fact that x is not free in t.

□

The completeness theorem is actually an instance of the above proposition, by taking the empty environment, and setting $t_0 := \downarrow_A t$.

Remark 18. Actually, we do not use the full power of impredicative second-order quantification in the completeness theorem. If we were to adapt this translation to a type theory with a hierarchy of types, it seems it would pass through without further modification of the type quantification. The types we use as instances of universal quantifications live at the level of the quantified type indeed. We leave this adaptation to further work.

There are quite a lot of things to say about this CPS. We detail here some points we want to discuss or at least highlight.

First, a comforting theorem that we know to hold: assuming enough typed equational theory (essentially η-expansion on arrows and positives) one can show that the pair $\uparrow_A(-)$ and $\downarrow_A(-)$ form a retraction on typed terms. This is quite reassuring, as it indicates that the translations essentially did nothing to the source term, except for potential supplementary computations. We do not detail this theorem here, because it relies on explicitly typed terms and equations, which would go against the presentation we used throughout this thesis.

The similarity between our completeness theorem and normalization-by-evaluation is striking. This is not surprising, as it is folklore that completeness of Kripke models actually implement a normalization-by-evaluation algorithm. As our CPS is inspired by a linear decomposition of Kripke models, this similarity was somehow expected. There are differences, though. Standard Kripke models (and thus forcing) is call-by-value, while our translation is call-by-name, making it closer to Krivine and Miquel's classical variant of forcing. Moreover, the usual dialectic between a semantic and a syntactic world that is at the root of NBE is totally absent here. Our transformation is a CPS, and as such it is a translation between programming languages.

The translation is also very close to the one proposed by Ilik [61] in Kripke models, although our motivations are very different: we want to provide the basic components allowing to reconstruct the Dialectica translation. Furthermore, we are basing ourselves on linear logic rather that trying to cope with the defects of Kripke models.

Our CPS also has close ties with delimited continuations, as found in [56]. The underlying CPS is the same, namely the Lafont-Reus-Streicher one, but our translation inserts reset operators by quantifying over all possible return types at polarity changes. This is even more obvious in the call-by-value version.

The fact that the CPS is complete for intuitionistic logic may seem a rebuttal, as we usually do program translations to recover additional logical expressive power. Nonetheless, this does not mean that we cannot use locally classical reasoning that would get erased by normalization. In particular, we can still observe the current stack by means of the dedicated ω variable, as in the stack reader translation. In addition, as the translation is a true CPS, it may be possible to reinstate stacks in precise conditions, though we did not study the details of this possibility.

We can nonetheless tweak the CPS to handle a given monad, by interleaving a monadic layer in lieu of the bare $\perp\!\!\!\perp$ type. This effectively turns all $\perp\!\!\!\perp$ right of an arrow into $T \perp\!\!\!\perp$ for some monad T. The translation is essentially similar, except for some additional monadic glue. The resulting translation actually encodes a call-by-name variant of standard direct-stylization of monads as found in Filinski's seminal paper [41].

Finally, the whole path that led us to this translation is rooted in the ideas of classical realizability, and in particular forcing in this context. The notation $\perp\!\!\!\perp$ for the return type is no coincidence. We believe that we can apply back the findings of this CPS into

classical realizability. In particular, we conjecture that changing the falsity interpretation of the arrow to handle the universal quantification from our CPS in the following way

$$\|A \to B\|_{\perp\!\!\!\perp} := \left(\bigcap_{\perp\!\!\!\perp} |A|_{\perp\!\!\!\perp}\right) \cdot \|B\|_{\perp\!\!\!\perp}$$

amounts to enforce an intuitionistic semantic for the arrow, where the $\perp\!\!\!\perp$ index insists on the dependency of the intepretation on the particular pole, and the intersection ranges over upward closed poles.

12.5.3 Call-by-value

We give here the call-by-value, that we get almost for free thanks to the linear decomposition. As usual, because we switched polarities, everything is reversed in the type translation: witness types are defined according to their generating values, and counters are defined as $\perp\!\!\!\perp$-returning functions.

Definition 171 (Type translation)**.** We define mutually recursively two translations $\mathbb{W}(A)$ and $\mathbb{C}_{\perp\!\!\!\perp}(A)$, where $\mathbb{C}_{\perp\!\!\!\perp}(A)$ has $\perp\!\!\!\perp$ as a free type variable.

$$\begin{array}{lcl}
\mathbb{W}(\alpha) & := & \alpha \\
\mathbb{W}(A \to B) & := & \forall \perp\!\!\!\perp. \mathbb{W}(A) \times \mathbb{C}_{\perp\!\!\!\perp}(B) \to \perp\!\!\!\perp \\
\mathbb{W}(1) & := & 1 \\
\mathbb{W}(A \times B) & := & \mathbb{W}(A) \times \mathbb{W}(B) \\
\mathbb{W}(0) & := & 0 \\
\mathbb{W}(A + B) & := & \mathbb{W}(A) + \mathbb{W}(B) \\
\mathbb{C}_{\perp\!\!\!\perp}(A) & := & \mathbb{W}(A) \to \perp\!\!\!\perp
\end{array}$$

This type translation is much more standard for a call-by-value CPS than our call-by-name CPS. Usually, one uncurries the product type in the witness translation of the arrow, but this hides the fact this translation comes from a linear decomposition.

As one can observe, call-by-value stacks are functional objects, a fact which corresponds with the usual intuition that identifies call-by-value continuations with functions.

Definition 172 (Term translation)**.** We define the translation on terms $(-)^\bullet$ by induction on the considered term, with the usual special variable ω. We use the notation $t^\#$ for $(\lambda\omega.\, t^\bullet)\ (\lambda x.\, x)$.

$$
\begin{array}{lcl}
x^\bullet & := & \omega\ x \\
(\lambda x.\, t)^\bullet & := & \omega\ (\lambda(x,\omega).\, t^\bullet) \\
(t\ u)^\bullet & := & (\lambda\omega.\, t^\bullet)\ (\lambda f.\, f\ (u^\#, \omega)) \\
()^\bullet & := & \omega\ () \\
(t,u)^\bullet & := & \omega\ (t^\#, u^\#) \\
(\mathrm{inl}\ t)^\bullet & := & \omega\ (\mathrm{inl}\ t^\#) \\
(\mathrm{inr}\ u)^\bullet & := & \omega\ (\mathrm{inr}\ u^\#) \\
(\mathtt{match}\ t\ \mathtt{with}\ () \mapsto u)^\bullet & := & (\lambda\omega.\, t^\bullet)\ (\lambda().\, u^\bullet) \\
(\mathtt{match}\ t\ \mathtt{with}\ (x,y) \mapsto u)^\bullet & := & (\lambda\omega.\, t^\bullet)\ (\lambda(x,y).\, u^\bullet) \\
(\mathtt{match}\ t\ \mathtt{with}\ [\cdot])^\bullet & := & (\lambda\omega.\, t^\bullet)\ (\lambda[\cdot]) \\
(\mathtt{match}\ t\ \mathtt{with}\ [x \mapsto u_1 \mid y \mapsto u_2])^\bullet & := & (\lambda\omega.\, t^\bullet)\ (\lambda[x \mapsto u_1{}^\bullet \mid y \mapsto u_2{}^\bullet])
\end{array}
$$

This CPS has a very special flavour, up to the point where one could deny the fact that this is a CPS. Indeed, the translation does not take position for any order of reductions. This is obvious when one looks at the translation of the pair, which is totally symmetrical. This is in turn made possible thanks to the translation $(-)^\#$ that acts as reset operator from the world of delimited translations, hence the notation.

Proposition 116 (Typing soundness)**.** *If* $\Gamma \vdash t : A$ *then for all* $\perp\!\!\!\perp$

$$\mathbb{W}(\Gamma), \omega : \mathbb{C}_{\perp\!\!\!\perp}(A) \vdash t^\bullet : \perp\!\!\!\perp$$

Proof. By induction on the typing derivation. □

A completeness theorem similar to the call-by-name case can be proved in the very same way. We do not give the details here for they would give no particular hindsight into the translation. Likewise, computational soundness follows immediately, but we do not think that spelling it out will help the understanding of the principles we want to highlight. The call-by-name case by itself already exposed much of the structure we wanted to present, so that we will not pursue into this direction.

12.6 Towards Dialectica

We now turn back to the Dialectica translation, and look at it with the new knowledge we acquired from the previous sections. We will compare it in particular with the call-by-name CPS presented just above.

The two translations share indeed a striking similarity. The reverse translation from the Dialectica interpretation and the CPS itself have the same encoding for stacks in the negative fragment, and a very similar one for positive types. This is better witnessed by recalling the counter translations below.

	Dialectica	Lafont-Reus-Streicher
$A \to B$	$\mathbb{W}(A) \times \mathbb{C}(B)$	$\mathbb{W}(A) \times \mathbb{C}_{\perp\!\!\!\perp}(B)$
$A \times B$	$\mathbb{W}(A) \times \mathbb{W}(B) \to \begin{cases} \mathfrak{M}\,\mathbb{C}(A) \\ \mathfrak{M}\,\mathbb{C}(B) \end{cases}$	$\mathbb{W}(A) \times \mathbb{W}(B) \to \perp\!\!\!\perp$

This kinship is undoubtedly due to the fact that those two translations factor through a linear decomposition. It seems that the Dialectica translation has something which makes it more general that the simple CPS, though. Indeed, while the CPS only cares about the return type of the expression, the Dialectica translation also manages the manipulation of the environment of the term through the reverse term translations.

This hints at the fact that one should be able to retrieve a variant of the Dialectica translation by carefully choosing a monad whose exact type depends on the environment of the term to compose with the CPS. Typically, one would want to translate a sequent $\Gamma \vdash t : A$ into

$$\mathbb{W}(!\Gamma) \vdash t^\bullet : \mathbb{C}_\perp(A) \to \perp\!\!\!\perp \times \mathfrak{M}\,\mathbb{C}_\perp(\Gamma_1) \times \ldots \times \mathfrak{M}\,\mathbb{C}_\perp(\Gamma_n)$$

so that the forward and reverse translation would be conflated into a unique translation. The implicit universal quantification over $\perp\!\!\!\perp$ would allow to retrieve the forward translation by an astute choice of return type, and each reverse translation would be reduced to a projection of this term. The problem with this idea is that it requires to heavily modify the interpretation of the types, and there is no particular instantiation that seems to work. Updating from the CPS

$$\begin{array}{lcl} \mathbb{W}(A \to B) & := & \forall \perp\!\!\!\perp.\, \mathbb{W}(A) \times \mathbb{C}_\perp(B) \to \perp\!\!\!\perp \\ \mathbb{C}_\perp(A \to B) & := & \mathbb{W}(A) \times \mathbb{C}_\perp(B) \end{array}$$

into a tentative reconstructed Dialectica

$$\begin{array}{lcl} \mathbb{W}(A \to B) & := & \forall \perp\!\!\!\perp.\, \mathbb{W}(A) \times \mathbb{C}_\perp(B) \to \perp\!\!\!\perp \times \mathfrak{M}\,\mathbb{C}_\perp(A) \\ \mathbb{C}_\perp(A \to B) & := & \mathbb{W}(A) \times \mathbb{C}_\perp(B) \end{array}$$

does not allow to interpret the application rule in a satisfactory way. Indeed, considering the translated terms

$$\mathbb{W}(!\Gamma) \vdash t^\bullet : \mathbb{W}(A) \times \mathbb{C}_\perp(B) \to \perp\!\!\!\perp \times \mathfrak{M}\,\mathbb{C}_\perp(\Gamma_1) \times \ldots \times \mathfrak{M}\,\mathbb{C}_\perp(\Gamma_n)$$

$$\mathbb{W}(!\Gamma) \vdash u^\bullet : \mathbb{C}_\perp(A) \to \perp\!\!\!\perp \times \mathfrak{M}\,\mathbb{C}_\perp(\Gamma_1) \times \ldots \times \mathfrak{M}\,\mathbb{C}_\perp(\Gamma_n)$$

one can see that there is no way to merge the stacks coming from t and the ones from u, not only because their return type $\perp\!\!\!\perp$ differs (u is fed to t with a universally quantified $\perp\!\!\!\perp$) but also simply because t loses the stacks produced by u. There is no *a priori* reason for the two terms to depend on the same variables.

The KAM intuition tells us that the $\mathbb{W}(A)$ should be interpreted not as closed terms, but rather as closures, so that it should take an additional parameter Γ indicating the type of the free variables this term depends on. A likely translation for the arrow would then be

$$\begin{array}{lcl} \mathbb{W}_\Gamma(A \to B) & := & \forall \perp\!\!\!\perp.\, \forall \Delta.\, \mathbb{W}_\Delta(A) \times \mathbb{C}_\perp(B) \to \perp\!\!\!\perp \times \mathfrak{M}\,\mathbb{C}_\perp(\Delta) \times \mathfrak{M}\,\mathbb{C}_\perp(\Gamma) \\ \mathbb{C}_\perp(A \to B) & := & (\exists \Delta.\, \mathbb{W}_\Delta(A)) \times \mathbb{C}_\perp(B) \end{array}$$

where Γ and Δ range over a given notion of first-class environments. Although seemingly sensible, we do not see any way to make this particular translation work, and in particular

how to encode environments as first-class objects in the target. This is why it seems difficult to synthesize a proper Dialectica-like translation from our stack-reading CPS, even though the two display a close relationship.

In any case, the Dialectica translation features a special handling of the environment which is absent from the CPS. We conjecture that the linear logic exponential is the connective which is the root of this particular handling, as witnessed by the $\mathbb{W}(\llbracket A \to B \rrbracket_{\mathrm{n}})$ and $\mathbb{C}(\llbracket A \times B \rrbracket_{\mathrm{n}})$ types, whose translation make explicit the appearance of the environment. Such a property is quite uncommon in the translations from the literature, which tend to care only about the return type of the object. In addition, in the proposed environment-caring translations proposed above, there seems to be a natural duality in call-by-name: while counters are parameterized by a return type, witnesses are themselves parameterized by the closure they depend on. There is probably a deep connection with the KAM environments hidden in plain sight there.

There is still a little remark to be made. Building upon the CPS translation, it is easy to observe that the Dialectica translation can be adapted to handle naturally the quantification over the $\bot\!\!\!\bot$ type. In the presentation we gave, this type is fixed as in the Lafont-Reus-Streicher CPS. As seen in the interpretation of $\mathbb{C}(1)$ in Section 10.1, the particular return type of the Dialectica translation is *always* set to the unit type 1. There is no reason for such an arbitrary choice, and the translation can perfectly accommodate a variable return type. It is sufficient indeed to tweak the type translations of the arrow as follows

$$\begin{array}{lcl} \mathbb{W}(A \to B) & := & (\mathbb{W}(A) \to \mathbb{W}(B)) \times (\mathbb{W}(A) \to \forall \bot\!\!\!\bot.\, \mathbb{C}_{\bot\!\!\!\bot}(B) \to \mathfrak{M}\, \mathbb{C}_{\bot\!\!\!\bot}(A)) \\ \mathbb{C}_{\bot\!\!\!\bot}(A \to B) & := & \mathbb{W}(A) \times \mathbb{C}_{\bot\!\!\!\bot}(B) \end{array}$$

to recover a kind of delimited Dialectica. The term translation is the same, because the second-order quantifications and eliminations are transparent. This suggests that call-by-name delimited continuations as described by Herbelin and Ghilezan [56] naturally fit in the Dialectica framework. Note that this may interfere with the interpretation of Markov's principle, which relies on a particular choice of $\bot\!\!\!\bot$ to work. We leave this remark for future work.

As for the completeness of the variant of the Lafont-Reus-Streicher CPS w.r.t. intuitionistic provability, we conjecture that a small variant of our Dialectica could be complete for linear logic provability. It is indeed a phenomenological observation that many of the the instances of models of linear logic from the literature can be seen as a special case of the double-glueing construction, itself being ultimately a generalization of the Dialectica translation in a categorical setting [58]. Such a Dialectica variant could be probably designed as a free model of linear logic featuring its rawest ingredients, namely a commutative monad equipped with a built-in notion of duality. It seems that the Dialectica we have at hand already provides those basic blocks, so that such a model may not be that far away.

13 Conclusion

> (An unmatched left parenthesis creates an unresolved tension that will stay with you all day.
>
> *Randall Munroe about the scientific process.*

Looking at a mathematical object with a modern eye can often bring out a brand new way to think of it. This is quite remarkably the case for the Dialectica translation: had it not been conceived half a century before Krivine realizability, one could probably think that it had been influenced by the latter. It is quite fascinating to witness that this translation can be reworded seamlessly as an untyped program translation using precise concepts such as closures and stacks.

Revealing that the Dialecticta translation could be considered as a translation pertaining to the classical realizability realm is the core result of this thesis, although this statement actually covers quite a few distinct achievements. To start with, observing that the realizers given by the historical translation were broken w.r.t. their operational behaviour was the first step into a deeper glance at their actual computational content. We are still puzzled by the fact that this syntactical misdemeanour was never recognized as such. The second step consisted in realizing that the Diller-Nahm variant almost solved this issue, even though once again nobody ever advertized this fact, because it had been built for totally unrelated motivations.

Once the operational issue was solved, everything followed naturally. The KAM simulation is straightforward as soon as one thinks of counters as stacks, which is an immediate reflex for anyone acquainted with classical realizability. The design of a dependent version is also direct, owing to the fact that the Dialectica translation is essentially intuitionistic. Moreover, the variants of the Dialectica are provided for free, thanks to the various linear decompositions we know of.

All these small results put side by side give a broader understanding of the Dialectica as a program translation taking stacks to be a serious matter. Even more interestingly, its careful study shows that there is still interest to look at it from the Curry-Howard point of view. We shortly summarize here the questions that arise from our modern description.

- Why does the Dialectica ignore sequentiality while still sticking to a particular calling convention? This seems to be a property stemming from linear logic itself, which is somehow the finest way to endow a calculus with a particular semantic without committing to a given order of reduction. It should be interesting to look at other commutative effects to see if there is such a simple way to encode them

syntaxically through linear logic. The opposite issue, namely to sequentialize the Dialectica translation, is not devoid of interest either.

- What is the general way to describe environment-aware translations? It looks like there are a few such instances in the literature, but they are made in a mostly ad-hoc way. In particular, call-by-need is a contrived example of semantics relying heavily on the ability to manipulate the environment as a first-class object [13]. A deep scrutiny of the way the Dialectica handles this as well as a clarification of its relation to the KAM should provide hindsights to design call-by-need reductions based on logical principles.

- Why do CPS-like translations, including Dialectica, fail at allowing dependent elimination? Even when they feature a mostly intuitionistic content, the dependency cannot pass through because of subtle typing mismatches. A naive way, that appears to be really difficult to implement properly, is to make the (implicit or not) return type of the translation depend on the object being built when it is positive. This suggests systems based on sequent calculus rather than natural deduction. Although there are some attemps to do so, they remain essentially rudimentary.

- What is the exact relationship between delimited control and linear logic? As shown in the last chapter, there is a broad continuum of translations that revolve around Dialectica. Some of them can be easily equipped with delimited control. As the modified CPS we obtained is complete for intuitionistic provability thanks to a very coarse use of delimitation, we believe that a fine-grained Dialectica featuring first-class delimited continuations may turn out to be complete for linear provability. There is a potential connection with the famous encoding due to Filinski [41], but restricted to commutative monads rather than arbitrary ones. A more recent work due to Munch [84, 85] seems to indicate that linearity can indeed be described by a semantic property deeply related to delimited contexts. Providing an answer to the above question may appear really difficult, but the hindsights given by our various decompositions should help to progress in that direction.

All these questions, albeit unanswered, have been clarified by the present thesis, as it gave us a handful of starting hints waiting to be analyzed and hopefully exploited to provide answers to them. We would be glad if this thesis were to sprout a renewed interest in research directions that have been given up for some time, as well as giving brand new areas to explore. We shall mention the computational interpretation of linear logic for the first point, and more intuitionistic variants of Krivine's realizability for the second one. In any case, science progresses by leaps. Let us just be patient.

Bibliography

[1] Andreas Abel. *Normalization by Evaluation: Dependent Types and Impredicativity.* Habilitation Thesis. May 2013.

[2] Yehoshua Bar-Hillel Abraham A. Fraenkel and Azriel Levy. "Foundations of Set Theory Second Revised Edition". In: vol. 67. Studies in Logic and the Foundations of Mathematics. Elsevier, 1973. DOI: http://dx.doi.org/10.1016/S0049-237X(08)70329-X.

[3] Beniamino Accattoli. "An abstract factorization theorem for explicit substitutions". In: *23rd International Conference on Rewriting Techniques and Applications (RTA'12).* Nagoya, Japan, May 2012.

[4] Beniamino Accattoli, Pablo Barenbaum, and Damiano Mazza. "Distilling Abstract Machines (Long Version)". In: *CoRR* abs/1406.2370 (2014).

[5] Beniamino Accattoli and Delia Kesner. "The permutative lambda calculus". In: *18th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning - LPAR-18.* Merida, Venezuela, Mar. 2012.

[6] Beniamino Accattoli and Delia Kesner. "The Structural λ-Calculus". In: *CSL.* 2010, pp. 381–395.

[7] Beniamino Accattoli and Ugo Dal Lago. "Beta Reduction is Invariant, Indeed". In: *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).* Vienna, Austria, July 2014. DOI: 10.1145/2603088.2603105.

[8] Beniamino Accattoli and Ugo Dal Lago. "On the Invariance of the Unitary Cost Model for Head Reduction (Long Version)". In: *CoRR* abs/1202.1641 (2012).

[9] Beniamino Accattoli and Luca Paolini. "Call-by-Value solvability, revisited". In: *11th International Symposium on Functional and Logic Programming - FLOPS 2012.* Kobe, Japan, May 2012.

[10] Beniamino Accattoli et al. "A nonstandard standardization theorem". In: *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014.* 2014, pp. 659–670. DOI: 10.1145/2535838.2535886.

[11] Zena M. Ariola and Matthias Felleisen. "The call-by-need lambda calculus". In: *Journal of Functional Programming* 7.3 (1997), pp. 265–301. ISSN: 0956-7968. DOI: http://dx.doi.org/10.1017/S0956796897002724.

[12] Zena M. Ariola, Hugo Herbelin, and Alexis Saurin. "Classical Call-by-need and duality". In: *Typed Lambda Calculi and Applications*. Vol. 6690. Lecture Notes in Computer Science. Springer, 2011.

[13] Zena M. Ariola et al. "Classical call-by-need sequent calculi: The unity of semantic artifacts". In: *Functional and Logic Programming - 11th International Symposium, FLOPS 2012, Kobe, Japan, May 23-25, 2012. Proceedings*. Ed. by Tom Schrijvers and Peter Thiemann. Vol. 7294. Lecture Notes in Computer Science. Springer, 2012, pp. 32–46. ISBN: 978-3-642-29821-9.

[14] Zena M. Ariola et al. "The Call-by-Need Lambda Calculus". In: *Symposium on Principles of Programming Languages, POPL 1995*. Ed. by Ron K. Cytron and Peter Lee. ACM Press, 1995, pp. 233–246. ISBN: 0-89791-692-1.

[15] Jeremy Avigad and Solomon Feferman. "Gödel's Functional ('Dialectica') Interpretation". In: *Handbook of Proof Theory*. Ed. by Samuel R. Buss. Amsterdam: Elsevier Science Publishers, 1998, pp. 337–405.

[16] Andrew Barber. *Dual Intuitionistic Linear Logic*. Tech. rep. University of Edinburgh, 1996. URL: `http://www.lfcs.inf.ed.ac.uk/reports/96/ECS-LFCS-96-347/index.html`.

[17] H. P. Barendregt. *The Lambda Calculus Its Syntax and Semantics*. Revised. Vol. 103. North Holland, 1984.

[18] Henk Barendregt et al. "Lambda Calculi with Types". In: *Handbook of Logic in Computer Science*. Oxford University Press, 1992, pp. 117–309.

[19] N. Benton and P. Wadler. "Linear logic, monads and the lambda calculus". In: *Proceedings of the 11th IEEE Symposium on Logic in Computer Science, Brunswick, New Jersey*. IEEE Press, July 1996.

[20] U. Berger and H. Schwichtenberg. "An inverse of the evaluation functional for typed lambda -calculus". In: *[1991] Proceedings Sixth Annual IEEE Symposium on Logic in Computer Science*. Amsterdam, Netherlands: IEEE Comput. Sco. Press, July 18, 1991, pp. 203–211. ISBN: 0-8186-2230-X. DOI: `10.1109/lics.1991.151645`.

[21] Bodil Biering. "Dialectica Interpretations: A Categorical Analysis". PhD thesis. IT University, 2008.

[22] George Boole. *Investigation of The Laws of Thought On Which Are Founded the Mathematical Theories of Logic and Probabilities*. 1853.

[23] Breuvart and Pagani. "Relational semantics for bounded calculus". Draft. 2015.

[24] Alois Brunel. "The Monitoring Power of Forcing Transformation". PhD thesis. Univ. Paris Nord, 2014.

[25] Aloïs Brunel et al. "A Core Quantitative Coeffect Calculus". In: *Proceedings of ESOP*. Ed. by Z. Shao. Vol. 8410. Lecture Notes in Computer Science. Springer, 2014, pp. 351–370.

[26] Georg Cantor. "Ueber unendliche, lineare Punktmannichfaltigkeiten". In: *Mathematische Annalen* 15.1 (1879), pp. 1–7. ISSN: 0025-5831. DOI: 10.1007/BF01444101.

[27] Stephen Chang and Matthias Felleisen. "The Call-by-need Lambda Calculus, Revisited". In: *European Symposium on Programming, ESOP 2012*. Lecture Notes in Computer Science. Springer, 2012.

[28] Adam Chlipala. *Certified Programming with Dependent Types*. http://adam.chlipala.net/cpdt/. MIT Press, 2011. URL: %7Bhttp://adam.chlipala.net/cpdt/%7D.

[29] A. Church. "A Set of Postulates for the Foundation of Logic". In: *Annals of Mathematics* 33.2 (1932), pp. 346–366.

[30] P.J. Cohen. *Set theory and the continuum hypothesis*. Mathematics lecture note series. W. A. Benjamin, 1966.

[31] H.P. Cooke and H. Tredennick. *Aristotle: The Organon*. Aristotle: The Organon v. 2. Harvard University Press.

[32] Thierry Coquand. "An Analysis of Girard's Paradox". In: *LICS*. IEEE Computer Society, 1986, pp. 227–236.

[33] Thierry Coquand and Gerard Huet. "The Calculus of Constructions". In: *Inf. Comput.* 76.2-3 (Feb. 1988), pp. 95–120. ISSN: 0890-5401. DOI: 10.1016/0890-5401(88)90005-3. URL: http://dx.doi.org/10.1016/0890-5401(88)90005-3.

[34] Roberto Di Cosmo. *The Linear Logic Primer*. 1992.

[35] Pierre-Louis Curien and Hugo Herbelin. "The duality of computation". In: *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000*. 2000, pp. 233–243. DOI: 10.1145/351240.351262. URL: http://doi.acm.org/10.1145/351240.351262.

[36] Vincent Danos, Hugo Herbelin, and Laurent Regnier. "Game Semantics & Abstract Machines". In: *Logic in Computer Science, LICS*. 1996, pp. 394–405.

[37] Vincent Danos and Laurent Regnier. "Head Linear Reduction". Unpublished. 2004.

[38] Olivier Danvy and Kevin Millikin. "A Rational Deconstruction of Landin's SECD Machine with the J Operator". In: *Logical Methods in Computer Science* 4.4 (2008). DOI: 10.2168/LMCS-4(4:12)2008. URL: http://dx.doi.org/10.2168/LMCS-4(4:12)2008.

[39] Olivier Danvy et al. "Defunctionalized Interpreters for Call-by-need evaluation". In: *Functional and Logic Programming Symposium, FLOPS 2010*. Lecture Notes in Computer Science. Springer, 2010.

[40] Justus Diller. "Eine Variante zur Dialectica-Interpretation der Heyting-Arithmetik endlicher Typen". German. In: *Archiv für mathematische Logik und Grundlagenforschung* 16.1-2 (1974), pp. 49–66. ISSN: 0003-9268. DOI: 10.1007/BF02025118. URL: http://dx.doi.org/10.1007/BF02025118.

[41] Andrzej Filinski. "Representing Monads". In: *Proceedings of the Twenty-First Annual ACM Symposium on Principles of Programming Languages*. ACM Press, 1994, pp. 446–457.

[42] Gottlob Frege. *Begriffsschrift, eine der arithmetischen nachgebildete Formelsprache des reinenDenkens*. Halle: Verlag von Louis Nebert, 1879.

[43] Jean-Yves Girard. "A New Constructive Logic: Classical Logic". In: *Mathematical Structures in Computer Science* 1.3 (1991), pp. 255–296. DOI: 10.1017/S0960129500001328. URL: http://dx.doi.org/10.1017/S0960129500001328.

[44] Jean-Yves Girard. "Linear Logic". In: *Theor. Comput. Sci.* 50 (1987), pp. 1–102.

[45] Jean-Yves Girard. "Linear Logic: Its Syntax and Semantics". In: *Proceedings of the Workshop on Advances in Linear Logic*. New York, NY, USA: Cambridge University Press, 1995, pp. 1–42. ISBN: 0-521-55961-8.

[46] Jean-Yves Girard. *The Blind Spot: Lectures on Logic*. European Mathematical Society, 2011.

[47] Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and Types*. New York, NY, USA: Cambridge University Press, 1989. ISBN: 0-521-37181-3.

[48] J.Y. Girard. "Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur". PhD thesis. 1972.

[49] V. Glivenko. "Sur Quelques Points de la Logique de M. Brouwer". In: *Bulletins de la classe des sciences*. 5th ser. 15 (1929), pp. 183–188.

[50] K. Gödel. "Zur intuitionistischen Arithmetik und Zahlentheorie". In: *Ergebnisse eines mathematisches Kolloquiums* 4 (1932), pp. 34–38.

[51] Kurt Gödel. "Über eine bisher noch nicht benützte Erweiterung des finiten Standpunktes". In: *Dialectica* 12 (1958), pp. 280–287.

[52] Kurt Gödel. "Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme". In: *Monatshefte für Mathematik und Physik* 38.1 (1931), pp. 173–198.

[53] Timothy G. Griffin. "A Formulae-as-type Notion of Control". In: POPL '90. San Francisco, California, USA: ACM, 1990, pp. 47–58. ISBN: 0-89791-343-4.

[54] Timothy G. Griffin. *Remarks on A Formulae-as-Types Notion of Control*. 2013.

[55] Hugo Herbelin. "An Intuitionistic Logic that Proves Markov's Principle". In: *Logic in Computer Science, Symposium on* (2010), pp. 50–56. ISSN: 1043-6871. DOI: http://doi.ieeecomputersociety.org/10.1109/LICS.2010.49.

[56] Hugo Herbelin and Silvia Ghilezan. "An Approach to Call-by-Name Delimited Continuations". In: *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*. Ed. by George C. Necula and Philip Wadler. ACM, Jan. 2008, pp. 383–394. ISBN: 978-1-59593-689-9.

[57] William A. Howard. "The formulas-as-types notion of construction". In: *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*. Ed. by J. P. Seldin and J. R. Hindley. Academic Press, 1980, pp. 479–490.

[58] J. M. E. Hyland. "Proof theory in the abstract". In: *Ann. Pure Appl. Logic* 114.1-3 (2002), pp. 43–78.

[59] Martin Hyland and Luke Ong. "On Full Abstraction for PCF". In: *Information and Computation* 163.2 (Dec. 2000), pp. 285–408.

[60] Martin Hyland and Andrea Schalk. "Glueing and orthogonality for models of linear logic". In: *Theor. Comput. Sci.* 294.1/2 (2003), pp. 183–231.

[61] Danko Ilik. "Continuation-passing style models complete for intuitionistic logic". In: *Annals of Pure and Applied Logic* 164.6 (2013), pp. 651–662. ISSN: 0168-0072. DOI: http://dx.doi.org/10.1016/j.apal.2012.05.003.

[62] Danko Ilik, Gyesik Lee, and Hugo Herbelin. "Kripke Models for Classical Logic". In: *Annals of Pure and Applied Logic* 161.11 (2010). Ed. by Steffen van Bakel, Stefano Berardi, and Ulrich Berger. Special Issue on Classical Logic and Computation, pp. 1367–1378.

[63] Guilhem Jaber, Nicolas Tabareau, and Matthieu Sozeau. "Extending Type Theory with Forcing". In: *LICS 2012 : Logic In Computer Science*. Dubrovnik, Croatia, June 2012, pp. –. URL: https://hal.archives-ouvertes.fr/hal-00685150.

[64] Jean-Yves Girard. "Proof-nets: The parallel syntax for proof-theory". In: *Logic and Algebra*. Marcel Dekker, 1996, pp. 97–124.

[65] S. C. Kleene. "On the interpretation of intuitionistic number theory". In: *The Journal of Symbolic Logic* 10 (04 Dec. 1945), pp. 109–124. ISSN: 1943-5886.

[66] Ulrich Kohlenbach. *Applied Proof Theory: Proof Interpretations and their Use in Mathematics*. Springer Monographs in Mathematics. Springer Verlag, 2008.

[67] Saul Kripke. "A Completeness Theorem in Modal Logic". In: *J. Symb. Log.* 24.1 (1959), pp. 1–14. DOI: 10.2307/2964568. URL: http://dx.doi.org/10.2307/2964568.

[68] Jean-Louis Krivine. "A call-by-name lambda-calculus machine". In: *Higher-Order and Symbolic Computation* 20.3 (2007), pp. 199–207.

[69] Jean-Louis Krivine. "Classical Logic, Storage Operators and Second-Order lambda-Calculus". In: *Ann. Pure Appl. Logic* 68.1 (1994), pp. 53–78. DOI: 10.1016/0168-0072(94)90047-7. URL: http://dx.doi.org/10.1016/0168-0072(94)90047-7.

[70] Jean-Louis Krivine. "Dependent choice, 'quote' and the clock". In: *Theor. Comput. Sci.* 308.1-3 (2003), pp. 259–276.

[71] Jean-Louis Krivine. "Realizability algebras: a program to well order R". In: *Logical Methods in Computer Science* 7.3 (2011). DOI: 10.2168/LMCS-7(3:2)2011. URL: http://dx.doi.org/10.2168/LMCS-7(3:2)2011.

[72] Y. Lafont, B. Reus, and T. Streicher. *Continuations Semantics or Expressing Implication by Negation*. Technical Report 9321. Ludwig-Maximilians-Universitat, Munchen, 1993.

[73] J. Laird et al. "Weighted relational models of typed lambda-calculi". In: *28th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS 2013), 25-28 June 2013, New Orleans, USA, Proceedings*. 2013, pp. 301–310.

[74] Olivier Laurent. "A study of polarization in logic". PhD thesis. Université de la Méditerranée - Aix-Marseille II, Mar. 2002.

[75] Xavier Leroy. *The ZINC experiment: an economical implementation of the ML language*. Technical report 117. INRIA, 1990.

[76] Paul Blain Levy. "Call-by-push-value". PhD thesis. Queen Mary, University of London, 2001.

[77] Zhaohui Luo. "ECC, an Extended Calculus of Constructions". In: *LICS*. IEEE Computer Society, 1989, pp. 386–395.

[78] John Maraist, Martin Odersky, and Philip Wadler. "The Call-by-Need lambda-Calculus". In: *Journal of Functional Programming* 8.3 (1998), pp. 275–317.

[79] Gianfranco Mascari and Marco Pedicini. "Head linear reduction and pure proof net extraction". In: *Theoret. Comput. Sci.* 135.1 (1994), pp. 111–137.

[80] Paul-andré Melliès. "Categorical semantics of linear logic". In: *Interactive Models of Computation and Program Behaviour, Panoramas et Synthèses 27, Société Mathématique de France 1–196*. 2009.

[81] Alexandre Miquel. "Forcing as a Program Transformation". In: *LICS*. IEEE Computer Society, 2011, pp. 197–206. ISBN: 978-0-7695-4412-0.

[82] Alexandre Miquel. "Relating Classical Realizability and Negative Translation for Existential Witness Extraction". In: *Typed Lambda Calculi and Applications, 9th International Conference, TLCA 2009, Brasilia, Brazil, July 1-3, 2009. Proceedings*. 2009, pp. 188–202. DOI: 10.1007/978-3-642-02273-9_15. URL: http://dx.doi.org/10.1007/978-3-642-02273-9_15.

[83] Eugenio Moggi. "Notions of Computation and Monads". In: *Inf. Comput.* 93.1 (1991), pp. 55–92.

[84] Guillaume Munch-Maccagnoni. "Focalisation and Classical Realisability". In: *Computer Science Logic, 23rd international Workshop, CSL 2009, 18th Annual Conference of the EACSL, Coimbra, Portugal, September 7-11, 2009. Proceedings*. 2009, pp. 409–423. DOI: 10.1007/978-3-642-04027-6_30. URL: http://dx.doi.org/10.1007/978-3-642-04027-6_30.

[85] Guillaume Munch-Maccagnoni. "Formulae-as-Types for an Involutive Negation". In: *Proceedings of the joint meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (CSL-LICS)*. 2014.

[86] Guillaume Munch-Maccagnoni. *Lambda-calcul, machines et orthogonalité*. Unpublished. 2011.

[87] Guillaume Munch-Maccagnoni. "Syntax and Models of a non-Associative Composition of Programs and Proofs". PhD thesis. Univ. Paris Diderot, Dec. 2013.

[88] Mitsuhiro Okada. "Phase semantic cut-elimination and normalization proofs of first- and higher-order linear logic". In: *Theoretical Computer Science* 227.1-2 (1999), pp. 333–396. ISSN: 0304-3975. DOI: http://dx.doi.org/10.1016/S0304-3975(99)00058-4.

[89] Paulo Oliva. "Unifying Functional Interpretations". In: *Notre Dame Journal of Formal Logic* 47.2 (Apr. 2006), pp. 263–290. DOI: 10.1305/ndjfl/1153858651. URL: http://dx.doi.org/10.1305/ndjfl/1153858651.

[90] Paulo Oliva and Thomas Streicher. "On Krivine's Realizability Interpretation of Classical Second-Order Arithmetic". In: *Fundam. Inform.* 84.2 (2008), pp. 207–220. URL: http://iospress.metapress.com/content/f51774wm73404583/.

[91] Nicolas Oury. "Extensionality in the Calculus of Constructions". English. In: *Theorem Proving in Higher Order Logics*. Ed. by Joe Hurd and Tom Melham. Vol. 3603. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2005, pp. 278–293. ISBN: 978-3-540-28372-0. DOI: 10.1007/11541868_18. URL: http://dx.doi.org/10.1007/11541868_18.

[92] Valeria de Paiva. "A Dialectica-like Model of Linear Logic". In: *Category Theory and Computer Science*. Ed. by David H. Pitt et al. Vol. 389. Lecture Notes in Computer Science. Springer, 1989, pp. 341–356.

[93] Valeria de Paiva. "The Dialectica Categories". In: *Categories in Computer Science and Logic: Proc. of the Joint Summer Research Conference*. Ed. by J. W. Gray and A. Scedrov. Providence, RI: American Mathematical Society, 1989, pp. 47–62.

[94] Michel Parigot. "Lambda-Mu-Calculus: An algorithmic interpretation of classical natural deduction". English. In: *Logic Programming and Automated Reasoning*. Ed. by Andrei Voronkov. Vol. 624. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1992, pp. 190–201. ISBN: 978-3-540-55727-2. DOI: 10.1007/BFb0013061. URL: http://dx.doi.org/10.1007/BFb0013061.

[95] Pierre-Marie Pédrot. "A Functional Functional Interpretation". 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)*. CSL-LICS '14. Vienna, Austria: ACM, 2014, 77:1–77:10. ISBN: 978-1-4503-2886-9. DOI: 10.1145/2603088.2603094. URL: http://doi.acm.org/10.1145/2603088.2603094.

[96] Laurent Regnier. "Lambda-calcul et réseaux". PhD thesis. Univ. Paris VII, 1992.

[97] Laurent Regnier. "Une équivalence sur les lambda-termes". In: *Theoretical Computer Science* 126 (1994), pp. 281–292.

[98] John C. Reynolds. "Towards a theory of type structure". In: *Programming Symposium, Proceedings Colloque sur la Programmation, Paris, France, April 9-11, 1974*. 1974, pp. 408–423. DOI: 10.1007/3-540-06859-7_148.

[99] Bertrand Russell. *The Principles of Mathematics*. 1903.

[100] Gabriel Scherer and Pierre-Evariste Dagand. "Normalization by realizability also evaluates". In: *JFLA*. 2015.

[101] Vincent Siles and Hugo Herbelin. "Pure Type System Conversion is Always Typable". In: *J. Funct. Program.* 22.2 (Mar. 2012), pp. 153–180. ISSN: 0956-7968. DOI: 10.1017/S0956796812000044. URL: http://dx.doi.org/10.1017/S0956796812000044.

[102] Thomas Streicher and Ulrich Kohlenbach. "Shoenfield is Gödel after Krivine". In: *Math. Log. Q.* 53.2 (2007), pp. 176–179.

[103] Kazushige Terui. "Computational ludics". In: *Theor. Comput. Sci.* 412.20 (2011), pp. 2048–2071. DOI: 10.1016/j.tcs.2010.12.026.

[104] A. M. Turing. "Computability and lambda-definability". In: *The Journal of Symbolic Logic* 2 (04 Dec. 1937), pp. 153–163. ISSN: 1943-5886. DOI: 10.2307/2268280.

[105] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study: http://homotopytypetheory.org/book, 2013.

[106] Christopher P. Wadsworth. "Semantics and pragmatics of the lambda-calculus". PhD thesis. Programming Research Group, Oxford University, 1971.

[107] Noam Zeilberger. "Polarity and the Logic of Delimited Continuations." In: *LICS*. IEEE Computer Society, 2010, pp. 219–227. ISBN: 978-0-7695-4114-3.