université PARIS-SACLAY

The Semantics of Effects: Centrality, Quantum Control and Reversible Recursion

De la sémantique des effets : centralité, contrôle quantique et récursivité réversible

Thèse de doctorat de l'université Paris-Saclay

École doctorale n° 580, Sciences et Technologies de l'Information et de la Communication (STIC)
Spécialité de doctorat : informatique
Graduate School : Informatique et Sciences du Numérique
Référent : Faculté des sciences d'Orsay

Thèse préparée au **Laboratoire Méthodes Formelles** (Université Paris-Saclay, CNRS, ENS Paris-Saclay, Inria), sous la direction de **Pablo Arrighi**, professeur Université Paris-Saclay, et le co-encadrement de **Benoît Valiron**, maître de conférence CentraleSupélec, et de **Vladimir Zamdzhiev**, chercheur Inria.

Thèse soutenue à Gif-sur-Yvette, le 19 juin 2024, par

Louis Lemonnier

Composition du jury

Membres du jury avec voix délibérative

Thomas Ehrhard Directeur de Recherche, CNRS & Université Paris-Cité	Rapporteur & Examinateur
Laurent Regnier Professeur, Université de Aix-Marseille	Rapporteur & Examinateur
Claudia Faggian Chargée de Recherche, CNRS & Université Paris-Cité	Examinatrice
Jean Goubault–Larrecq Professeur, ENS Paris-Saclay	Examinateur
Marie Kerjean Chargée de Recherche, CNRS & Université Sorbonne Paris-Nord	Examinatrice

Membres invités

Pablo Arrighi Professeur, Université Paris-Saclay	Directeur de thèse
Benoît Valiron Maître de conférence, CentraleSupélec	Co-encadrant
Vladimir Zamdzhiev Chercheur, Inria	Co-encadrant

Thèse de Doctorat
NNT : 2024UPASG030

université PARIS-SACLAY

ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Titre : De la sémantique des effets : centralité, contrôle quantique et récursivité réversible
Mots clés : Informatique quantique – Sémantique – Languages de programmation – Théorie des catégories

Résumé : Le sujet de cette thèse est axé sur la théorie des langages de programmation. Dans un langage de programmation suffisamment bien défini, le comportement des programmes peut être étudié à l'aide d'outils empruntés à la logique et aux mathématiques, énonçant des résultats sans exécuter le code. Ce domaine de l'informatique est appelé *sémantique*. La sémantique d'un langage peut se présenter sous plusieurs formes : dans notre cas, des sémantiques opérationnelles, des théories équationnelles et des sémantiques dénotationnelles. Les premières donnent un sens opérationnel aux programmes, au sein de la syntaxe du langage. Elles simulent les opérations qu'un ordinateur est censé effectuer s'il exécute le programme. Une théorie équationnelle fonctionne également de manière syntaxique : elle indique si deux programmes effectuent la même opération sans informer sur la procédure. Enfin, la sémantique dénotationnelle est l'étude mathématique des programmes, généralement à l'aide de la théorie des catégories. Elle permet par exemple de prouver qu'un programme se termine ou non.

Cette thèse se concentre sur la sémantique des effets dans les langages de programmation – une fonctionnalité ajoutée à un langage, gérant des données secondaires ou des résultats probabilistes. Eugenio Moggi, en 1991, a publié un travail fondateur sur l'étude de la sémantique des effets, soulignant la relation avec les monades en théorie des catégories. La première contribution de cette thèse suit directement le travail de Moggi, en étudiant la commutativité des effets dans un langage de programmation à travers le prisme des monades. Les monades sont la généralisation de structures algébriques telles que les monoïdes, qui ont une notion de centre : le centre d'un monoïde est une collection d'éléments qui commutent avec tous les autres dans le monoïde. Nous fournissons les conditions nécessaires et suffisantes pour qu'une monade ait un centre. Nous détaillons également la sémantique d'un langage de programmation avec des effets qui portent des informations sur les effets qui sont centraux. De plus, nous fournissons un lien fort – un résultat de langage interne – entre ses théories équationnelles et sa sémantique dénotationnelle.

Le deuxième axe de la thèse est l'informatique quantique, perçue comme un effet réversible. Le quantique est un domaine émergent de l'informatique qui utilise la puissance de la mécanique quantique pour calculer. Au niveau des langages de programmation, de nouveaux paradigmes doivent être développés pour être fidèles aux opérations quantiques. Les opérations quantiques physiquement permises sont toutes réversibles, à l'exception de la mesure ; cependant, la mesure peut être reportée à la fin du calcul, ce qui nous permet de nous concentrer d'abord sur la partie réversible et d'appliquer ensuite la mesure pour obtenir des résultats. Dans le chapitre correspondant, nous définissons un langage de programmation réversible, avec types simples, qui effectue des opérations quantiques *unitaires*. Une sémantique dénotationnelle et une théorie équationnelle adaptées au langage sont présentées, et nous prouvons que cette dernière est complète. Ce travail vise à fournir des bases solides pour l'étude du contrôle quantique d'ordre supérieur.

En outre, nous étudions la récursion réversible, en fournissant une sémantique opérationnelle et dénotationnelle adéquate à un langage de programmation fonctionnel, réversible et Turing-complet. La sémantique dénotationnelle utilise l'enrichissement dcpo des catégories inverses. Ce modèle mathématique sur l'informatique réversible ne se généralise pas directement à sa version quantique. Dans la conclusion, nous détaillons les limites et l'avenir possible du contrôle quantique d'ordre supérieur.

Title: The Semantics of Effects: Centrality, Quantum Control and Reversible Recursion
Keywords: Quantum computing – Semantics – Programming Languages – Category theory

Abstract: The topic of this thesis revolves around the theory of programming languages. In a sufficiently well-defined programming language, the behaviour of programs can be studied with tools borrowed from logic and mathematics, allowing us to state results without executing the code. This area of computer science is called "semantics". The semantics of a programming language can take several forms: in this thesis, we work with operational semantics, equational theories, and denotational semantics. The former gives an operational meaning to programs but within the language's syntax. It simulates the operations a computer is supposed to perform if it were running the program. An equational theory also works syntactically: it indicates whether two programs perform the same operation without giving any information on the procedure. Lastly, denotational semantics is the mathematical study of programs, usually done with the help of category theory. For example, it allows us to prove whether a program terminates.

This thesis focuses on the semantics of effects in programming languages – namely, a feature added to a language, *e.g.* handling side data or probabilistic outputs. Eugenio Moggi, in 1991, published foundational work on the study of the semantics of effects, highlighting the relationship with monads in category theory. The first contribution of this thesis directly follows Moggi's work, studying the commutativity of effects in a programming language through the prism of monads. Monads are the generalisation of algebraic structures such as monoids, which have a notion of centre: the centre of a monoid is a collection of elements which commute with all others in the monoid. We provide the necessary and sufficient conditions for a monad to have a centre. We also detail the semantics of a programming language with effects that carry information on which effects are central. Moreover, we provide a strong link – an internal language result – between its equational theories and its denotational semantics.

The second focus of the thesis is quantum computing, which is seen as a reversible effect. Quantum computing is an emergent field in computer science that uses the power of quantum mechanics to compute. At the level of programming languages, new paradigms need to be developed to be faithful to quantum operations. Physically permissible quantum operations are all reversible, except measurement; however, measurement can be deferred at the end of the computation, allowing us to focus on the reversible part first and then apply measurement to obtain results. In the corresponding chapter, we define a simply-typed reversible programming language performing quantum operations called "unitaries". A denotational semantics and an equational theory adapted to the language are presented, and we prove that the latter is complete. The aim of this work is to provide a solid foundation for the study of higher-order quantum control.

Furthermore, we study recursion in reversible programming, providing adequate operational and denotational semantics to a Turing-complete, reversible, functional programming language. The denotational semantics uses the dcpo enrichment of rig join inverse categories. This mathematical account of higher-order reasoning on reversible computing does not directly generalise to its quantum counterpart. In the conclusion, we detail the limitations and possible future for higher-order quantum control.

Remerciements/Acknowledgements

> "Écrire une thèse, c'est un peu comme courir un marathon, sauf que tu prends du poids au lieu d'en perdre." — LL.

La langue. Quand j'ai commencé à réfléchir à ces remerciements, j'ai rapidement décidé de me laisser la liberté de passer d'une langue à une autre. C'est bien le seul endroit où je peux me le permettre, et c'est important que j'y intègre du français, tout comme de l'anglais, car des futur·es lecteurices de ces paragraphes ne parlent pas forcément les deux langues.

Avant la thèse. Comme nous allons parler de sciences, partons du début. Non pas le début de ma scolarité, mais disons le moment où j'ai commencé à envisager de faire de la recherche mon avenir. Je ne suis pas certain que j'aurais suivi cette voie si mes professeurs de mathématiques de Terminale, Vincent Hugerot et Vincent Boillon, ne m'avaient pas transmis leur passion. Puis, en sup et en spé, c'est surtout l'informatique qui attiré mon attention. Cela ne serait jamais arrivé sans Serge Aubert, le meilleur enseignant que j'ai pu connaître. Il ne cachait pas sa passion profonde pour l'informatique, qu'elle soit théorique ou pratique ; il transmettait son savoir à la fois avec douceur et fermeté, ce caractère faisant de lui mon modèle – et il l'est peut-être toujours. Je ne peux pas finir ce paragraphe sans mentionner Pascal Guelfi, qui a profondément bousculé mon savoir et ma culture mathématiques, et sans qui je n'aurais jamais pu intégrer une ENS. J'ajouterai que je n'aurais pas pu atteindre la rédaction de cette thèse sans la formation du département informatique de l'ENS Cachan (maintenant Paris-Saclay) et ses excellents professeurs. Observons, à regret, que toutes les personnes mentionnées ci-dessus sont des hommes. Je n'en ai pas souffert, mais si j'avais été une femme, j'aurais pu être impactée par ce manque de manque de représentation.

Starting into research. This thesis has been undirectly helped by how I was introduced in the world of research and academia, which was first done by Frédéric Dupuis, thanks to Simon Perdrix. Frédéric m'a permis de découvrir la recherche dans un environnement sain et encadré, je l'en remercie tout particulièrement. My other research experience before my PhD journey has happened with Aleks Kissinger and John van de Wetering. I really enjoyed those months playing with ZH-diagrams and !-boxes during my stay in Nijmegen ['nɛiˌmeːɣə].

My environment. During my PhD, I have met and spent time with a crazy amout of nice people. Merci aux (ex-)membres de l'ADEPS [1], en particulier Céline, Émilie, Giann Karlo, Sofiane. Merci pour ces soirées, ces conversations qui m'ont permis à la fois de décrocher du travail et de grandir en tant qu'humain. Petit mot pour mes étudiant·es à l'ENS avec qui j'ai passé des moments sympathiques même si ça ne l'était pas forcément pour elleux. Je tiens à remercier les personnes qui dirigent et font tourner le LMF [2], ça a été un grand plaisir d'intéragir avec vous. Merci à Nicolas, pour nos conversations, notre soutien mutuel, nos tentatives de collaboration, mais qu'on a eu du mal à entretenir par manque de temps. Merci à Kostia, pour m'avoir accompagné sur toute la première moitié de ma thèse, pour ces petits moments de non travail au labo ou en conf, tes régulières victoires à Towerfall et non régulières à Mario Kart, et enfin pour m'avoir fait découvrir le GT Scalp et plus généralement la communauté λ-calcul et logique linéaire. Peut-être un jour sauras-tu que les normalien·nes peuvent être des gens sympathiques. Merci à Titouan, qui a été mon premier collaborateur en dehors de mes encadrants, c'est toujours un plaisir de travailler avec toi. Merci à James pour son soutien et pour m'avoir préparé psychologiquement à déménager au Royaume-Uni. I hope I will keep hearing from you about *pro*strong *pro*functors. Merci aux camarades de conférences pour leur soutien, en particulier Lison, Aloÿs, Tito et Axel. I also thank Agustín (ciclismo y asados), Dongho (토포스와 우리의 대화), Evi (για την χαρά σου και για τις μπύρες μας), Jérôme (nos conversations du jeudi matin), Kinnari (તમારા એકસાથે કામ કરવા અને મને સમજવામાં મદદ કરવા બદલ આભાર), and the members of the QuaCS team and of the LMF. My stay there has been overall nice and enjoyable.

My supervisors. Of course, I would not be writing any of this if Benoît and Vladimir were not there to help me figure how to write a thesis.

Les mots ne sont pas suffisants pour remercier Benoît de m'avoir pris sous son aile, de m'avoir fait vivre la recherche comme il l'entendait, avec sa joie, son humour et sa bienveillance, et surtout de m'avoir laissé une liberté presque totale, afin d'explorer et de collaborer avec un nombre de personnes grandissant, tout au long de ma thèse. Benoît est également un expert pour corriger et améliorer les présentations orales. Tout ce qu'il y a de bon dans mes interventions est probablement grâce à lui.

Before starting at QuaCS, I was supposed to do my PhD thesis with Vladimir in Nancy. After many *péripéties* [3], I had funding to work in Saclay and not in Nancy. I was surprised when, a year or so later, Vladimir announced that he was joining us, and that he could supervise me. His supervision style is more serious and stricter, which was not a bad thing, all things considered. Работата на ум учи. I will probably miss the usual jokes and stories at the coffee break or at the bar (although I think I know most of them now).

To young people reading this: being successful with a PhD thesis is not only about having strong scientific skills; it also requires to work with the right people *for you*. Knowing the people you are going to spend 3+ years with is very important, and I would even say that this is more important than the PhD subject.

Collaborating. Benoît, Kostia and I have been working with Robin Kaarsgaard, quite unsuccessfully for now, because the subject I wanted to tackle with them happened to be harder than

1. Association des Doctorant·es de l'ENS Paris-Saclay.
2. Laboratoire d'informatique dans lequel j'ai réalisé ma thèse.
3. Roughly, "twists" in French.

what I thought. Thanks to Robin for his patience. Kommer tid, kommer råd.

Le jury. Merci à Laurent Regnier et Thomas Ehrhard qui ont accepté de relire ce manuscrit en détail, et merci à Marie Kerjean, Claudia Faggian et Jean Goubault-Larrecq de participer au jury.

Relecture. Ce manuscrit est passé sous le peigne fin de plusieurs personnes, à qui je dois beaucoup. Un gigantesque merci à James, Kostia et Manon, ainsi que mes encadrants Benoît et Vladimir. À chaque relecture, j'avais des choses à améliorer ; chaque personne remarquait des typos ou des erreurs différentes, c'était un processus intéressant, et ça aurait été presque impossible pour moi de le faire seul.

Mes proches. Je tiens à remercier toustes mes ami·es – avec une attention particulière pour Elric, Marie, Adrien, Leela, Lucas, Lucas, Marion, Marion, Jean-Xavier, Victor, Alexandre, Alexandre, Rosemonde, Cécile, Gaël, Gaspard, Clara, Yoan, Yan, Valentin, Baptiste, Simon, Loïc, Quentin, Charlie, Arthur, Maxime, Maxime, Manon, le mot "ami·e" n'étant pas toujours suffisant – qui m'ont soutenu et accompagné ces dernières années. Il y a presque deux ans, j'ai commencé à jouer de la basse d'abord grâce à Gaël qui m'a prêté un instrument, puis en groupe avec Adrien, Hugo, Lucas et Marie. En plus d'avoir eu le plaisir de découvrir une nouvelle passion, ça m'a permis de me changer les idées quand j'en avais besoin.

Ma famille a toujours été un pilier inébranlable de ma vie, je leur dois une grande partie de ce que je suis. Cette thèse est dédiée à ma grand-mère, Marie-Thérèse, et mon grand-père, Jean-François, qui ne sont plus là pour lire ces remerciements mais qui m'accompagnent dans tout ce que je fais.

Contents

Résumé en français

> "Lae mathématicien·ne est engagé·e dans la poursuite d'un rêve sans fin, comprendre la structure de toute chose." — d'après Charles Ehresmann, mathématicien et membre fondateur du groupe Bourbaki.

Le principal objectif de cette thèse réside dans l'exploration des structures fondamentales de la programmation, avec un fort aspect théorique. En particulier, sont utilisées des structures provenant des mathématiques et de la logique pour prouver des propriétés sur les programmes. Ce domaine de l'informatique théorique est appelé *méthodes formelles*. Dans cette thèse, nous nous concentrons sur les langages de programmation *formels*. Ces derniers sont développés et étudiés pour affirmer et extraire des propriétés formelles dans des domaines spécifiques de la conception des langages de programmation. Le langage de programmation formel le plus standard est le λ-calcul (prononcez « *lambda* calcul »), introduit par Alonzo Church [Chu32] dans le cadre de son programme de recherche sur les fondements des mathématiques. La présentation du λ-calcul est en apparence simple : dans sa syntaxe, on peut former des fonctions et appliquer ces fonctions. Cependant, ce langage est *Turing complet*, ce qui signifie que tout programme *calculable* peut être représenté dans le λ-calcul. Il est particulièrement pratique de travailler en λ-calcul et il est simple d'y ajouter presque n'importe quel type de fonctionnalité, en ajoutant par exemple des combinateurs. Il est également flexible, en présentant plusieurs stratégies de calcul : en effet, on peut par exemple choisir de calculer d'abord le contenu des fonctions ou l'argument pris par ces fonctions.

Dans cette thèse, l'objectif est d'étudier deux aspects de la conception des langages de programmation, à savoir le flot de contrôle et les effets. Le *flot de contrôle* délimite les prises de décision sous-jacentes à l'exécution des tâches dans un paradigme de programmation. Il existe différentes manières de contrôler le flot d'un langage de programmation. Les langages impératifs, tels que `Python`, sont contrôlés par des instructions comme « if » et « while », tandis que les langages fonctionnels comme `Caml` sont plus subtils, avec un contrôle réalisé par des appels de fonctions ; ce dernier s'inspire du λ-calcul et de son interprétation mathématique. L'autre aspect qui nous intéresse est celui des *effets* : un calcul *à effet* se distingue fondamentalement aux calculs *purs* ; en d'autres termes, traiter des effets en programmation signifie que l'on fait la

distinction entre les opérations de base d'un langage et ses interactions avec le monde extérieur. Ces effets se présentent sous différentes formes. Il y a des effets généraux, parfois appelés *effets de bord*, qui interagissent directement avec un agent extérieur au programme – par exemple, l'écriture sur une bande séparée, la gestion des entrées et sorties de données. Il est également possible de rencontrer des *effets algébriques*, tels que l'introduction de non-déterminisme ou de comportements probabilistes. Le calcul quantique peut également être vu comme un effet algébrique. Ces effets sont *algébriques* dans le sens où ils présentent des caractéristiques issus de l'algèbre en mathématique. Cela signifie, en particulier, que les questions qui s'appliquent aux structures algébriques peuvent également s'appliquer aux effets – par exemple, si deux effets commutent entre eux ou non.

Dans cette thèse, nous proposons une étude formelle de ces aspects à travers le prisme de la sémantique, un paradigme en informatique qui attribue des interprétations logiques et mathématiques aux programmes. Une étude sémantique d'un langage de programmation nous permet de déduire des propriétés sur les programmes – par exemple, s'ils terminent ou non. Les composantes mathématiques d'une étude sémantique sont, dans notre cas, réalisées avec l'aide de la théorie des catégories.

Sémantique

Du grec ancien σημᾰντῐκός, *qui donne du sens*.

La notion de sémantique en théorie de l'informatique a débuté avec Robert Floyd [Flo67], dans une tentative de formaliser ce qui est attendu d'un langage de programmation d'un point de vue logique. La sémantique d'un langage de programmation peut revêtir différentes formes. Dans cette thèse, nous nous intéressons en particulier à trois d'entre elles.

- La première est appelée *sémantique opérationnelle*. Une sémantique opérationnelle décrit généralement, à travers des règles de réécriture ou des règles d'inférence, les opérations qu'un langage de programmation est censé effectuer. Par exemple, étant donné un programme *formel* $\mathtt{t}$, la sémantique opérationnelle pourrait détailler par exemple l'état du programme après une étape de calcul. Cela s'écrit souvent $\mathtt{t} \to \mathtt{t}'$.
- Une autre forme de sémantique consiste à fournir une *théorie équationnelle*. Une théorie équationnelle formalise si deux programmes sont censés produire le même résultat, et cela s'écrit généralement $\mathtt{t} = \mathtt{t}'$. Elle est souvent plus générale qu'une sémantique opérationnelle, dans le sens où si $\mathtt{t} \to \mathtt{t}'$, alors $\mathtt{t} = \mathtt{t}'$.
- La dernière forme de sémantique utilisée dans cette thèse est la sémantique dénotationnelle, où cette fois, ce sont les mathématiques qui sont utilisées pour donner un sens à un programme. Cette sémantique représente les actions d'un programme sous forme d'une fonction qui prend en entrée l'état de départ du programme. Elle est considérée comme une manière de s'abstraire de la syntaxe du langage. Elle est pratique à plusieurs égards : on peut prouver des propriétés sur un langage sans dépendre de la syntaxe ; elle nous permet également d'utiliser des travaux mathématiques antérieurs réalisés de manière indépendante, et elle fournit parfois de nouvelles intuitions sur le paradigme de programmation utilisé.

L'isomorphisme de *Curry-Howard* [Cur34, How80] établit un lien fort entre les programmes

et les preuves en logique formelle : c'est en réalité une correspondance bijective entre la sémantique opérationnelle des programmes et la réécriture en théorie de la preuve. L'ajout de la sémantique dénotationnelle dans l'étude des langages de programmation, en particulier avec l'aide de la théorie des catégories, a conduit à une correspondance cette fois entre les programmes, les preuves et les catégories, appelée la *correspondance Curry-Howard-Lambek*.

La théorie des catégories est la science des fonctions, décrivant les structures mathématiques à travers des morphismes qui peuvent être composés ; contrairement à la théorie des ensembles où l'accent est mis sans surprise sur les ensembles, et non sur les fonctions. La composition signifie que, étant donné un morphisme $X \to Y$ et un morphisme $Y \to Z$, il existe un morphisme $X \to Z$ qui est le résultat de la composition des deux précédents. Une catégorie est une collection d'objets et de morphismes, et la théorie des catégories définit un cadre formel pour parler de ces morphismes. Son vocabulaire permet de formaliser des énoncés généraux sur diverses structures mathématiques. Ce vocabulaire est la clé de voûte du contenu mathématique de cette thèse et il est donné et expliqué tout au long du Chapitre 1.

La théorie des catégories a un impact significatif dans l'étude mathématique des langages de programmation, car les programmes sont eux-mêmes des morphismes : ils transforment des ensemblese de données en d'autres ensembles de données. De plus, deux programmes peuvent être composés ; dans de nombreux langages de programmation, la composition de deux programmes est obtenue par concaténation. Il est donc naturel d'utiliser la théorie des catégories pour étudier mathématiquement les langages de programmation.

Les effets vus de manière externe

Dans le domaine des langages de programmation, le concept d'effets englobe les interactions observables entre un programme et son environnement, encapsulant des actions qui dépassent le domaine qu'on appelle du calcul *pur*. Contrairement aux calculs purement fonctionnels, qui présentent un comportement déterministe, les calculs avec effet permettent aux programmes d'interagir avec des entités externes, de manipuler des états ou d'effectuer des opérations d'entrée/sortie. Ces effets jouent un rôle crucial dans la définition du comportement et de la fonctionnalité des langages de programmation.

Les effets peuvent être traités de manière *externe*. Par exemple, dans un système typé – c'est-à-dire un langage où des labels spécifiques, appelées *types*, sont attribuées aux termes – on peut séparer les types de calculs purs des types de calculs à effet. Ces derniers sont souvent attribués à une *modalité*. C'est ce qui est fait dans le métalangage de Moggi [Mog91], basé sur un λ-calcul simplement typé avec des types supplémentaires pour les effets.

De plus, la sémantique dénotationnelle des effets en théorie des catégories a été largement étudiée par Moggi [Mog91, Mog89] (voir les détails dans la section §1.5.2). Il montre que les effets sont correctement interprétés par des *monades*. Ces dernières sont la généralisation catégorique des monoïdes de la théorie des ensembles, où un calcul sans effet correspond à l'élément neutre du monoïde, et une composition d'effets est similaire à la multiplication. Il est naturel de se demander si les propriétés sur les monoïdes s'appliquent également aux monades. En particulier, nous nous concentrons sur la question de la commutativité. Deux éléments x et y dans un monoïde commutent si le produit de x et y est le même que le produit de y et x. Un élément x est *central* s'il commute avec tous les autres éléments du monoïde. Par conséquent,

on peut se demander ce qu'est un *effet central* dans une monade, qui est la généralisation directe de la notion de monoïde. Dans le Chapitre 2, nous fournissons les réponses à la question de la centralité des effets.

Les effets vus de manière interne

Contrairement aux effets travaillant avec un périphérique externe – par exemple, l'entrée/sortie –, les *effets algébriques* sont souvent considérés internes au langage et l'informatique quantique n'y fait pas exception.

Informatique quantique et réversibilité. Les données quantiques sont caractérisées par la *superposition*. Alors qu'un bit classique prend ses valeurs dans l'ensemble $\{0, 1\}$, un bit quantique – souvent écrit *qubit* – est donné par une superposition :

$$|\varphi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle$$

où $|0\rangle$ et $|1\rangle$ sont des vecteurs dans un espace de Hilbert et α et β sont des nombres complexes. Avec cette présentation, on peut voir dès lors que l'informatique quantique a sa place parmi les effets algébriques. Cependant, il existe d'autres conditions sur un qubit pour qu'il soit physiquement admissible. Les vecteurs $|0\rangle$ et $|1\rangle$ doivent être orthogonaux et les nombres complexes α et β doivent vérifier la condition suivante : $|\alpha|^2 + |\beta|^2 = 1$. Ce sont les conditions de *normalisation*, et un état $|\varphi\rangle$ qui vérifie ces conditions est dit *normalisé*.

Pour préserver cette normalisation, les opérations quantiques admissibles – appelées *unitaires* – doivent être réversibles. Il existe d'autres opérations en informatique quantique qui ne sont pas réversibles : une pour créer des états quantiques, par exemple initialisant à $|0\rangle$, et une pour détruire des données quantiques, appelée *mesure*. Cette dernière envoie l'état $|\varphi\rangle$ sur 0 avec une probabilité $|\alpha|^2$ et sur 1 avec une probabilité $|\beta|^2$.

Une première approche de la programmation intégrant du quantique est le λ-calcul quantique [SV09]. Néanmoins, ce langage ne traite pas la programmation quantique comme un effet algébrique, car il nécessite de mesurer les qubits pour en tirer des données classiques pour contrôler le flux d'exécution.

Pour rester dans un effet algébrique quantique, une solution serait de considérer uniquement les opérations quantiques réversibles, comme le montre [SVV18]. Dans cet article, la réversibilité des fonctions du langage est assurée grâce à l'aide du pattern-matching réversible.

Pattern-matching réversible. Considérons la fonction booléenne constante valant toujours 1. Cette fonction n'est pas *réversible* car elle n'est pas injective. Pour être réversible, une fonction $f\colon X \to Y$ doit être *déterministe en avant* et *déterministe en arrière*. Le premier est généralement supposé : cela signifie que tout $x \in X$ a une seule image par la fonction f. Le second signifie que pour tout $y \in Y$, il existe au plus un $x \in X$ tel que $f(x) = y$. C'est un synonyme d'*injectivité*.

En informatique, le type des bits est donné par $1 \oplus 1$ où la somme directe $\oplus$ peut par exemple dénoter l'union disjointe d'ensembles. Nous introduisons également deux combinateurs : l'injection à gauche $\mathtt{inj}_l$ et l'injection à droite $\mathtt{inj}_r$, tels que les termes $\mathtt{inj}_l\ *$ et $\mathtt{inj}_r\ *$ représentent respectivement le bit 0 et le bit 1. La fonction constante peut alors être donnée

par :

$$\begin{cases} 1 \oplus 1 & \to & 1 \oplus 1 \\ \mathtt{inj}_l * & \mapsto & \mathtt{inj}_r * \\ \mathtt{inj}_r * & \mapsto & \mathtt{inj}_r * \end{cases}$$

Cette présentation fournit une intuition sur l'injectivité grâce à la syntaxe, car les deux sorties possibles sont injectées du même côté. Pour assurer la réversibilité de la mise en correspondance de motifs, nous forçons les motifs d'un même côté à être *orthogonaux*, une notion qui fait écho en algèbre linéaire, où les vecteurs sont orthogonaux s'ils appartiennent à des parties distinctes d'une somme directe. Étant donné n'importe quel terme t, $\mathtt{inj}_l\ t$ et $\mathtt{inj}_r\ t$ sont orthogonaux. Nous voyons dans le chapitre 3 comment cette notion d'orthogonalité peut être formalisée syntaxiquement.

Étant donnés deux programmes $f : \mathbb{B} \to \mathbb{B}$ et $g : \mathbb{B} \to \mathbb{B}$ qui sont réversibles, le programme suivant :

$$\phi = \begin{cases} \mathbb{B} \otimes \mathbb{B} & \to & \mathbb{B} \otimes \mathbb{B} \\ (0, x) & \mapsto & (0, fg(x)) \\ (1, x) & \mapsto & (1, gf(x)) \end{cases} \tag{1}$$

est par exemple réversible.

Sémantique de la programmation quantique. Il existe, dans la littérature, de nombreux modèles sémantiques différents pour les langages de programmation quantiques [Val08, Mal13, PSV14, CW16, JKL+22, TA24]. Cependant, toutes ces approches sont des modèles appropriés pour l'informatique quantique avec un *contrôle classique* uniquement : les tests, tels que les instructions `if` ou les appels récursifs, sont contrôlés par des données classiques. Dans le λ-calcul quantique, un qubit doit être mesuré avant d'influencer le contrôle d'un programme. Ces modèles ne nous concernent donc pas, car l'utilisation de la mesure détruit la superposition quantique, et elle ne préserve donc pas l'effet quantique susmentionné. Dans cette thèse, nous souhaitons préserver cet effet, d'où un accent sur un *flux de contrôle quantique*.

En informatique quantique, l'exemple de fonction réversible donné ci-dessus (2), où les bits sont généralisé à des qubits, est appelé le *quantum switch* [CDPV13]. La fonction $\lambda f.\lambda g.\phi$ ne peut pas être exprimée dans le λ-calcul quantique ni dans aucun langage avec un contrôle classique, car le flux du programme ϕ doit être contrôlé par des données quantiques. L'un des objectifs de cette thèse est donc d'établir des bases solides pour la sémantique d'un langage de programmation avec un flux de contrôle quantique.

Contribution de la Thèse

Cette thèse aborde les effets d'un point de vue algébrique. Tout d'abord, dans le chapitre 2, nous étudions la question de la commutativité des effets à travers leur sémantique dénotationnelle – à savoir, les monades fortes. Nous commençons par poser les bases catégoriques pour définir ce qu'est le centre d'une monade (voir le théorème 2.11) et ce qu'est une *sous-monade centrale* (voir le théorème 2.29). Nous proposons ensuite une syntaxe proche du métalangage de Moggi pour capturer les effets centraux, et nous introduisons à la fois des théories et une sémantique dénotationnelle pour ce métalangage, que nous avons appelé *Central Submonad Calculus*. Nous montrons un résultat de langage interne (voir le théorème 2.60), prouvant que

les théories équationnelles du Central Submonad Calculus sont essentiellement équivalentes aux modèles de ce calcul.

En seconde partie, dans le chapitre 3, nous nous concentrons sur un sujet plus spécifique, qui est l'informatique quantique vue comme un effet algébrique réversible. Nous présentons un langage de programmation réversible qui capture cet effet de manière interne. En particulier, le langage est conçu pour manipuler des états quantiques *normalisés* et pour préserver cette normalisation. Cela se fait par l'introduction d'une notion syntaxique d'*orthogonalité*, et de l'équivalent syntaxique des bases orthonormales appelé une *décomposition orthogonale*. Nous proposons une théorie équationnelle et une sémantique dénotationnelle pour le langage, et nous prouvons la *complétude* (voir le théorème 3.67) : étant donnés deux termes bien typés, ils sont égaux dans la théorie équationnelle si et seulement s'ils sont égaux dans la sémantique dénotationnelle.

Ensuite, nous abordons la question des types de données infinis et de la récursivité dans la programmation réversible, dans le but de l'adapter à l'effet réversible quantique. Dans le Chapitre 4, nous introduisons un langage réversible similaire au précédent, cette fois-ci sans effets quantiques, mais où des types de données inductifs et la récursivité sont ajoutées. Nous donnons une sémantique opérationnelle au langage, où les opérations *d'ordre supérieur*, telles que les appels récursifs, sont considérées séparément des opérations réversibles. Nous fournissons également une sémantique dénotationnelle dans les *catégories join inverse rig* qui ont les propriétés exactes nécessaires pour modéliser le langage. Nous montrons que ce modèle est *adéquat* par rapport à la sémantique opérationnelle (voir le théorème 4.29), et nous fournissons ensuite un résultat proche de la *complétude totale* (voir le théorème 4.61), montrant que toute fonction calculable dans le modèle concret des injections partielles est représentable par une fonction dans le langage.

Enfin, le chapitre 5 contient des commentaires sur la récursion dans le contexte de l'effet réversible quantique. En effet, il s'avère que cet effet ne peut pas être étudié comme une monade. De plus, les techniques utilisées dans le Chapitre 4 ne se généralisent pas au cas quantique : les catégories en jeu ne sont pas enrichies dans $\mathbf{DCPO}$ et ne semblent pas être tracées de manière convenable. Sur une note plus positive, nous proposons une solution potentielle à la récursion quantique avec l'aide de la récursion gardée, un cadre dans lequel les appels récursifs sont *gardés* par des modalités de retard. Pour ce faire, nous établissons un modèle catégorique pour la récursion quantique gardée, et prouvons que ce modèle est adapté pour interpréter la récursion (voir le théorème 5.19) et les types inductifs (voir le théorème 5.32).

Introduction

> "Mathematical Science shows what is. It is the language of the unseen relations between things. But to use & apply that language we must be able fully to appreciate, to feel, to seize, the unseen, the unconscious." — according to Ada Lovelace.

The primary focus of this thesis resides in the exploration of foundational structures within programming, with a strong theoretical aspect. We use structures drawn from mathematics and logic to prove properties on programs. This area of theoretical computer science is called *formal methods*. In particular in this thesis, we set our gaze on *formal* programming languages. They are developed and studied to assert and extract formal properties in specific areas of programming language design. The most standard formal programming language is the λ-calculus, introduced by Alonzo Church [Chu32] as a part of his research programme in the foundations of mathematics. The presentation of the λ-calculus is in appearance simple: in its syntax, one can either form functions, or apply those functions. However, this language is *Turing complete*, meaning that any *computable* program can be represented in the λ-calculus. It is especially convenient to work with it because it is simple to add almost any kind of feature to it, by adding combinators for example. It is also flexible, with different possible computation strategies.

In this thesis, we set ourselves to study two aspects of language design, namely, control flow and effects. The *control flow* delineates the decision-making processes underlying task execution within a programming paradigm. There are various ways to control the flow of a programming language. Imperative languages, such as `Python`, are controlled by statements like "if" and "while", whereas functional languages such as `Caml` are more subtile with control through functions calls; and the latter takes inspiration in the λ-calculus and its mathematical interpretation. The other aspect we care about is *effects*: an *effectful* computation is a fundamental distinction from *pure* computation; in other words, dealing with effects in programming means distinguishing between the core operations of a language and its interactions with the outside world. These effects come in different forms. There are some general effects, sometimes called *side* effects, that interact directly with an agent outside of the program – for example,

writing on a separate tape, managing inputs and outputs of data. We also encounter *algebraic effects*, such as the introduction of non-determinism or probabilistic behaviour. Quantum computation can also be seen as an algebraic effect. These effects are *algebraic* in the sense that they exhibit algebraic characteristics. This means, in particular, that the questions that apply to algebraic structures can also apply to effects – for example, whether two effects commute with each other.

In this thesis, we propose a formal study of these aspects through the lens of semantics, a paradigm in computer science that assigns logical and mathematical interpretations to programs. A semantic study of a programming language allows us to derive statements about programs – for example, whether they terminate. The mathematical components of a semantic study is, in our case, done with the help of category theory.

Semantics

From Ancient Greek σημᾰντῐκός, *which gives meaning*.

Semantics started with Robert Floyd [Flo67], as an attempt at formalising what is expect from a programming language with a logic point of view. The semantics of a programming language can come in many different shapes. In this thesis, we are concerned with three in particular.

- The first one is called *operational semantics*. An operational semantics usually outlines, through rewriting rules or inference rules, the operations a programming language is expected to perform. For example, given a *formal* program $\mathtt{t}$, the operational semantics could detail for example what is the state of the program after one computational step. This is often written $\mathtt{t} \to \mathtt{t}'$.
- Another form of semantics consists in providing an *equational theory*. An equational theory formalises whether two programs are expected to eventually perform the same operation, and we write that $\mathtt{t} = \mathtt{t}'$. It is usually more general than an operational semantics, in the sense that if $\mathtt{t} \to \mathtt{t}'$, then we have $\mathtt{t} = \mathtt{t}'$.
- The final form of semantics used in this thesis is denotational semantics. As mentioned above, the denotational semantics of a programming language involves mathematics. It represents the actions of a program as a function on the inputs. It is thought as a way of abstracting away from the syntax of the language. It is practical in several ways: one can prove properties about a language without depending on the syntax, it also allows us to use previous mathematical work realised independently, and it sometimes provides new intuitions on the matters at hand.

The *Curry-Howard* isomorphism [Cur34, How80] establishes a strong link between programs and proofs in formal logic: it states a one-to-one correspondence between the operational semantics of programs and proof-theoretic rewriting. The addition of denotational semantics in the study of programming language, especially with the help of category theory, led to a one-to-one correspondence between programs, proofs and categories, called the *Curry-Howard-Lambek correspondence*.

Category theory describes mathematical structures through morphisms – for example, functions or relations – that can be composed; as opposed to set theory in which the emphasis is on

sets, and not functions. Composition means that given a morphism $X \to Y$ and a morphism $Y \to Z$, there exists a morphism $X \to Z$ that is the result of the two former morphisms *composed*. A category is a collection of objects and morphisms, and category theory defines a framework for talking about morphisms. This vocabulary helps formalise general statements on various mathematical structures. This vocabulary is the cornerstone of the mathematical content of this thesis. It is given and explained along Chapter 1.

Category theory is especially meaningful in the mathematical study of programming language, because programs are themselves morphisms: they transform bits of data into bits of data. Moreover, two programs can be composed; in imperative programming languages for instance, the composition of two programs is obtained by concatenation. Thus it is only natural to use category theory to study programming languages.

Effects as External Behaviour

In the domain of programming languages, the concept of "effects" encompasses the interactions between a program and its environment, encapsulating actions that extend beyond the realm of pure computation. Unlike purely functional computations, which adhere strictly to mathematical principles and exhibit deterministic behaviour, effectful computations enable programs to interact with external entities, manipulate states, or perform input/output operations. These effects play a pivotal role in shaping the behaviour and functionality of programming languages.

Because of this distinction between internal states and the environment, a natural approach is to consider effects *externally* to the program. For example, in a typed system – *i.e.* a language where secific labels, called *types*, are assigned to terms – one can separate the types of pure computations from the types of effectful computations. The latter are often assigned a *modality*. This is what is done in Moggi's metalanguage [Mog91], based on a typed λ-calculus with an additional type for effects.

Moreover, the denotational semantics of effects in category theory have been extensively studied by Moggi [Mog91, Mog89] (see details in §1.5.2). He shows that effects are suitably interpreted by *monads*. The latter are the category theoretical generalisation of monoids in set theory, where a computation without effect corresponds to the neutral element of the monoid, and a composition of effects is akin to the multiplication. It is only natural to wonder whether properties on monoids also apply to monads. In particular, we focus on the question of commutativity. Two elements x and y in a monoid are commutative if the product of x and y is the same as the product of y and x. An element x is central if it commutes with all other elements in the monoid. Consequently, one can wonder about what is a *central effect* in a monad. In Chapter 2, we provide the answers to the question of centrality of effects.

Effects as an Internal Behaviour

Contrary to effects working with an actual external device – for example, input/output –, *algebraic* effects – for instance, probabilities or non determinism – are often considered internally to the language. Quantum computing is not an exception in that regard.

Quantum computing and reversibility. Quantum data is characterised by *superpositions*. While a classical bit takes its values in the set $\{0, 1\}$, a quantum bit – often written *qubit* – is given by the superposition:

$$|\varphi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle$$

where $|0\rangle$ and $|1\rangle$ are vectors in a Hilbert space and α and β are complex numbers. With this presentation, we can already see that quantum computing has its place among algebraic effects. However, there are more conditions on a qubit for it to be physically admissible. The vectors $|0\rangle$ and $|1\rangle$ need to be orthogonal and the complex numbers α and β need to verify $|\alpha|^2 + |\beta|^2 = 1$. These are called *normalisation conditions*, and a state $|\varphi\rangle$ that verifies these conditions is said *normalised*.

To preserve this normalisation, the admissible quantum operations – called *unitaries* – need to be reversible. There are other operations in quantum computing that are not reversible: one to create quantum data, for example initialising at $|0\rangle$, and one to destroy quantum data, called *measurement*. The latter maps the state $|\varphi\rangle$ to 0 with probability $|\alpha|^2$ and to 1 with probability $|\beta|^2$.

A first approach to programming with quantum effects is the quantum λ-calculus [SV09]. Nevertheless, that language does not handle quantum programming as an algebraic effect, since it requires measurement into classical data to control the flow of execution.

To stay within a quantum algebraic effect, a solution would be to consider only the quantum reversible operations, as shown in [SVV18]. In that paper, the reversibility of functions of the language is ensured through the help of reversible pattern-matching.

Reversible pattern-matching. Consider the constant boolean function 1. This function is not *reversible* because it is not injective. To be reversible, a function $f : X \to Y$ has to *forward deterministic* and *backward deterministic*. The former is traditionally assumed: it means that any $x \in X$ has a single image by the function f. The latter means that for all $y \in Y$, there exists a most one $x \in X$ such that $f(x) = y$. It is a synonym of *injectivity*.

In computer science, the type of bits is given by $1 \oplus 1$ where the direct sum $\oplus$ can for example be the disjoint union of sets. We also introduce two combinators, the left injection $\mathtt{inj}_l$ and the right injection $\mathtt{inj}_r$, such that the terms $\mathtt{inj}_l\ *$ and $\mathtt{inj}_r\ *$ respectively respresent the bit 0 and the bit 1. The constant boolean function can then be given by:

$$\begin{cases} 1 \oplus 1 & \to & 1 \oplus 1 \\ \mathtt{inj}_l\ * & \mapsto & \mathtt{inj}_r\ * \\ \mathtt{inj}_r\ * & \mapsto & \mathtt{inj}_r\ * \end{cases}$$

This presentation provides an intuition on injectivity thanks to the syntax, as both possible outputs are injected to the same side. To ensure reversibility of pattern-matching, we force the patterns of a same side to be *orthogonal*, a notion that echoes in linear algebra, where vectors are orthogonal if they belong to separate parts of a direct sum. Given any term t, $\mathtt{inj}_l\ t$ and $\mathtt{inj}_r\ t$ are orthogonal. We see in Chapter 3 how this notion of orthogonality can be formalised syntactically.

Given two programs $f : \mathbb{B} \to \mathbb{B}$ and $g : \mathbb{B} \to \mathbb{B}$ that are reversible, the following program:

$$\phi = \begin{cases} \mathbb{B} \otimes \mathbb{B} & \to & \mathbb{B} \otimes \mathbb{B} \\ (0, x) & \mapsto & (0, fg(x)) \\ (1, x) & \mapsto & (1, gf(x)) \end{cases} \tag{2}$$

is for instance reversible.

Semantics of quantum computing. There are, in the literature, many different semantic models of quantum programming languages [Val08, Mal13, PSV14, CW16, JKL+22, TA24]. However, all those approaches are proper model for quantum computing with *classical control*: tests, such as `if` statements or recursive calls, are controlled by classical data. In the quantum λ-calculus, a qubit has to be measured before influencing the control of a program. These models are not of interest to this thesis, because the use of measurement breaks superposition, therefore it does not preserve the aforementioned quantum effect. In this thesis, we wish to preserve the effect. Hence a focus on a *quantum-controlled* flow.

In quantum computing, the example of reversible function given above (2), where bits are replaced with qubits, is called the *quantum switch* [CDPV13]. The function $\lambda f.\lambda g.\phi$ cannot be expressed in the quantum λ-calculus nor any language with classical control, because the flow of the program ϕ needs to be controlled by quantum data. One of the goals of this thesis is to lay foundations for the semantics of a language with a quantum control flow.

Contribution of the Thesis

This thesis tackles effects with an algebraic point of view. First, in Chapter 2, we study the question of commutativity of effects through their denotational semantics – namely, strong monads. We start by laying out categorical grounds to define what is the centre of a monad (see Theorem 2.11) and what is a *central* submonad (see Theorem 2.29). We then provide a syntax close to Moggi's metalanguage to capture central effects, and we introduce both theories and denotational semantics for this metalanguage, that we called the *Central Submonad Calculus*. We show an internal language result (see Theorem 2.60), proving that equational theories of the Central Submonad Calculus are basically equivalent to models of this calculus.

Secondly, in Chapter 3, we focus on a more specific subject, which is quantum computing seen as a reversible algebraic effect. We provide a reversible programming language that captures this effect internally. In particular, the language is designed to manipulate *normalised* quantum states and to preserve this normalisation. This is done through the introduction of a syntactical notion of *orthogonality* and of *orthogonal decomposition*, which is the syntactical equivalent to an orthonormal basis. We provide an equational theory and a denotational semantics for the language, and we prove *completeness* (see Theorem 3.67): given two well-typed terms, they are equal in the equational theory if and only if they are equal in the denotational semantics.

Then, we tackle the question of infinite data types and recursion in reversible programming, as an attempt to adapt it to the quantum reversible effect. In Chapter 4, we introduce a reversible language akin to the one before, this time without quantum effects, but where inductive data types and recursion are added. We give an operational semantics to the language, where *higher-order* operations, such as recursive calls, are considered separately to reversible operations. We also provide a denotational semantics in *join inverse rig categories* which have the exact properties needed to model the language. We show that this model is *adequate* with regard to the operational semantics (see Theorem 4.29), and we later provide a result close to *full completeness* (see Theorem 4.61), showing that any computable function in the concrete

model of partial injections is representable by a function in the language.

Finally, Chapter 5 contains comments on recursion in the context of quantum reversible effects. Indeed, it turns out that this effect cannot be studied as a monad. Moreover, the techniques used in Chapter 4 do not generalise to the quantum case: the categories at play are not enriched in $\mathbf{DCPO}$ and do not seem to be properly traced. On a more positive note, we provide a potential solution to quantum recursion with the help of guarded recursion, a framework in which recursive calls are *guarded* by delay modalities. To do so, we lay out a categorical model for guarded quantum recursion, and prove that this model is suitable to interpret recursion (see Theorem 5.19) and inductive types (see Theorem 5.32).

Chapter 1
Mathematical Background

"You cannot outsmart the model." — Vladimir Zamdzhiev.

Abstract

We introduce the background material in mathematics, and especially in category theory, necessary to navigate the thesis seamlessly. In some sections, basic notions of type theory and programming languages are presented and linked to their categorical semantics.

References. This background chapter is only made up of earlier work, published by several different authors. It is meant as an introduction to the material this thesis requires, and not as a literature review. The few proofs inserted here and there are provided by the author, for didactic purposes.

1.1 Category theory: some definitions

In this thesis, category theory is used as vocabulary to express a mathematical point of view on programs and programming languages; and what we often refer to as *interpretation*, *denotational semantics* or *denotation* is a map from a syntax – or a *language* – to a category, usually written $[\![-]\!]$. In other words, given a piece of syntax $\mathtt{t}$, which we refer to as a *term* of the syntax, its interpretation $[\![\mathtt{t}]\!]$ is given in a fixed category $\mathbf{C}$. If a few coherence properties are satisfied, we allow ourselves to call $\mathbf{C}$ a *model* of the language. We provide examples along the chapter, in §1.1.1, §1.2.1 and §1.2.2.

In this section, we recall the definitions required to work with category theory as a denotational model of programming languages. The author recommends the book of Tom Leinster [Lei16], which introduces category theory with more background, details and examples.

Definition 1.1 (Category)**.** A *category* $\mathbf{C}$ is a collection of objects – usually written with capital Latin letters $X, Y, Z, \ldots$ – and a collection of morphisms – written $f : X \to Y$ to indicate that f is a morphism from X to Y – such that:

- for every object X, there is a morphism $\mathrm{id}_X : X \to X$,

- for every pair of morphisms $f\colon X \to Y$, $g\colon Y \to Z$, there is a morphism $g \circ f\colon X \to Z$ called the composition of f and g,
- composition is associative: $(f \circ g) \circ h = f \circ (g \circ h)$,
- and for every morphism $f\colon X \to Y$, we have $\mathrm{id}_Y \circ f = f = f \circ \mathrm{id}_X$.

We write $\mathbf{C}(X, Y)$ for the collection of morphisms from X to Y.

Example 1.2. A very well-known category is the one with objects that are sets, and morphisms that are functions between sets; we write $\mathbf{Set}$ for this category. Note that whenever X and Y are sets, $\mathbf{Set}(X, Y)$ is also a set.

Example 1.3. Vector spaces – additive groups together with the outer action of a field $\mathbb{K}$ – with linear maps as morphisms also form a category, written $\mathbf{Vect}$.

A category $\mathbf{C}$ is called *locally small* if all $\mathbf{C}(X, Y)$ are sets. They are then called *homsets*, short for "sets of homomorphisms". A locally small category $\mathbf{C}$ is *small* if its collection of objects is a set.

Remark 1.4. Throughout the thesis, given two morphisms $f\colon X \to Y$ and $g\colon Y \to Z$, we write $gf\colon X \to Z$ for the composition $g \circ f\colon X \to Z$ when it is not ambiguous.

Category theory is better pictured with diagrams to represent morphisms. In a category $\mathbf{C}$, the composition of two morphisms $f\colon X \to Y$ and $g\colon Y \to Z$ is seen as the diagram:

$$X \xrightarrow{f} Y \xrightarrow{g} Z$$

Diagrams are sound thanks to associativity. It allows us to write the following diagram:

$$X \xrightarrow{f} Y \xrightarrow{g} Z \xrightarrow{h} T$$

without any need to be precise about in which order the composition is taken. Moreover, given $h\colon X \to Z$, the condition $h = g \circ f$ is described as the *commutativity* of the following diagram:

$$\begin{array}{ccccc} X & \xrightarrow{f} & Y & \xrightarrow{g} & Z \\ & \searrow & & \nearrow & \\ & & h & & \end{array}$$

Example 1.5 (Opposite Category)**.** Given a category $\mathbf{C}$, one can form the opposite category, written $\mathbf{C}^{\mathrm{op}}$, with the same objects as $\mathbf{C}$, such that there is a morphism $Y \to X$ for every morphism $X \to Y$ in $\mathbf{C}$. These new morphisms respect the same diagrams as in $\mathbf{C}$, but with reversed arrows.

Once we master the definition of a category, we can introduce some vocabulary on morphisms. We define what an isomorphism is. This definition echoes to the one in set theory.

Definition 1.6 (Isomorphism)**.** Given a category $\mathbf{C}$ and a morphism $f\colon X \to Y$ in that category, we say that f is an *isomorphism* if there exists a (unique) $f^{-1}\colon Y \to X$ such that $f \circ f^{-1} = \mathrm{id}_Y$ and $f^{-1} \circ f = \mathrm{id}_X$.

Given an isomorphism $f\colon X \to Y$, we say that the objects X and Y are isomorphic. We also introduce notions akin to injective and to surjective functions.

Definition 1.7 (Monomorphism). Given a category $\mathbf{C}$ and a morphism $f\colon X \to Y$ in that category, we say that f is a *monomorphism* – or that f is *monic* – if for all objects Z and all morphisms $g_1, g_2\colon Z \to X$, if $f \circ g_1 = f \circ g_2$, then $g_1 = g_2$.

Definition 1.8 (Epimorphism). Given a category $\mathbf{C}$ and a morphism $f\colon X \to Y$ in that category, we say that f is an *epimorphism* – or that f is *epic* – if for all objects Z and all morphisms $g_1, g_2\colon Y \to Z$, if $g_1 \circ f = g_2 \circ f$, then $g_1 = g_2$.

Remark 1.9. In $\mathbf{Set}$, the category of sets and functions, monomorphisms (resp. epimorphisms) are exactly injective (resp. surjective) functions.

A morphism that is monic and epic is not necessarily an isomorphism.

> **Lemma 1.10** ([ML98]). *Given a morphism f in a category $\mathbf{C}$, f is a monomorphism iff it is an epimorphism in $\mathbf{C}^{\mathrm{op}}$.*

Definition 1.11 (Functor). Given two categories $\mathbf{C}$ and $\mathbf{D}$, a *functor* $F\colon \mathbf{C} \to \mathbf{D}$ is a function on objects and on morphisms, such that: for all objects X in $\mathbf{C}$, there is an object $F(X)$ in $\mathbf{D}$, and for all morphsisms $f\colon X \to Y$, there is a morphism $F(f)\colon F(X) \to F(Y)$ in $\mathbf{D}$, and

- for all objects X in $\mathbf{C}$, $F(X)$ is an object of $\mathbf{D}$;
- for all morphisms $X \to Y$ in $\mathbf{C}$, $F(f)\colon F(X) \to F(Y)$ is a morphism in $\mathbf{D}$;
- for all objects X in $\mathbf{C}$, $F(\mathrm{id}_X) = \mathrm{id}_{F(X)}$;
- for all pairs of morphisms $f\colon X \to Y, g\colon Y \to Z$, $F(gf) = F(g)F(f)$.

Functors are often written with Latin capital letters F and G.

We abuse notations and sometimes drop the parenthesis when applying a functor. For example, the object $F(X)$ is often written FX when it is not ambiguous.

Example 1.12 (Identity functor). Given any category $\mathbf{C}$, one can define the identity functor $\mathrm{id}_{\mathbf{C}}\colon \mathbf{C} \to \mathbf{C}$ that maps any object to itself and any morphism to itself.

Example 1.13. We define $U\colon \mathbf{Vect} \to \mathbf{Set}$ that maps a vector space to its underlying set and that maps a linear map to itself, now seen as a function between sets. U is a functor, and is called the *forgetful functor*, because it forgets the structure of a vector space.

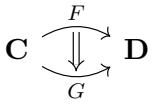
Example 1.14 (Hom functor). Given any locally small category $\mathbf{C}$ and an object X of $\mathbf{C}$, the assignment $\mathbf{C}(-, X)$ forms a functor $\mathbf{C}^{\mathrm{op}} \to \mathbf{Set}$ that maps an object Y to the set $\mathbf{C}(Y, X)$ and a morphism $f\colon Y \to Z$ in $\mathbf{C}$ to the function between sets $(- \circ f)\colon \mathbf{C}(Z, X) \to \mathbf{C}(Y, X)$. A similar functor, namely $\mathbf{C}(X, -)\colon \mathbf{C} \to \mathbf{Set}$, can also be defined.

Example 1.15 (Category of small categories). One can define the category $\mathbf{Cat}$, with small categories as objects and functors between them as morphisms. The identity functor $\mathbf{C} \to \mathbf{C}$ is described in Example 1.12. Given two functors $F\colon \mathbf{C} \to \mathbf{D}$ and $G\colon \mathbf{D} \to \mathbf{E}$, it is routine to show that $G \circ F$ is a functor $\mathbf{C} \to \mathbf{E}$.

Category theory is the theory of *arrows*, trying to establish morphisms whenever it is possible. Given two categories $\mathbf{C}$ and $\mathbf{D}$, and two functors F and G between them, we obtain the following diagram:

$$\mathbf{C} \underset{G}{\overset{F}{\rightleftarrows}} \mathbf{D}$$

This diagram does not necessarily commute. However, it can be filled with a new kind of arrow, as pictured below.



That new arrow is called a *natural transformation* and its definition is as follows.

Definition 1.16 (Natural transformation)**.** Given two categories $\mathbf{C}$ and $\mathbf{D}$, given two functors $F, G\colon \mathbf{C} \to \mathbf{D}$, a *natural transformation* $\alpha\colon F \Rightarrow G$ is a collection of morphisms indexed by the objects of $\mathbf{C}$ such that, for all morphisms $f\colon X \to Y$, the following diagram:

$$\begin{array}{ccc} F(X) & \xrightarrow{\alpha_X} & G(X) \\ {\scriptstyle F(f)}\downarrow & & \downarrow{\scriptstyle G(f)} \\ F(Y) & \xrightarrow[\alpha_Y]{} & G(Y) \end{array}$$

commutes.

Example 1.17. Given a set M, we define a functor $T \stackrel{\text{def}}{=} M \times -\colon \mathbf{Set} \to \mathbf{Set}$. Moreover, given a function $(- \cdot -)\colon M \times M \to M$ and an element $e \in M$, there is a natural transformation $\eta\colon \mathrm{id}_{\mathbf{Set}} \Rightarrow T$ and a natural transformation $\mu\colon T \circ T \Rightarrow T$, such that for all sets X:

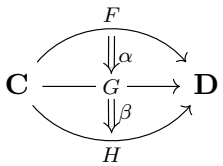
$$\eta_X = \left\{ \begin{array}{lcl} X & \to & M \times X \\ x & \mapsto & (e, x) \end{array} \right. \qquad \mu_X = \left\{ \begin{array}{lcl} M \times (M \times X) & \to & M \times X \\ (m_1, (m_2, x)) & \mapsto & (m_1 \cdot m_2, x) \end{array} \right.$$

In fact, if $(M, \cdot, e)$ is a monoid, then T is a monad (see Definition 1.95).

Example 1.18 (Functor Category)**.** Given two categories $\mathbf{C}$ and $\mathbf{D}$, we write $[\mathbf{C} \to \mathbf{D}]$ or $\mathbf{D}^{\mathbf{C}}$ for the category of functors $\mathbf{C} \to \mathbf{D}$ and natural transformations between them. Given a functor $F\colon \mathbf{C} \to \mathbf{D}$, the identity natural transformation $\mathrm{id}_F\colon F \Rightarrow F$ is a natural transformation whose components are all the identity; and given two natural transformations $\alpha\colon F \Rightarrow G$ and $\beta\colon G \Rightarrow H$, for all $f\colon X \to Y$ in $\mathbf{C}$, the diagram:

$$\begin{array}{ccccc} F(X) & \xrightarrow{\alpha_X} & G(X) & \xrightarrow{\beta_X} & H(X) \\ {\scriptstyle F(f)}\downarrow & & \downarrow{\scriptstyle G(f)} & & \downarrow{\scriptstyle H(f)} \\ F(Y) & \xrightarrow[\alpha_Y]{} & G(Y) & \xrightarrow[\beta_Y]{} & H(Y) \end{array}$$

commutes, and thus $\beta \circ \alpha$ defined as the pointwise composition is a natural transformation. This composition of natural transformations is also called the *vertical composition* because of the following diagram:

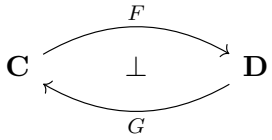


Some functor categories are often used in the literature, and therefore have a name of their own. For example, given a small category $\mathbf{C}$, $\mathbf{Set}^{\mathbf{C}^{\mathrm{op}}}$ is called the *category of presheaves* over $\mathbf{C}$.

As a cultural note, a category of presheaves is a topos: a cartesian closed category whose objects and morphisms carry a logical meaning, we say that a topos has an *internal logic*. The details on cartesian closed categories are found later in this chapter.

Example 1.19 (Yoneda embedding)**.** Given a category $\mathbf{C}$, the *Yoneda embedding* is a functor よ: $\mathbf{C} \to \mathbf{Set}^{\mathbf{C}^{\mathrm{op}}}$ (よ is the Japanese hiragana "yo" after 米田 信夫（よねだ のぶお）, Yoneda Nobuo) such that for all object X in $\mathbf{C}$, よ$(X) = \mathbf{C}(-, X)$ (see Example 1.14), and for all morphism $f\colon X \to Y$ in $\mathbf{C}$, よ(f) is a natural transformation $\mathbf{C}(-, X) \Rightarrow \mathbf{C}(-, Y)$ whose components are morphisms よ$(f)_Z\colon \mathbf{C}(Z, X) \to \mathbf{C}(Z, Y) :: g \mapsto f \circ g$ in $\mathbf{Set}$.

Definition 1.20 (Adjunction)**.** Given two categories $\mathbf{C}$ and $\mathbf{D}$, we say that two functors $F\colon \mathbf{C} \to \mathbf{D}$ and $G\colon \mathbf{D} \to \mathbf{C}$ are respectfully *left adjoint* and *right adjoint* if for all objects X in $\mathbf{C}$ and Y in $\mathbf{D}$, there is a bijection $\mathbf{D}(FX, Y) \cong \mathbf{C}(X, GY)$ that is natural in X and Y. An adjunction can be written with a diagram, as follows:



and is also written $F \dashv G$.

An adjunction also gives rise to two natural transformations:

- $\varepsilon\colon FG \Rightarrow \mathrm{id}_{\mathbf{D}}$, called the *counit*,
- $\eta\colon \mathrm{id}_{\mathbf{C}} \Rightarrow GF$, called the *unit*,

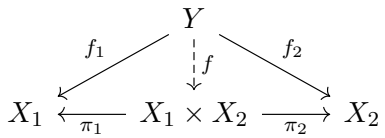
such that for every object X in $\mathbf{C}$ and every object Y in $\mathbf{D}$:

$$\mathrm{id}_{FX} = \varepsilon_{FX} \circ F(\eta_X), \qquad \mathrm{id}_{GY} = G(\varepsilon_Y) \circ \eta_{GY}.$$

Definition 1.21 (Initial and terminal object)**.** Given a category $\mathbf{C}$, an object X of $\mathbf{C}$ is said to be *initial* if for every object Y in $\mathbf{C}$, there is a unique morphism $X \to Y$. Conversely, an object X of $\mathbf{C}$ is said to be *terminal* if for every object Y in $\mathbf{C}$, there is a unique morphism $Y \to X$.

An initial object is often written 0, and a terminal object is often written 1. Moreover, given a terminal object 1 and any object X, we write $!_X$ for the unique morphism $X \to 1$.

Definition 1.22 (Product)**.** Given a category $\mathbf{C}$ and two objects X_1 and X_2 of $\mathbf{C}$, a *product* of X_1 and X_2 is an object of $\mathbf{C}$, usually written $X_1 \times X_2$, equipped with two morphisms $\pi_1\colon X_1 \times X_2 \to X_1$ and $\pi_2\colon X_1 \times X_2 \to X_2$, such that for every object Y and morphisms $f_1\colon Y \to X_1$ and $f_2\colon Y \to X_2$, there is a unique morphism $f\colon Y \to X_1 \times X_2$ such that the following diagram:



commutes. The unique morphism obtained is often written $\langle f_1, f_2 \rangle$.

Example 1.23. Given two categories $\mathbf{C}$ and $\mathbf{D}$, their product $\mathbf{C} \times \mathbf{D}$ is also a category.

Remark 1.24. Given a category $\mathbf{C}$ with products for any pair of objects, observe that for all objects X of $\mathbf{C}$, $- \times X\colon \mathbf{C} \to \mathbf{C}$ is a functor, as well as $X \times -\colon \mathbf{C} \to \mathbf{C}$. Actually, $- \times -\colon \mathbf{C} \times \mathbf{C} \to \mathbf{C}$ is a functor, and is commonly called a *bifunctor*.

1.1.1 Cartesian closed categories and λ-calculus

Cartesian closed categories are the main tool to study the semantics of functional classical programming languages – *classical* as opposed to *quantum*, which is one focus of this thesis. The adjective *cartesian* refers to the products, as introduced in Definition 1.22. The notion of closure is more subtle: one of the main properties is that function spaces are themselves objects of the category. Formally, a cartesian category $\mathbf{C}$ is closed if for all objects Y, the functor $- \times Y \colon \mathbf{C} \to \mathbf{C}$ admits a right adjoint $[Y \to -] \colon \mathbf{C} \to \mathbf{C}$, sometimes written $-^Y$. This adjunction embodies the notion of currying, meaning that a program $(A \times B) \to C$ is equivalent to a program $A \to (B \to C)$.

The author recommends reading the lecture notes of Awodey and Bauer [AB23] to have complete details about the topic of this section, and more about the link between logic, categories and programming languages.

Definition 1.25 (Cartesian closed category)**.** A *cartesian closed* category $\mathbf{C}$ is a category that has the following properties:

- $\mathbf{C}$ has a terminal object, usually written 1;
- for all pairs of objects X and Y, there is a product $X \times Y$ in $\mathbf{C}$;

such that for all objects Y, the assignment $(- \times Y) \colon \mathbf{C} \to \mathbf{C}$ is a left adjoint functor.

For all objects Y, we write $[Y \to -] \colon \mathbf{C} \to \mathbf{C}$ for the right adjoint of $(- \times Y) \colon \mathbf{C} \to \mathbf{C}$. Given a pair of objects X and Y, the object $[X \to Y]$ is called the *exponential*.

Example 1.26. The category $\mathbf{Set}$ of sets and functions between them is cartesian closed. Any singleton set is a terminal object. The product of two sets X and Y is the usual cartesian product $X \times Y$, which is the set:

$$\{(x, y) \mid x \in X, y \in Y\}$$

and the exponential of X and Y is the set of functions from X to Y, namely:

$$\{f \mid f \colon X \to Y\}.$$

There are many more examples of cartesian closed categories, such as the category of dcpos and Scott continuous functions, introduced later in §1.2.1.

Cartesian closed categories are remarkable because of their link with λ-calculi. The latter is a paradigm for computation, at the same level as Turing machines and recursive functions, and its raw presentation – the untyped λ-calculus – is known to represent all computable functions. In this thesis, we rather focus on typed λ-calculi, and typed programming languages in general, because of their link to logic and category theory. Next, we introduce briefly a simply-typed λ-calculus.

Simply-typed λ-calculus. First, we give a definition of the types, that are generated by the following grammar:

$$A ::= 1 \mid A \times A \mid A \to A$$

Note that, throughout the thesis, we might be less formal, writing for example:

$$A, B, \ldots ::= 1 \mid A \times B \mid A \to B \tag{1.1}$$

for readability; and the two definitions are to be regarded as identical. To define a language, one must also provide a set of terms, usually also introduced by a grammar. In our case, the terms of the simply-typed λ-calculus are given by:

$$M, N, \dots ::= x \mid * \mid \lambda x^A.M \mid MN \mid \langle M, N \rangle \mid \pi_i M \tag{1.2}$$

where x can range among a set of variables $\{x, y, z, \dots\}$, A is a type as introduced above and $i \in \{1, 2\}$. Note that variables can be free in a term, or bound by a λ-abstraction. A term without any free variables is called a *closed* term. In order to avoid conflicts between variables we will always work up to α-conversion and use Barendregt's convention [Bar84, p.26] which consists in keeping all bound and free variables names distinct.

The types allows us to formalise what a well-typed term is, through typing rules. A typing judgement is written $\Gamma \vdash M : A$, where M is a term of (1.2), A is a type of (1.1) and Γ is a context that contains variables each associated with a type $x_1 : A_1,\ x_2 : A_2, \dots,\ x_n : A_n$. The rule to form *correct* typing judgements are presented in a way that is usual in logic, *i.e.* with inference rules. Those typing rules in the case of the simply-typed λ-calculus are introduced in Figure 1.1.

$$\frac{}{\Gamma, x : A \vdash x : A} \qquad \frac{\Gamma \vdash M : A \to B \quad \Gamma \vdash N : A}{\Gamma \vdash MN : B}$$

$$\frac{}{\Gamma \vdash * : 1} \qquad \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x^A.M : A \to B} \qquad \frac{\Gamma \vdash M : A_1 \times A_2}{\Gamma \vdash \pi_i M : A_i}$$

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash N : B}{\Gamma \vdash \langle M, N \rangle : A \times B}$$

Figure 1.1 – Typing rules of the simply-typed λ-calculus.

One important concept in programming language design is substitution, which allows us to replace each occurrence of a free variable with a term of the syntax; and this term can also contain variables.

Definition 1.27 (Substitution)**.** Given two well-typed terms M, N of (1.2), we write $M[N/x]$ for the term where the free occurrences of x in M are replaced by N.

Whenever a term is introduced, it is important to show that it can be well-typed with the typing rules.

Lemma 1.28 ([Bar84])**.** *Given two well-typed terms* $\Gamma,\ x : A \vdash M : B$ *and* $\Gamma \vdash N : A$*, the judgement* $\Gamma \vdash M[N/x] : B$ *is valid.*

The computational behaviour of a programming language is formalised through an operational semantics. In a small step operational semantics, $M \to N$ informally means that the term M evaluates to N after one computational step. The most prominent rule of the λ-calculus is called β-reduction:

$$(\lambda x^A.M)N \to M[N/x]. \tag{1.3}$$

Equational Theory. Instead of working operationally, one can consider equations between terms. This new point of view loses information on the computational aspect of the language, but gains in convenience. The equational theory of the simply-typed λ-calculus is given in Figure 1.2.

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash M = M : A}\ (refl) \qquad \frac{\Gamma \vdash N = M : A}{\Gamma \vdash M = N : A}\ (symm)$$

$$\frac{\Gamma \vdash M = N : A \quad \Gamma \vdash N = P : A}{\Gamma \vdash M = P : A}\ (trans)$$

$$\frac{}{\Gamma, x : 1 \vdash * = x : A}\ (1.\eta) \qquad \frac{\Gamma \vdash M : A \quad \Gamma, x : A \vdash N = P : B}{\Gamma \vdash N[M/x] = P[M/x] : B}\ (subst)$$

$$\frac{\Gamma \vdash M = M' : A \quad \Gamma \vdash N = N' : B}{\Gamma \vdash \langle M, N\rangle = \langle M', N'\rangle : A \times B}\ (\langle,\rangle.eq) \qquad \frac{\Gamma \vdash M_1 : A_1 \quad \Gamma \vdash M_2 : A_2}{\Gamma \vdash \pi_i\langle M_1, M_2\rangle = M_i : A_i}\ (\times.\beta)$$

$$\frac{\Gamma \vdash M : A \times B}{\Gamma \vdash \langle \pi_1 M, \pi_2 M\rangle = M : A \times B}\ (\times.\eta)$$

$$\frac{\Gamma \vdash M = M' : A \to B \quad \Gamma \vdash N = N' : A}{\Gamma \vdash MN = M'N' : B}\ (app.eq)$$

$$\frac{\Gamma, x : A \vdash M = N : B}{\Gamma \vdash \lambda x^A.M = \lambda x^A.N : A \to B}\ (\lambda.eq) \qquad \frac{\Gamma, x : A \vdash M : B \quad \Gamma \vdash N : A}{\Gamma \vdash (\lambda x^A.M)N = M[N/x] : B}\ (\lambda.\beta)$$

$$\frac{\Gamma \vdash M : A \to B}{\Gamma \vdash \lambda x^A.Mx = M : A \to B}\ (\lambda.\eta) \qquad \frac{\Gamma \vdash M = N : B}{\Gamma, x : A \vdash M = N : B}\ (weak)$$

Figure 1.2 – Equational rules of the simply-typed λ-calculus.

Denotational semantics. The point of denotational semantics is to provide a mathematical interpretation to well-typed terms in a programming language to extract properties, design or a better understanding of the language. This requires first an interpretation for the types. Let us fix a cartesian closed category $\mathbf{C}$, and give a semantics to types as objects in $\mathbf{C}$. The semantics of types is defined by induction on their grammar:

$$[\![1]\!] = 1 \qquad [\![A \times B]\!] = [\![A]\!] \times [\![B]\!] \qquad [\![A \to B]\!] = [[\![A]\!] \to [\![B]\!]]$$

A context $\Gamma = x_1 : A_1, \ldots, x_n : A_n$ is interpreted as an object in $\mathbf{C}$ given by the product of the interpretations of all the types involved: $[\![A_1]\!] \times \cdots \times [\![A_n]\!]$.

Given a well-typed term $\Gamma \vdash M : A$ in the simply-typed λ-calculus, we write its denotational interpretation $[\![\Gamma \vdash M : A]\!]$. If we fix a cartesian closed category $\mathbf{C}$ (see Definition 1.25), the interpretation $[\![\Gamma \vdash M : A]\!]$ is given as a morphism in $\mathbf{C}$ from the interpretation of the context Γ to the interpretation of the type A. The interpretation of term judgements can then be defined by induction on the typing rules. The details can be found in Figure 1.3, where $\mathrm{curry}_{X,Y,Z}$ is the natural isomorphism $\mathbf{C}(X \times Y, Z) \cong \mathbf{C}(X, [Y \to Z])$ given by the adjunction $(- \times Y) \dashv [Y \to -]$, and $\mathrm{eval}_{X,Y} : [X \to Y] \times X \to Y$ is the counit of the adjunction.

Relationship between the semantics. So far, we have introduced some operational semantics, an equational theory and a denotational semantics to a simply-typed λ-calculus. However, we have yet to show what links them.

$$
\begin{aligned}
[\![\Gamma \vdash M : A]\!] &\in \mathbf{C}([\![\Gamma]\!], [\![A]\!]) \\
[\![\Gamma \vdash * : 1]\!] &= !_{[\![\Gamma]\!]} \\
[\![\Gamma, x : A \vdash x : A]\!] &= \pi_{[\![A]\!]} \\
[\![\Gamma \vdash MN : B]\!] &= \mathrm{eval}_{[\![A]\!],[\![B]\!]} \circ \langle [\![\Gamma \vdash M : A \to B]\!], [\![\Gamma \vdash N : A]\!] \rangle \\
[\![\Gamma \vdash \lambda x^A.M : A \to B]\!] &= \mathrm{curry}_{[\![\Gamma]\!],[\![A]\!],[\![B]\!]}([\![\Gamma, x : A \vdash M : B]\!]) \\
[\![\Gamma \vdash \pi_i M : A]\!] &= \pi_i \circ [\![\Gamma \vdash M : A_1 \times A_2]\!] \\
[\![\Gamma \vdash \langle M, N \rangle : A \times B]\!] &= \langle [\![\Gamma \vdash M : A]\!], [\![\Gamma \vdash N : B]\!] \rangle
\end{aligned}
$$

Figure 1.3 – Denotational semantics of terms in the simply-typed λ-calculus.

For example, we can state a *soundness* result between an operational semantics and the denotational semantics:

$$\text{Given } \Gamma \vdash M : A, \text{ if } M \to N, \text{ then } [\![\Gamma \vdash M : A]\!] = [\![\Gamma \vdash N : A]\!].$$

which is often simple to prove, by induction on the rules of the operational semantics. The converse is expected to be trickier, and is not necessarily true. The converse of a soundness statement is called *adequacy*.

In addition, a relationship between the equational theory and the denotational semantics can be given, called *completeness*:

$$\Gamma \vdash M = N : A \text{ if and only if } [\![\Gamma \vdash M : A]\!] = [\![\Gamma \vdash N : A]\!].$$

Soundness is often the minimal requirement to consider a category as a model of a specific language equipped with an operational semantics or an equational theory. Usually, adequacy or completeness is also expected; with this stronger property, the model can be used to study and to improve a programming language. For example, as shown is §1.2.1, if the language can be interpreted in a model that allows for fixed points, then fixed points can be safely added to the syntax.

1.1.2 Symmetric monoidal categories

We introduce *symmetric monoidal* categories, which are more general than cartesian categories. They have applications as models of linear logic and of quantum computing.

Definition 1.29. A *monoidal* category $\mathbf{C}$ is a category equipped with the following structure:

- a bifunctor $\otimes : \mathbf{C} \times \mathbf{C} \to \mathbf{C}$, called the *tensor product*;
- an object I called the *unit*;
- a natural isomorphism $\alpha_{X,Y,Z} : (X \otimes Y) \otimes Z \to X \otimes (Y \otimes Z)$, called the *associator*;
- a natural isomorphism $\lambda_X : I \otimes X \to X$, called the *left unitor*;
- a natural isomorphism $\rho_X : X \otimes I \to X$, called the *right unitor*;

such that, for all objects X, Y, Z and T, the two diagrams:

$$\begin{array}{ccccc}
((X \otimes Y) \otimes Z) \otimes T & \xrightarrow{\alpha_{X\otimes Y,Z,T}} & (X \otimes Y) \otimes (Z \otimes T) & \xrightarrow{\alpha_{X,Y,Z\otimes T}} & X \otimes (Y \otimes (Z \otimes T)) \\
\downarrow{\scriptstyle \alpha_{X,Y,Z}\otimes \mathrm{id}_T} & & & & \uparrow{\scriptstyle \mathrm{id}_X \otimes \alpha_{Y,Z,T}} \\
(X \otimes (Y \otimes Z)) \otimes T & & \xrightarrow{\qquad\qquad\alpha_{X,Y\otimes Z,T}\qquad\qquad} & & X \otimes ((Y \otimes Z) \otimes T)
\end{array}$$

$$\begin{array}{ccc}
(X \otimes I) \otimes Y & \xrightarrow{\alpha_{X,I,Y}} & X \otimes (I \otimes Y) \\
& {\scriptstyle \rho_X \otimes \mathrm{id}_Y} \searrow \quad \swarrow {\scriptstyle \mathrm{id}_X \otimes \lambda_Y} & \\
& X \otimes Y &
\end{array}$$

commute.

Diagrams such as the ones above are sometimes called *coherence conditions*, because they picture out loud conditions that one would expect to have.

Definition 1.30. A *symmetric monoidal* category $\mathbf{C}$ is a monoidal category equipped with a natural isomorphism $\sigma_{X,Y} \colon X \otimes Y \to Y \otimes X$, called the *symmetry* such that, for all objects X, Y and Z, the diagrams:

$$\begin{array}{ccc}
X \otimes I & \xrightarrow{\sigma_{X,I}} & I \otimes X \\
& {\scriptstyle \rho_X} \searrow \quad \swarrow {\scriptstyle \lambda_X} & \\
& X &
\end{array}$$

$$\begin{array}{ccc}
(X \otimes Y) \otimes Z & \xrightarrow{\sigma_{X,Y}\otimes \mathrm{id}_Z} & (Y \otimes X) \otimes Z \\
\downarrow{\scriptstyle \alpha_{X,Y,Z}} & & \downarrow{\scriptstyle \alpha_{Y,X,Z}} \\
X \otimes (Y \otimes Z) & & Y \otimes (X \otimes Z) \\
\downarrow{\scriptstyle \sigma_{X,Y\otimes Z}} & & \downarrow{\scriptstyle \mathrm{id}_Y \otimes \sigma_{X,Z}} \\
(Y \otimes Z) \otimes X & \xrightarrow{\alpha_{Y,Z,X}} & Y \otimes (Z \otimes X)
\end{array}$$

commute, and for all objects X and Y, $\sigma_{X,Y} \circ \sigma_{Y,X} = \mathrm{id}$.

Example 1.31. Any category with finite products (see Definition 1.22) is, in particular, a symmetric monoidal category. Note that the converse is not true.

The example above covers many instances of categories, such as $\mathbf{Set}$ or $\mathbf{Vect}$.

More examples of symmetric monoidal categories are given in the thesis. One noticeable difference between a cartesian product and a monoidal product, is that the monoidal one does not allow for copying in general. Indeed, with products, the morphism $\langle \mathrm{id}, \mathrm{id} \rangle \colon X \to X \times X$ necessarily exists, whereas there is in general no canonical morphism $X \to X \otimes X$ in a monoidal category. This hints at the fact that symmetric monoidal categories are the right tool to reason about a linear λ-calculus, where each variable is used exactly once.

1.1.3 Enriched categories

Categories in computer science are usually *locally small*, meaning that given two objects A and B, there is a *set* of morphisms $A \to B$. Enrichment is the study of the structure of those sets of morphisms, which could be vector spaces or topological spaces for example, more details can be found in [Kel65, Kel82, Mar65].

Definition 1.32 (Enriched Category)**.** Given a monoidal category $(\mathbf{V}, \otimes, I, \alpha, \lambda, \rho)$, a category $\mathbf{C}$ *enriched* in $\mathbf{V}$ (sometimes called a $\mathbf{V}$-category) is given by:

- a collection of objects of $\mathbf{C}$;
- an object $\mathbf{C}(X, Y)$ in $\mathbf{V}$ for all objects X and Y in $\mathbf{C}$;
- a morphism $\mathrm{id}_X \colon I \to \mathbf{C}(X, X)$ in $\mathbf{V}$, for all objects X in $\mathbf{C}$, and that is called the *identity*;
- a morphism $\mathrm{comp}_{X,Y,Z} \colon \mathbf{C}(Y, Z) \otimes \mathbf{C}(X, Y) \to \mathbf{C}(X, Z)$ in $\mathbf{V}$ for all objects X, Y and Z in $\mathbf{C}$, called the *composition*;

such that for all objects X, Y, Z and T, the following diagrams:

$$\begin{array}{ccc}
(\mathbf{C}(Z,T) \otimes \mathbf{C}(Y,Z)) \otimes \mathbf{C}(X,Y) & \xrightarrow{\mathrm{comp}_{Y,Z,T} \otimes \mathrm{id}} & \mathbf{C}(Y,T) \otimes \mathbf{C}(X,Y) \\
\downarrow{\scriptstyle \alpha} & & \downarrow{\scriptstyle \mathrm{comp}_{X,Y,T}} \\
\mathbf{C}(Z,T) \otimes (\mathbf{C}(Y,Z) \otimes \mathbf{C}(X,Y)) & & \mathbf{C}(X,T) \\
 & \xrightarrow{\mathrm{id} \otimes \mathrm{comp}_{X,Y,Z}} & \uparrow{\scriptstyle \mathrm{comp}_{X,Z,T}} \\
 & & \mathbf{C}(Z,T) \otimes \mathbf{C}(X,Z)
\end{array}$$

$$\begin{array}{ccc}
I \otimes \mathbf{C}(X,Y) & \xrightarrow{\mathrm{id}_Y \otimes \mathrm{id}_{\mathbf{C}(X,Y)}} & \mathbf{C}(Y,Y) \otimes \mathbf{C}(X,Y) \\
 & {\scriptstyle \rho_{\mathbf{C}(X,Y)}} \searrow \quad \swarrow {\scriptstyle \mathrm{comp}_{X,Y,Y}} & \\
 & \mathbf{C}(X,Y) &
\end{array}$$

$$\begin{array}{ccc}
\mathbf{C}(X,Y) \otimes I & \xrightarrow{\mathrm{id}_{\mathbf{C}(X,Y)} \otimes \mathrm{id}_X} & \mathbf{C}(X,Y) \otimes \mathbf{C}(X,X) \\
 & {\scriptstyle \lambda_{\mathbf{C}(X,Y)}} \searrow \quad \swarrow {\scriptstyle \mathrm{comp}_{X,X,Y}} & \\
 & \mathbf{C}(X,Y) &
\end{array}$$

commute.

Example 1.33. A locally small category is $\mathbf{Set}$-enriched. This is obtained directly with the definition of category and the facts that homsets are sets, thus they are objects of the category $\mathbf{Set}$. The coherence conditions are satisfied thanks to the associativity of composition in a category and to the axioms of the identity morphism.

Example 1.34. A cartesian closed category is enriched over itself. Indeed, composition is obtained by currying the following morphism:

$$[Y \to Z] \times [X \to Y] \times X \xrightarrow{\mathrm{id}_{[Y \to Z]} \times \mathrm{eval}_{X,Y}} [Y \to Z] \times Y \xrightarrow{\mathrm{eval}_{Y,Z}} Z$$

and it is routine to show that it verifies the coherence conditions.

Definition 1.35 (Enriched Functor)**.** Given two $\mathbf{V}$-enriched categories $\mathbf{C}$ and $\mathbf{D}$, a $\mathbf{V}$-enriched functor F maps every object X of $\mathbf{C}$ to an object of $\mathbf{D}$, written FX, and provides, for all objects X and Y in $\mathbf{C}$, a morphism $F_{X,Y} \colon \mathbf{C}(X,Y) \to \mathbf{D}(FX,FY)$ in $\mathbf{V}$ such that:

$$F_{X,X} \circ \mathrm{id}_X = \mathrm{id}_{FX} \qquad F_{X,Z} \circ \mathrm{comp}_{X,Y,Z} = \mathrm{comp}_{FX,FY,FZ} \circ (F_{Y,Z} \otimes F_{X,Y})$$

for all objects X, Y and Z in $\mathbf{C}$.

1.2 Fixed Points

This section introduces fixed point theorems that are relevant to this thesis; namely, fixed points in partially ordered sets, which allow for the interpretation of recursion and while loops, and initial algebras, which are a canonical tool to provide a semantics for inductive and recursive data types.

1.2.1 Dcpos

We work with the notion of partially ordered sets, usually called posets.

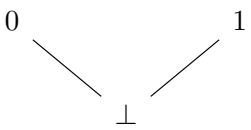
Definition 1.36 (Directed subset)**.** A non-empty subset D of a poset X is *directed* if every pair of elements in the subset D has an upper bound also in D.

Definition 1.37 (Dcpo)**.** A dcpo – short for *directed complete partial order* – is a poset X such that every directed subset $D \subseteq X$ has a supremum in X. A *pointed* dcpo $(X, \bot)$ is a dcpo X that has a least element, that we usually write $\bot$. If D is directed, we write $\sup D$ for its upper bound.

Example 1.38. The set of booleans $\mathbb{B} = \{0, 1\}$ with equality as an order is a dcpo. We can add a bottom element $\bot$, and we write $\mathbb{B}_\bot$ for the set $\{\bot, 0, 1\}$ with the following order:

$$\bot \leq 0 \qquad \bot \leq 1.$$

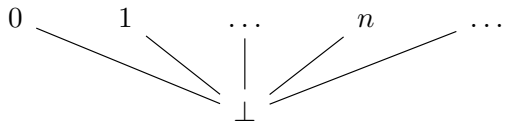
The partially ordered set $(\mathbb{B}_\bot, \leq)$ is a pointed dcpo. We say that the order is *flat*. A partial order is sometimes drawn for a better view of its behaviour. The dcpo $\mathbb{B}_\bot$ is then pictured as:



where $x \leq y$ iff there is a line between x and y and x is *below* y.

A common example of pointed dcpo used in the semantics of programming languages, *e.g.* PFC [Plo77], is the flat dcpo of natural numbers, also called by Plotkin *the standard collection of domains for arithmetic*. This is given in the next example.

Example 1.39. The flat dcpo of natural numbers $\mathbb{N}_\perp$ is given by:



Its directed subsets are simple: they are either sigletons or $\{\perp, n\}$ with $n \in \mathbb{N}$.

Next, we define functions between dcpos called *Scott continuous* after Dana Scott. He is famous for his contribution to *domain theory*, which encompasses all the content of this subsection about dpcos. A detailed account of domain theory can be found in [AJ95]. A continuous function needs first to be monotone – *i.e.*, given $x \leq y$, then $fx \leq fy$; what French speakers would prefer to call an *increasing* function.

Definition 1.40 (Scott continuous)**.** Given two dcpos X and Y, a monotone function $f\colon X \to Y$ is *Scott continuous* if for every directed subset $D \subseteq X$, $f(\sup D) = \sup f(D)$.

Example 1.41. The function $f\colon \mathbb{N}_\perp \to \mathbb{N}_\perp$ defined as:

$$f = \begin{cases} n & \mapsto & n+1 \\ \perp & \mapsto & \perp \end{cases}$$

is Scott continuous.

If X and Y are dcpos, then the set of Scott continuous functions $X \to Y$ is also a dcpo.

Note that dcpos and Scott continuous maps form a category $\mathbf{DCPO}$. The category of pointed dcpos and Scott continuous maps is written $\mathbf{DCPO}_\perp$. The category $\mathbf{DCPO}_\perp$ is cartesian closed, which means that it can be used for the denotational semantics of a λ-calculus.

Theorem 1.42 (Kleene's Fixed Point [SHLG94])**.** *If $(X, \perp)$ is a pointed dcpo and $f\colon X \to X$ is a Scott continuous function, then the function f has a least fixed point, obtained as* $\mathrm{fix} f = \sup\{f^n(\perp) \mid n \in \mathbb{N}\}$.

Recursion. We can add a new term to our λ-calculus to capture *recursion*. With the same types given in (1.1), we add to the grammar in (1.2) the following:

$$M, N, \ldots ::= \cdots \mid \texttt{fix}\ M \tag{1.4}$$

with the typing rule given below.

$$\frac{\Gamma \vdash M\colon A \to A}{\Gamma \vdash \texttt{fix}\ M\colon A}$$

The operational behaviour of this new term is as expected:

$$\texttt{fix}\ M \to M(\texttt{fix}\ M)$$

and its denotational semantics in $\mathbf{DCPO}_\perp$ is given by Kleene's fixed point (see Theorem 1.42), taken pointwise. Details can be found in the original paper by Plotkin [Plo77]. We can easily see that this semantics is sound, namely

$$[\![\texttt{fix}\ M]\!] = [\![M(\texttt{fix}\ M)]\!]\,.$$

1.2.2 Initial Algebras

Inductive data types are written in the syntax as some least fixed point of a type judgement, *e.g.* $\mu X.A$. As an example, the type of natural numbers is given by $\mu X.1 + X$ and the type of trees, whose nodes have type A, is $\mu X.1 + (X \times A \times X)$. As said earlier, types are represented as objects in the category; but handling inductive types means that we need to handle type variables too, and thus a type judgement $\Theta \vdash A$, where Θ is a set of type variables, is an *object mapping*, or rather a functor. We show below how to consider fixed points of functors in our categorical setting.

Definition 1.43 (Algebra)**.** Given a functor $F\colon \mathbf{C} \to \mathbf{C}$, a pair (X, f) composed of an object X and a morphism $f\colon FX \to X$ is a called an *F-algebra*. Given two F-algebras (X, f) and (Y, g), a morphism $h\colon X \to Y$ is an *F-algebra homomorphism* if the following diagram commutes:

$$\begin{array}{ccc} FX & \xrightarrow{f} & X \\ {\scriptstyle Fh}\downarrow & & \downarrow{\scriptstyle h} \\ FY & \xrightarrow{g} & Y \end{array}$$

When the category of F-algebras and F-algebra homomorphisms has an initial object, the latter is called the *initial F-algebra*.

> **Lemma 1.44** (Lambek's lemma [AMM18])**.** *Given an endofunctor $F : \mathbf{C} \to \mathbf{C}$ and an initial F-algebra $(X, \alpha\colon FX \to X)$, then α is an isomorphism.*

With Lambek's lemma, we know that an initial algebra provides an object X such that $X \cong FX$. Therefore we can see that the object X is a fixed point of the endofunctor F, as requested. However, we need a stonger notion of algebra for the denotational semantics of inductive types. Hence the next definition.

Definition 1.45 (Parameterised Initial Algebra)**.** Given two categories $\mathbf{C}$ and $\mathbf{D}$ and a functor $F\colon \mathbf{C} \times \mathbf{D} \to \mathbf{D}$, a *parameterised initial algebra* for F is a pair $(F^\dagger, \phi^F)$, such that:

- $F^\dagger\colon \mathbf{C} \to \mathbf{D}$ is a functor;
- $\phi^F\colon F \circ \langle \mathrm{id}, F^\dagger \rangle \Rightarrow F^\dagger\colon \mathbf{C} \to \mathbf{D}$ is a natural isomorphism;
- for every object X in $\mathbf{C}$, the pair $(F^\dagger, \phi^F_X)$ is an initial $F(X, -)$-algebra.

Remark 1.46. Observe that the previous definition with $\mathbf{C} = 1$, the category with one object and the identity, we recover Definition 1.43. The notion of parameterised initial algebra is then more general.

The existence of parameterised initial algebras is given by the theorem found in [Fio04, Corollary 7.2.4] and recalled in Theorem 1.50.

Definition 1.47. A category $\mathbf{C}$ is *parameterised $\mathbf{DCPO}$-algebraically complete* if all functors as described in Definition 1.45 admit a parameterised initial algebra.

In the following, we present sufficient conditions, outlined by Fiore in [Fio04], for a category to be $\mathbf{DCPO}$-algebraically complete.

Definition 1.48 (Ep-pair [Fio04]). Given a $\mathbf{DCPO}$-category $\mathbf{C}$, a morphism $e\colon X \to Y$ in $\mathbf{C}$ is called an *embedding* if there exists a morphism $p\colon Y \to X$ such that $p \circ e = \mathrm{id}_X$ and $e \circ p \leq \mathrm{id}_Y$. The morphisms e and p form an *embedding-projection pair* (e, p), also called *ep-pair*.

Remark 1.49. Similarly to embedding, a morphism p is called a *projector* if it is part of an ep-pair (e, p).

We recall that an *ep-zero* [Fio04, Definition 7.1.1] is an object 0 such that:

- 0 is an initial object;
- given any morphism $f\colon 0 \to Y$, f is an embedding;
- 0 is a terminal object;
- given any morphism $g\colon X \to 0$, g is a projector.

Theorem 1.50 ([Fio04]). *A $\mathbf{DCPO}$-category $\mathbf{C}$ with an ep-zero and colimits of ω-chains of embeddings is parameterised $\mathbf{DCPO}$-algebraically complete.*

Actually, a category that verifies the conditions above has stronger properties: it is parameterised $\mathbf{DCPO}$-algebraically ω-compact, namely it has parameterised initial algebras and parameterised final coalgebras for all $\mathbf{DCPO}$-functors. However, we do not need such a strong result in this thesis.

The latest theorem above means that any F as introduced in Definition 1.45 admits a parameterised initial algebra given that $\mathbf{C}$ is a $\mathbf{DCPO}$-category with an ep-zero and colimits of ω-chains of embeddings.

Inductive types. We take an example inspired from the metalanguage FPC [Gun92], with details in [Fio04, Chapter 8]. We are given the following types:

$$A, B \ ::= \ X \ \mid \ A + B \ \mid \ A \otimes B \ \mid \ \mid \ \mu X.A \tag{1.5}$$

with the typing rules:

$$\frac{}{\Theta, X \vdash X} \qquad \frac{\Theta \vdash A \quad \Theta \vdash B}{\Theta \vdash A \star B}\ \star \in \{+, \otimes\} \qquad \frac{\Theta, X \vdash A}{\Theta \vdash \mu X.A}$$

Their semantics is given in a symmetric monoidal $\mathbf{DCPO}$-enriched category $\mathbf{C}$ with coproducts that verifies the hypothesis of Theorem 1.50. The semantics of a type context Θ is $[\![\Theta]\!] = \mathbf{C}^{|\Theta|}$ and thus the interpretation of a type judgement $\Theta \vdash A$ is given by a functor $[\![\Theta \vdash A]\!] : \mathbf{C}^{|\Theta|} \to \mathbf{C}$. This interpretation is defined by induction on the typing rules, and the only non-trivial case is the fixed point constructor, whose semantics is given by:

$$[\![\Theta \vdash \mu X.A]\!] = [\![\Theta, X \vdash A]\!]^{\dagger}$$

where $(-)^{\dagger}$ is defined in Definition 1.45.

1.3 Restriction and Inverse Categories

A significant part of this thesis is the study of reversiblity in programming languages and in their semantics. This section is concerned with presenting the categorical tools in the literature

that formalise invertibility. Note that *reversible* does not mean *bijective*: a partial injection is reversible, in the sense that all of its possible outputs have a unique and deterministic corresponding input. Let us say more about partial injections. Given two sets X and Y, the image of a partial function f is given as follows: $f(x)$ if x is in the domain of f, and $\perp$ or *undefined* if x is not in the domain of f. A simple example would be the partial injection $f\colon \{0,1\} \to \{0,1\}$ such that $f(0) = 1$ and f is undefined on 1. Sets and partial injections form a category $\mathbf{PInj}$, which is the canonical example among restriction and inverse categories, the focus of this section.

For further reading, the author recommends the original work of Guo [Guo12], Giles [Gil14] and Kaarsgaard [Kaa17].

1.3.1 Basic structure

The axiomatisation of inverse categories gives the conditions for the morphisms of a category to be *partial injections*. First, the notion of restriction allows us to capture the *actual* domain of a morphism through a partial identity function. Historically, *inverse* categories [Kas79] were introduced before *restriction* categories, but the latter are more convenient to introduce the subject.

Definition 1.51 (Restriction [CL02])**.** A *restriction* structure is an operator that maps each morphism $f\colon X \to Y$ to a morphism $\overline{f}\colon X \to X$ such that for all $g\colon X \to Z$ and $h\colon Y \to T$, we have:

$$f \circ \overline{f} = f, \qquad \overline{f} \circ \overline{g} = \overline{g} \circ \overline{f},$$
$$\overline{f \circ \overline{g}} = \overline{f} \circ \overline{g}, \qquad \overline{h} \circ f = f \circ \overline{h \circ f}.$$

A morphism f is said to be *total* if $\overline{f} = \mathrm{id}_X$. A category with a restriction structure is called a *restriction category*.

Remark 1.52. Note that the definition implies that for all $f\colon X \to Y$, there is a unique $\overline{f}\colon X \to X$.

Example 1.53. Given two sets X and Y, any partial injection $f\colon X \to Y$ defined on a subset $X' \subseteq X$ and undefined on $X \setminus X'$ is given as follows:

$$\begin{cases} f(x) & \text{when } x \in X' \\ \text{undefined} & \text{when } x \notin X' \end{cases}$$

Then, the restriction of f, the morphism, $\overline{f}\colon X \to X$, is given by:

$$\begin{cases} x & \text{when } x \in X' \\ \text{undefined} & \text{when } x \notin X' \end{cases}$$

which is the identity on $X' \subseteq X$ and undefined on $X \setminus X'$. This example shows that $\mathbf{PInj}$ is a restriction category.

Definition 1.54 (Restriction Functor [CL02])**.** Given two restriction categories $\mathbf{C}$ and $\mathbf{D}$, a functor $F\colon \mathbf{C} \to \mathbf{D}$ is a *restriction functor* if $\overline{F(f)} = F\overline{f}$ for all morphism f of $\mathbf{C}$. The definition is canonically extended to bifunctors.

To interpret reversibility, we need to introduce a notion of reversed process, a process that exactly reverses another process. This is given by a generalised notion of inverse.

Definition 1.55 (Inverse category)**.** An *inverse category* is a restriction category where all morphisms are partial isomorphisms; meaning that for $f\colon X \to Y$, there exists $f^\circ\colon Y \to X$ such that $f^\circ \circ f = \overline{f}$ and $f \circ f^\circ = \overline{f^\circ}$, and is called the *partial inverse*.

Lemma 1.56. *In an inverse category, the partial inverse $f^\circ\colon Y \to X$ of a morphism $f\colon X \to Y$ is unique.*

Proof. Assume there exists two morphisms $g, h\colon : Y \to X$ such that

$$gf = \overline{f} \qquad fg = \overline{g} \qquad hf = \overline{f} \qquad fh = \overline{h}.$$

Therefore, we have $gf = \overline{f} = hf$. We have then:

$$\begin{aligned}
g\overline{h} &= gfh && \text{(hyp. above)}\\
&= hfh && \text{(observation above)}\\
&= \overline{f}h && \text{(hyp. above)}\\
&= h\overline{fh} && \text{(Def. 1.51, eq. 4)}\\
&= h\overline{\overline{h}} && \text{(hyp. above)}\\
&= h\overline{h}\\
&= h && \text{(Def. 1.51, eq. 1)}
\end{aligned}$$

Thus, $g\overline{h} = h$. Symmetrically, we can prove that $h\overline{g} = g$. We then obtain:

$$\begin{aligned}
\overline{h} &= \overline{g\overline{h}} && \text{(observation above)}\\
&= \overline{g}\overline{h} && \text{(Def. 1.51, eq. 3)}\\
&= \overline{h}\overline{g} && \text{(Def. 1.51, eq. 2)}\\
&= \overline{h\overline{g}} && \text{(Def. 1.51, eq. 3)}\\
&= \overline{g} && \text{(Def. 1.51, eq. 1)}
\end{aligned}$$

Finally, we have $h = g\overline{h} = g\overline{g} = g$. □

Remark 1.57. Given an inverse category $\mathbf{C}$, $(-)^\circ$ is actually a contravariant functor $\mathbf{C}^{\mathrm{op}} \to \mathbf{C}$. We can also observe that if $\mathbf{C}$ is an inverse category, then $\mathbf{C}^{\mathrm{op}}$ is also.

Example 1.58. In $\mathbf{PInj}$, let us consider the partial function $f\colon \{0,1\} \to \{0,1\}$ as $f(0) = 1$ and undefined on 1. Its restriction $\overline{f}$ is undefined on 1 also but $\overline{f}(0) = 0$. Its *inverse* f° is undefined on 0 and such that $f^\circ(1) = 0$.

The example above generalises and $\mathbf{PInj}$ is an actual inverse category. Even more, it is *the* inverse category: [Kas79] proves that every locally small inverse category is isomorphic to a subcategory of $\mathbf{PInj}$.

Definition 1.59 (Restriction compatible [Guo12]). Two morphisms $f, g\colon X \to Y$ in a restriction category $\mathbf{C}$ are *restriction compatible* if $f\overline{g} = g\overline{f}$. The relation is written $f \smile g$.

Example 1.60. We build upon Example 1.58, consider an additional partial injection $g\colon \{0,1\} \to \{0,1\}$ such that $g(0) = 1$ and $g(1) = 0$. We have $f \smile g$. The morphism g is *more defined* than f, and this intuition is made precise by the next definition.

Definition 1.61 (Partial order [CL02]). Let $f, g\colon X \to Y$ be two morphisms in a restriction category. We then define $f \leq g$ as $g\overline{f} = f$.

The next lemma – which is rather an observation – links the latest introduced notions of compatibility and order between maps in a restriction category.

> **Lemma 1.62.** *Given $\mathbf{C}$ a restriction category and two morphisms $f, g\colon X \to Y$ in $\mathbf{C}$, if $f \leq g$ then $f \smile g$.*

Proof. Remember that $f \leq g$ means that $g\overline{f} = f$. We can precompose by $\overline{g}$ and get:

$$\begin{aligned} f\overline{g} &= g\overline{f}\,\overline{g} \\ &= g\overline{g}\,\overline{f} && \text{(Def. 1.51, eq. 2)} \\ &= g\overline{f} && \text{(Def. 1.51, eq. 1)} \end{aligned}$$

and $f\overline{g} = g\overline{f}$ is the definition of compatibility. □

Definition 1.63 (Inverse compatible [KAG17]). Given $\mathbf{C}$ an inverse category, $f, g\colon X \to Y$ in $\mathbf{C}$ are inverse compatible if $f \smile g$ and $f^\circ \smile g^\circ$, noted $f \asymp g$.

Definition 1.64. A set S of morphisms of the same type $A \to B$ is restriction compatible (*resp.* inverse compatible) if all elements of S are pairwise restriction compatible (*resp.* inverse compatible).

This thesis makes use only of inverse categories, but note that most of the definitions below have a counterpart for restriction categories.

Definition 1.65 (Joins [Guo12]). An inverse category $\mathbf{C}$ is equipped with *joins* if for all inverse compatible sets S of morphisms $X \to Y$, there exists a morphism in $\mathbf{C}$ written $\bigvee_{s\in S} s\colon X \to Y$ such that, for all $t\colon X \to Y$ and for all $s \in S$, $s \leq t$, the following holds:

$$s \leq \bigvee_{s\in S} s, \qquad \bigvee_{s\in S} s \leq t, \qquad \overline{\bigvee_{s\in S} s} = \bigvee_{s\in S} \overline{s},$$

$$f \circ \left(\bigvee_{s\in S} s\right) = \bigvee_{s\in S} fs, \qquad \left(\bigvee_{s\in S} s\right) \circ g = \bigvee_{s\in S} sg.$$

Such a category is called a *join inverse category*.

Example 1.66. The category $\mathbf{PInj}$ is a join inverse category. Following Example 1.60, f and g are inverse compatible and $f \vee g = g$.

Building up from Definition 1.51, a *join restriction functor* is a restriction functor that preserves all thus constructed joins.

Remark 1.67 (Zero). Given a join inverse category $\mathbf{C}$, and since $\emptyset \subseteq \mathbf{C}(X,Y)$ with all of its elements that are inverse compatible, there exists a morphism $0_{X,Y} \doteq \bigvee_{s\in\emptyset} s \colon X \to Y$, called *zero map*. It satisfies the following equations, for all $f \colon Y \to Z$ and $g \colon Z \to X$:

$$f \circ 0_{X,Y} = 0_{X,Z} \qquad 0_{X,Y} \circ g = 0_{Z,Y} \qquad 0_{X,Y}^{\circ} = 0_{Y,X} \qquad \overline{0_{X,Y}} = 0_{X,X}.$$

Moreover, $0_{X,Y}$ is the least element in $\mathbf{C}(X,Y)$ for the order introduced above, in Definition 1.61.

Example 1.68. Given two sets X, Y, the morphism $0_{X,Y} \colon X \to Y$ mentioned above in $\mathbf{PInj}$ is the partial injection $X \to Y$ that is defined nowhere.

Lemma 1.69. *Given an inverse category $\mathbf{C}$ and a morphism $f \colon X \to Y$ such that $\overline{f} = 0_{X,X}$, then $f = 0_{X,Y}$.*

Proof. By Definition 1.51, we know that $f = f\overline{f}$, and thus $f = f\overline{f} = f0_{X,X} = 0_{X,Y}$. □

Lemma 1.70. *Given an inverse category $\mathbf{C}$ and two morphisms $f, g \colon X \to Y$ in $\mathbf{C}$ such that $f^{\circ}g = 0_{X,X}$ and $fg^{\circ} = 0_{Y,Y}$, then f and g are inverse compatible.*

Proof. Our goal is to prove that $f\overline{g} = g\overline{f}$, but we are going to prove that both are equal to zero. Definition 1.51 ensures that $\overline{f\overline{g}} = \overline{f}\overline{g}$, and then Definition 1.55 gives that $\overline{f\overline{g}} = f^{\circ}fg^{\circ}g$, which is then equal to $f^{\circ}0_{Y,Y}g$ by hypothesis, and thus $\overline{f\overline{g}} = \overline{f}\overline{g} = 0_{X,X}$; Lemma 1.69 ensures then that $f\overline{g} = 0_{X,Y}$. Note that $\overline{g\overline{f}} = \overline{f}\overline{g} = \overline{g}\overline{f}$ by Definition 1.51, thus $g\overline{f} = 0_{X,Y}$ too. We have proven that $f \smile g$. The proof that $f^{\circ} \smile g^{\circ}$ is similar. □

1.3.2 Additional Structure

Definition 1.71 (Restriction Zero)**.** An inverse category $\mathbf{C}$ has a *restriction zero* object 0 iff for all objects X and Y, there exists a unique morphism $0_{X,Y} \colon X \to Y$ that factors through 0 and satisfies $\overline{0_{X,Y}} = 0_{X,X}$.

Remark 1.72. In a join inverse category with a restriction zero, the zero morphisms given by the join structure (see Remark 1.67) coincide with the ones of the previous definition.

Lemma 1.73. *Given an inverse category $\mathbf{C}$ with a restriction zero 0 and that is $\mathbf{DCPO}$-enriched, 0 is an ep-zero.*

Proof. The restriction zero 0 is an initial and terminal object by definition. The pairs of embedding and projector are given by any morphism and its inverse $-^{\circ}$. □

Definition 1.74 (Disjointness tensor [Gil14])**.** A join inverse category $\mathbf{C}$ is said to have a *disjointness tensor* if it is equipped with a symmetric monoidal restriction bifunctor $(- \oplus -) \colon \mathbf{C} \times \mathbf{C} \to \mathbf{C}$, with as unit a restriction zero 0 and for all objects X and Y, morphisms $\iota_l \colon X \to X \oplus Y$ and $\iota_r \colon Y \to X \oplus Y$ that are total, jointly epic, and such that their inverses are jointly monic and $\overline{\iota_l^\circ}\ \overline{\iota_r^\circ} = 0_{X \oplus Y}$. The morphisms ι are called *injections*.

Remark 1.75. A precise way of writing the injections would be $\iota_l^{X,Y} \colon X \to X \oplus Y$ and $\iota_r^{X,Y} \colon Y \to X \oplus Y$. However, we choose to loosen the notations when it is not ambiguous. This choice is motivated by the recurrent use of the contravariant functor $(-)^\circ$, and ι_l° appears to be more readable than $(\iota_l^{X,Y})^\circ$.

The last requirement in the previous definition can be described as the injections having *orthogonal* outputs; with a set theoretic vocabulary, we could say that their images have an empty intersection. We show in the next lemma that this intuition is verified, with a simpler presentation than in Definition 1.74.

Lemma 1.76. *In a join inverse category $\mathbf{C}$ with disjointness tensor, for all objects X and Y, we have $\iota_l^\circ \circ \iota_r = 0_{Y,X}$.*

Proof. We know from Definition 1.74 that $\overline{\iota_l^\circ}\ \overline{\iota_r^\circ} = 0_{X \oplus Y}$, thus by postcomposition $\iota_l^\circ\ \overline{\iota_l^\circ}\ \overline{\iota_r^\circ} = 0_{X \oplus Y, X}$. Since $\mathbf{C}$ is a restriction category (see Definition 1.51), $\iota_l^\circ\ \overline{\iota_l^\circ} = \iota_l^\circ$, and thus $\iota_l^\circ\ \overline{\iota_r^\circ} = 0_{X \oplus Y, X}$. With Definition 1.55, we have $\overline{\iota_r^\circ} = \iota_r \iota_r^\circ$, thus $\iota_l^\circ \iota_r \iota_r^\circ = 0_{X \oplus Y, X}$. Finally, we precompose by ι_r, to obtain $\iota_l^\circ \iota_r \iota_r^\circ \iota_r = 0_{Y,X}$. Definition 1.55 again tells us that $\iota_r^\circ\ \iota_r = \overline{\iota_r}$, and also $\iota_r\ \overline{\iota_r} = \iota_r$, hence the equality: $\iota_l^\circ \iota_r = 0_{Y,X}$. □

This orthogonality between morphisms introduced by the disjointness tensor shows how pattern-matching could be handled. The author of this thesis has proven in [CLV21] that the disjointness tensor is not the only way of performing pattern-matching within an inverse category, however it remains the canonical way.

Remark 1.77. As we have started to describe the previous lemma as an *orthogonality* assertion, we are only a few inches away from defining an *inner product*, in a vector space fashion – although we are definitely not working with vector spaces. In the same scenery as Lemma 1.76, given two morphisms $f \colon X \to Z$ and $g \colon Y \to Z$, we allow ourselves to call the morphism $f^\circ \circ g \colon Y \to X$ the inner product of f and g, in a very loose way. Note that $g^\circ \circ f \colon X \to Y$ can also be said to be their inner product, and is not the same morphism in general.

A programming language usually involves pairs or tuples of terms, often denoted with a symmetric monoidal category. The next definition proposes a definition of a model for a (simple) reversible programming language handling pattern-matching.

Definition 1.78 (Rig)**.** Let us consider a join inverse category equipped with a symmetric monoidal tensor product $(\otimes, 1)$ and a disjointness tensor $(\oplus, 0)$ that are join preserving, and such that there are isomorphisms $\delta_{A,B,C} : A \otimes (B \oplus C) \to (A \otimes B) \oplus (A \otimes C)$ and $\nu_A \colon A \otimes 0 \to 0$. This is called a *join inverse rig category*.

It is proven in [KAG17] that a join inverse category can be considered enriched in $\mathbf{DCPO}$ without loss of generality, showing that there is a way of working with fixed points and general recursion in reversible settings. The next results prove that the operations involved in a reversible programming language preserve this $\mathbf{DCPO}$-enriched structure.

Lemma 1.79 ([KAG17]). *Let* $\mathbf{C}$ *and* $\mathbf{D}$ *be join inverse rig* $\mathbf{DCPO}$*-categories, and* $F\colon \mathbf{C} \to \mathbf{D}$ *be a join preserving restriction functor. Then* F *is a* $\mathbf{DCPO}$*-functor.*

A conclusion of this lemma is that the functors used to interpret simple data types in a programming language are $\mathbf{DCPO}$-functors.

Corollary 1.80. *Let* $\mathbf{C}$ *be a join inverse rig category. The functors* $(- \otimes -) : \mathbf{C} \times \mathbf{C} \to \mathbf{C}$ *and* $(- \oplus -) : \mathbf{C} \times \mathbf{C} \to \mathbf{C}$ *are* $\mathbf{DCPO}$*-functors.*

Separately, it is important that the inverse structure also preserves the enrichment.

Lemma 1.81 ([KAG17]). *Let* $\mathbf{C}$ *be a join inverse rig category. The functor* $(-)^\circ : \mathbf{C}^{op} \to \mathbf{C}$ *is a* $\mathbf{DCPO}$*-functor.*

Finally, we import a result from the literature ensuring that a join inverse rig category can safely be generalised to carry the interpretation of inductive types.

Proposition 1.82 ([KAG17]). *Any join inverse rig category can be faithfully embedded in a join inverse rig category with colimits of* ω*-chains of embeddings.*

This shows, with Lemma 1.73, that a join inverse rig $\mathbf{DCPO}$-category can verify the hypotheses of Theorem 1.50 whithout loss of generality; and thus this kind of category is a model for inductive types. The details of join inverse rig $\mathbf{DCPO}$-categories used as a denotational model is found in Chapter 4.

1.4 Hilbert spaces

One of the main focus in the study of mathematical quantum mechanics is Hilbert spaces. We assume basic knowledge of linear algebra, such as: vectors, linear maps, bases, kernels, *etc.* The author recommends the book written by Heunen and Vicary [HV19] for a more detailed introduction to Hilbert spaces, and to category theory applied to quantum computing.

1.4.1 Introductory Definitions

Formally, a Hilbert space is a complex vector space equipped with an inner product, written $\langle - \mid - \rangle$, such that this inner product induces a complete metric space. The inner product is used to compute probabilities of measurement outcomes in quantum theory. Note that, given a complex number α, we write $\overline{\alpha}$ for its conjugate.

Definition 1.83 (Inner product)**.** An inner product on a complex vector space V is a function $\langle - \mid - \rangle\colon V \times V \to \mathbb{C}$, such that:

- for all $x, y \in V$, $\langle x \mid y \rangle = \overline{\langle y \mid x \rangle}$;
- for all $x, y, z \in V$ and $\alpha \in \mathbb{C}$,

$$\langle x \mid \alpha y \rangle = \alpha \langle x \mid y \rangle, \qquad \langle x \mid y + z \rangle = \langle x \mid y \rangle + \langle x \mid z \rangle;$$

- for all $x \in V$, $\langle x \mid x \rangle \geq 0$, and if $\langle x \mid x \rangle = 0$, then $x = 0$.

Given a vector space with an inner product, the canonical norm of a vector x is defined as $\|x\| \stackrel{\text{def}}{=} \sqrt{\langle x \mid x \rangle}$.

This is enough to state the definition of a Hilbert space.

Definition 1.84 (Hilbert space)**.** A Hilbert space is a complex vector space H equipped with an inner product such that H is *complete* with regard to its canonical norm. By complete, we mean that: if a sequence of vectors $(v_i)_{i \in \mathbb{N}}$ is such that $\sum_{i=0}^{\infty} \|v_i\| < \infty$, then there exists a vector $v \in H$ such that $\|v - \sum_{i=0}^{\infty} v_i\|$ tends to zero as n goes to the infinity. The vector v is called a *limit*.

It is interesting to observe that all finite-dimentional vector spaces with an inner product are complete. Moreover, any vector space with an inner product can be *completed*, by adding the adequate limit vectors.

Remark 1.85. A basis in a Hilbert space is not exactly defined the same way as a basis in a vector space. A basis in a Hilbert space is such that any vector is *limit* of linear combinations of the elements of the basis. An orthonormal basis in a Hilbert space is such that its elements are pairwise orthogonal, have norm 1, and their linear span is dense in the Hilbert space.

Definition 1.86 (Bounded linear map)**.** Given two Hilbert spaces H_1 and H_2, a linear map $f \colon H_1 \to H_2$ is *bounded* if there exists $\alpha \in \mathbb{R}$ such that $\|fx\| \leq \alpha \|x\|$ for all $x \in H_1$.

1.4.2 Additional Structure

We make use of differents kinds of structure in vector spaces, such as direct sums and tensor products.

Definition 1.87 (Direct sum)**.** Given two complex vector spaces V and W, one can form their *direct sum* $V \oplus W$, whose elements are (v, w) with $v \in V$ and $w \in W$, such that, for all $v, v' \in V$ and $w, w' \in W$ and $\alpha, \beta \in \mathbb{C}$, $\alpha(v, w) + \beta(v', w') = (\alpha v + \beta v', \alpha w + \beta w')$.

Remark 1.88. $V \oplus \{0\}$ is isomorphic to V, and given $v \in V$, the vector $(v, 0)$ can be written v when there is no ambiguity. Given $v \in V$ and $w \in W$, the vector (v, w) can sometimes be written $v + w$.

Hilbert spaces are closed under direct sums, with the following inner product: $\langle (x_1, y_1) \mid (x_2, y_2) \rangle_{X \oplus Y} = \langle x_1 \mid x_2 \rangle_X + \langle y_1 \mid y_2 \rangle_Y$. However, this is not true for the tensor product: the *linear algebraic* tensor product of two Hilbert spaces is not necessarily a Hilbert space. We explain below how we can get around this issue with *completion*.

Definition 1.89 (Tensor product)**.** Given two complex vector spaces V, W, there is a vector space $V \otimes W$, together with a bilinear map $- \otimes - \colon V \times W \to V \otimes W :: (v, w) \mapsto v \otimes w$, such that for every bilinear map $h \colon V \times W \to Z$, there is a unique linear map $h' \colon V \otimes W \to Z$, such that $h = h' \circ \otimes$.

The tensor product of two Hilbert spaces X and Y is obtained through the tensor product of the underlying vector spaces, with the inner product $\langle x_1 \otimes y_1 \mid x_2 \otimes y_2 \rangle_{X \otimes Y} = \langle x_1 \mid x_2 \rangle_X \langle y_1 \mid y_2 \rangle_Y$ and then the completion of this space gives the desired Hilbert spaces. We abuse notation and write $X \otimes Y$ for the resulting Hilbert space.

The category of Hilbert spaces and bounded linear maps between them, written $\mathbf{Hilb}$, admits several different monoidal structures: with $(\otimes, \mathbb{C})$ and with $(\oplus, \{0\})$ – the latter is in fact a biproduct. They even give it a *rig* structure, in the sense of Definition 1.78. Classical computers operate on bits, while quantum computers apply operations on qubits, written $|0\rangle$ and $|1\rangle$. They are usually denoted as vectors in the Hilbert space $\mathbb{C} \oplus \mathbb{C}$ with $|0\rangle \stackrel{\text{def}}{=} (1, 0)$ and $|1\rangle \stackrel{\text{def}}{=} (0, 1)$ the elements of its canonical basis.

The next lemma outlines another important structure to Hilbert spaces, called the *adjoint*. It is due to Frigyes Riesz ['friɟɛʃ 'riːs].

Lemma 1.90 ([RSN55]). *Given a bounded linear map $f \colon H_1 \to H_2$ between Hilbert spaces, there is a unique bounded linear map $f^\dagger \colon H_2 \to H_1$ such that for all $x \in H_1$ and $y \in H_2$, $\langle f(x) \mid y \rangle = \langle x \mid f^\dagger(y) \rangle$. The map $f^\dagger$ is called the adjoint of f.*

The category $\mathbf{Hilb}$ is equipped with a structure of symmetric monoidal *dagger* category, meaning that $(-)^\dagger$ is an involutive contravariant endofunctor which is the identity on objects. Moreover, the dagger and the monoidal tensor respect some coherence conditions. For example, given two bounded linear maps $f \colon H_1 \to H_2$ and $g \colon H_1' \to H_2'$,

$$(f \otimes g)^\dagger = f^\dagger \otimes g^\dagger \colon H_2 \otimes H_2' \to H_1 \otimes H_1'.$$

Remark 1.91. Remember that throughout the thesis, given two maps $f \colon H_1 \to H_2$ and $g \colon H_2 \to H_3$, we sometimes write $gf \colon H_1 \to H_3$ for the composition $g \circ f \colon H_1 \to H_3$. In addition, given a complex number α and a map $f \colon H_1 \to H_2$, we write $\alpha f \colon H_1 \to H_2$ for the multiplication of the vector space $\alpha.f \colon H_1 \to H_2$.

Definition 1.92. Given a morphism $f \colon H_1 \to H_2$ in $\mathbf{Hilb}$, we say that f is:

- a unitary, if it is an isomophism and $f^{-1} = f^\dagger$;
- a contraction (or contractive map), if for all $x \in A$, $\|fx\| \leq \|x\|$;
- an isometry, if $f^\dagger f = \mathrm{id}$;
- a coisometry, if $f f^\dagger = \mathrm{id}$.

The category $\mathbf{Hilb}$ enjoys many properties, such as being its own opposite category and having a zero object – the zero space $\{0\}$. However, it is not monoidal closed. Besides, neither $\mathbf{Hilb}$ nor its subcategories obtained with the above definition are inverse categories, because projectors do not commute.

1.4.3 Quantum Computing

Quantum physics is the science of the infinitesimal. Its laws rule the world of small particules, in a way often described as hardly understandable to the macroscopic human intuition. In quantum theory, a particle can be in several states at the same time – this is called a *superposition* of states. This superposition holds as long as the particule is not *observed* – by

the operator by the environment. Once we have a look at the particule, it fixes itself in one particular state, with a certain probability.

This description can be simplified to the level of bits. Imagine that the two possible states are 0 or 1, usually written respectively $|0\rangle$ and $|1\rangle$. The general state of a *quantum* bit – or *qubit* – is given by the expression $\alpha|0\rangle + \beta|1\rangle$, where α and β are complex numbers. Once this qubit is observed, it becomes either 0 with probability $|\alpha|^2$, or 1 with probability $|\beta|^2$. Because 0 and 1 are the only two possibilities in this simple case, we need to have a proper probability distribution, and thus $|\alpha|^2 + |\beta|^2 = 1$. When this condition is verified, we say that the state $\alpha|0\rangle + \beta|1\rangle$ is *normalised*. There are several states that represent a qubit with an equal probability to be measured to 0 and 1; the most usual ones are $|+\rangle = \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$ and $|-\rangle = \frac{1}{\sqrt{2}}|0\rangle - \frac{1}{\sqrt{2}}|1\rangle$. When manipulating two qubits in states $|x\rangle$ and $|y\rangle$ in parallel, we write $|x\rangle \otimes |y\rangle$ or even $|xy\rangle$ for the resulting state.

Quantum computing is the science of performing operations on those qubits – and more generally, on a quantum superposition of data – to compute. The most traditional way of explicating a quantum algorithm is with a quantum circuit; the latter is the quantum generalisation of logical circuits. A quantum circuit is thus a sequence of quantum *logic gates* applied to a fixed number of qubits. Among quantum logic gates, one can find the `not` gate, similar to the classical one, the Hadamard gate which maps $|0\rangle$ to $|+\rangle$ and $|1\rangle$ to $|-\rangle$, rotation gates that map $|0\rangle$ to $|0\rangle$ and $|1\rangle$ to $e^{i\pi\theta}|1\rangle$, where θ is a real number, and the controlled `not` on two qubits, which applies the `not` to the second qubits when the first one is 1, else is the identity.

These operations are usually represented with complex matrices, and the states are given by vectors in finite-dimensional Hilbert spaces.

$$|0\rangle = \begin{pmatrix}1\\0\end{pmatrix} \qquad |1\rangle = \begin{pmatrix}0\\1\end{pmatrix} \qquad \alpha|0\rangle + \beta|1\rangle = \begin{pmatrix}\alpha\\\beta\end{pmatrix}$$

Multiples states in parallel, such as $|1\rangle \otimes |0\rangle$, are obtained with the usual tensor product. Thus the gates described above are the following:

$$\texttt{not} = \begin{pmatrix}0 & 1\\1 & 0\end{pmatrix} \qquad \texttt{had} = \begin{pmatrix}\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}}\\\frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}}\end{pmatrix} \qquad R_\theta = \begin{pmatrix}1 & 0\\0 & e^{i\pi\theta}\end{pmatrix} \qquad \texttt{cnot} = \begin{pmatrix}1 & 0 & 0 & 0\\0 & 1 & 0 & 0\\0 & 0 & 0 & 1\\0 & 0 & 1 & 0\end{pmatrix}$$

Any quantum algorithm can be expressed as a finite sequence of gates [NC02], and we say that quantum circuits are *universal*.

Another important notation in quantum computing besides $|\cdot\rangle$ is $\langle\cdot|$, which is obtained by taking the dagger:

$$\langle 0| = |0\rangle^\dagger = \begin{pmatrix}1 & 0\end{pmatrix} \qquad \langle 1| = |1\rangle^\dagger = \begin{pmatrix}0 & 1\end{pmatrix} \qquad \alpha\langle 0| + \beta\langle 1| = \alpha|0\rangle^\dagger + \beta|1\rangle^\dagger = \begin{pmatrix}\alpha & \beta\end{pmatrix}$$

Note that Hilbert spaces and unitaries (resp. contractions) form a dagger category. They are wide subcategories of $\mathbf{Hilb}$. Unitary maps are of central importance because they are the proper quantum operations, as solutions of the Schrödinger equation. One of the most significant unitary maps in quantum computing is the basis change in $\mathbb{C}\oplus\mathbb{C}$, also known as the *Hadamard gate*: $|0\rangle\langle+| + |1\rangle\langle-|$. We observe here that $|0\rangle\langle+|$ and $|1\rangle\langle-|$ are contractions,

and that unitary maps can be formulated as linear combination of compatible contractive maps. Note also that the *states*, like $|0\rangle$, are isometries.

We write $\mathbf{Contr}$, the category of *countably-dimensional* Hilbert spaces and contractive maps, $\mathbf{Isom}$, the category of *countably-dimensional* Hilbert and isometries between them, and $\mathbf{Coiso}$ the category of *countably-dimensional* Hilbert spaces and coisometries between them. The category of *countably-dimensional* Hilbert spaces and bounded linear maps is written $\mathbf{Hilb}^{\aleph_o}$ in this thesis.

Definition 1.93 (Zero map)**.** Given any pair of Hilbert spaces H_1 and H_2, we write $0_{H_1,H_2} : H_1 \to H_2$ for the linear map whose image is $\{0\}$ (we also write that $\mathrm{Ker}(0_{H_1,H_2}) = H_1$). When it is not ambiguous, we write $0 : H_1 \to H_2$. It is a contractive map.

Remark 1.94. Given a Hilbert space H, the morphism $0 : H \to \{0\}$ is unique for every H and makes $\{0\}$ a terminal object in $\mathbf{Contr}$ and in $\mathbf{Coiso}$.

Contractions are widely used in the literature for the denotational semantics of quantum programming languages [HK22a, AMHK23, CHKS23]. Some recent developments expose axioms for the categories involved in this thesis [HK22b, HKvdS22, MH24]. This better mathematical understanding of the category theory behind Hilbert spaces can only be beneficial for the theory of programming languages.

The ℓ^2 functor. As said in [Heu13], the ℓ^2 construction is the closest thing there is to a free Hilbert space. Given a set X, the following:

$$\ell^2(X) \stackrel{\text{def}}{=} \{\phi : X \to \mathbb{C} \mid \sum_{x \in X} |\phi(x)|^2 < \infty\} \tag{1.6}$$

is actually a Hilbert space. Even more, $\ell^2(-)$ is a functor $\mathbf{PInj} \to \mathbf{Hilb}$; given a morphism $f : X \to Y$ in $\mathbf{PInj}$, we have:

$$\ell^2(f)(\phi) = \phi \circ f^\circ.$$

This functor comes with many properties (see [Heu13]) except the one programming language theorists would want: it has no adjoints. Because of this, a *quantum effect* based on ℓ^2 cannot be studied as a usual computational effect (see §5.2.1 where this point is discussed, and see the next section §1.5 for the *usual* semantics of effects).

Given an element $x \in X$, its counterpart in $\ell^2(X)$ – in other words, the ϕ such that $\phi(x) = 1$ and for all $y \neq x$, $\phi(y) = 0$ – is written $|x\rangle$. The family $(|x\rangle)_{x \in X}$ is called the *canonical basis* of $\ell^2(X)$.

1.5 Monads

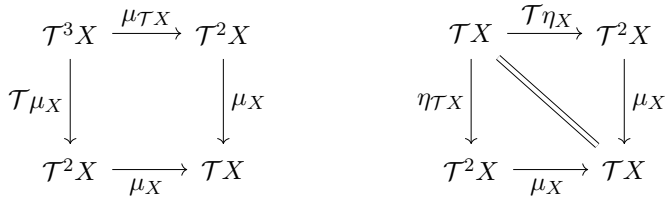
We introduce some background on strong and commutative monads and their premonoidal structure. Monads appear to be the most usual tool to interpret effects in a programming language. This is thanks to Moggi's work [Mog91, Mog89].

Monads are the generalisation of monoids in category theory, where the operation is the composition. One is likely to hear at least one the sentence "a monad is just a monoid in the category of endofunctors". While this sentence is correct, we introduce a bit more background to understand the later use of monads in Chapter 2.

1.5.1 Strong and Commutative Monads

We begin by recalling the definition of a monad.

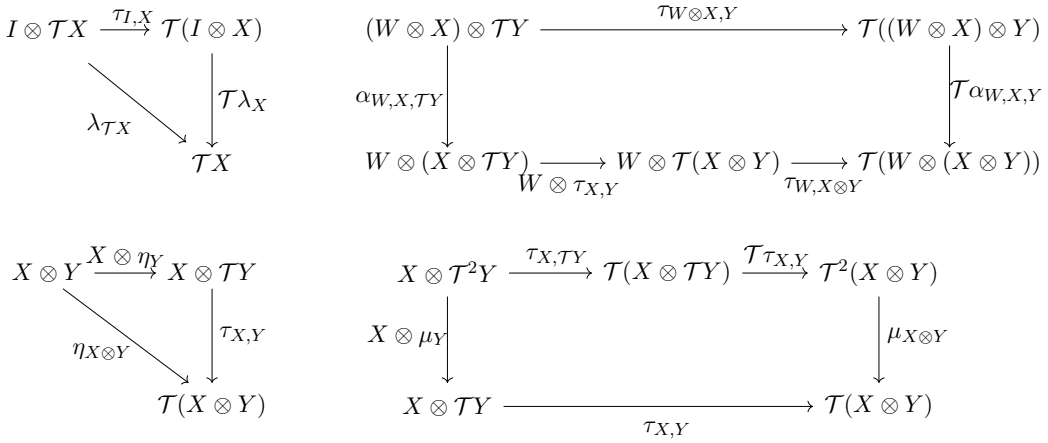
Definition 1.95 (Monad). A *monad* over a category $\mathbf{C}$ is an endofunctor $\mathcal{T}\colon \mathbf{C} \to \mathbf{C}$ equipped with two natural transformations $\eta\colon \mathrm{id} \Rightarrow \mathcal{T}$ and $\mu\colon \mathcal{T}^2 \Rightarrow \mathcal{T}$ such that the following diagrams



commute. We call η the *unit* of $\mathcal{T}$ and we say that μ is the *multiplication* of $\mathcal{T}$.

Next, we recall the definition of a *strong* monad, which is the main object of study in Chapter 2. As we already explained in the introduction, these monads are more computationally relevant (compared to non-strong ones) for most use cases. The additional structure, called the *monadic strength*, ensures the monad interacts appropriately with the monoidal structure of the base category.

Definition 1.96 (Strong Monad). A *strong monad* over a monoidal category $(\mathbf{C}, \otimes, I, \alpha, \lambda, \rho)$ is a monad $(\mathcal{T}, \eta, \mu)$ equipped with a natural transformation $\tau_{X,Y} : X \otimes \mathcal{T}Y \to \mathcal{T}(X \otimes Y)$, called *left strength*, such that the following diagrams commute:



We now recall the definition of a *commutative* monad which is of central importance here and in Chapter 2. Compared to a strong monad, a commutative monad enjoys even stronger coherence properties with respect to the monoidal structure of the base category (see also §1.5.3).

Definition 1.97 (Commutative Monad). Let $(\mathcal{T}, \eta, \mu, \tau)$ be a strong monad on a *symmetric* monoidal category $(\mathbf{C}, \otimes, I, \gamma)$. The *right strength* $\tau'_{X,Y}\colon \mathcal{T}X \otimes Y \to \mathcal{T}(X \otimes Y)$ of $\mathcal{T}$ is given

by the assignment $\tau'_{X,Y} \overset{\text{def}}{=} \mathcal{T}(\gamma_{Y,X}) \circ \tau_{Y,X} \circ \gamma_{\mathcal{T}X,Y}$. Then, $\mathcal{T}$ is said to be *commutative* if the following diagram commutes:

$$\begin{array}{ccccc} \mathcal{T}X \otimes \mathcal{T}Y & \xrightarrow{\tau_{\mathcal{T}X,Y}} & \mathcal{T}(\mathcal{T}X \otimes Y) & \xrightarrow{\mathcal{T}\tau'_{X,Y}} & \mathcal{T}^2(X \otimes Y) \\ \Big\downarrow{\scriptstyle \tau'_{X,\mathcal{T}Y}} & & & & \Big\downarrow{\scriptstyle \mu_{X\otimes Y}} \\ \mathcal{T}(X \otimes \mathcal{T}Y) & \xrightarrow[\mathcal{T}\tau_{X,Y}]{} & \mathcal{T}^2(X \otimes Y) & \xrightarrow[\mu_{X\otimes Y}]{} & \mathcal{T}(X \otimes Y) \end{array} \tag{1.7}$$

Remark 1.98. In the literature, the left and right strengths are sometimes called "strength" and "costrength" respectively.

Definition 1.99 (Morphism of Strong Monads [Jac16])**.** Given two strong monads $(\mathcal{T}, \eta^{\mathcal{T}}, \mu^{\mathcal{T}}, \tau^{\mathcal{T}})$ and $(\mathcal{P}, \eta^{\mathcal{P}}, \mu^{\mathcal{P}}, \tau^{\mathcal{P}})$ over a category $\mathbf{C}$, a *morphism of strong monads* is a natural transformation $\iota : \mathcal{T} \Rightarrow \mathcal{P}$ that makes the following diagrams commute:

$$\begin{array}{ccc} & X & \\ {\scriptstyle \eta^{\mathcal{T}}_X} \swarrow & & \searrow {\scriptstyle \eta^{\mathcal{P}}_X} \\ \mathcal{T}X & \xrightarrow[\iota_X]{} & \mathcal{P}X \end{array} \qquad \begin{array}{ccc} A \otimes \mathcal{T}B & \xrightarrow{A \otimes \iota_B} & A \otimes \mathcal{P}B \\ {\scriptstyle \tau^{\mathcal{T}}_{A,B}}\downarrow & & \downarrow{\scriptstyle \tau^{\mathcal{P}}_{A,B}} \\ \mathcal{T}(A \otimes B) & \xrightarrow[\iota_{A\otimes B}]{} & \mathcal{P}(A \otimes B) \end{array}$$

$$\begin{array}{ccccc} \mathcal{T}^2X & \xrightarrow{\iota_{\mathcal{T}X}} & \mathcal{P}\mathcal{T}X & \xrightarrow{\mathcal{P}\iota_X} & \mathcal{P}^2X \\ {\scriptstyle \mu^{\mathcal{T}}_X}\downarrow & & & & \downarrow{\scriptstyle \mu^{\mathcal{P}}_X} \\ \mathcal{T}X & & \xrightarrow[\iota_X]{} & & \mathcal{P}X \end{array}$$

Strong monads over a (symmetric) monoidal category $\mathbf{C}$ and strong monad morphisms between them form a category which we denote by writing $\mathbf{StrMnd}(\mathbf{C})$. In the situation of Definition 1.99, if ι is a monomorphism in $\mathbf{StrMnd}(\mathbf{C})$, then $\mathcal{T}$ is said to be a *strong submonad* of $\mathcal{P}$ and ι is said to be a *strong submonad morphism*.

Definition 1.100 (Kleisli category)**.** Given a monad $(\mathcal{T}, \eta, \mu)$ over a category $\mathbf{C}$, the *Kleisli category* $\mathbf{C}_{\mathcal{T}}$ of $\mathcal{T}$ is the category whose objects are the same as those of $\mathbf{C}$, but whose morphisms are given by $\mathbf{C}_{\mathcal{T}}(X, Y) = \mathbf{C}(X, \mathcal{T}Y)$. Composition in $\mathbf{C}_{\mathcal{T}}$ is given by $g \odot f \overset{\text{def}}{=} \mu_Z \circ \mathcal{T}g \circ f$ where $f : X \to \mathcal{T}Y$ and $g : Y \to \mathcal{T}Z$. The identity at X is given by the monadic unit $\eta_X : X \to \mathcal{T}X$.

Proposition 1.101 ([Jac16])**.** *If $\iota : \mathcal{T} \Rightarrow \mathcal{P}$ is a submonad morphism, then the functor $\mathcal{I} : \mathbf{C}_{\mathcal{T}} \to \mathbf{C}_{\mathcal{P}}$, defined by $\mathcal{I}(X) = X$ on objects and $\mathcal{I}(f : X \to \mathcal{T}Y) = \iota_Y \circ f : X \to \mathcal{P}Y$ on morphisms, is an embedding of categories.*

The functor $\mathcal{I}$ above is the canonical embedding of $\mathbf{C}_{\mathcal{T}}$ into $\mathbf{C}_{\mathcal{P}}$ induced by the submonad morphism $\iota : \mathcal{T} \Rightarrow \mathcal{P}$.

1.5.2 Semantics of the λ-calculus with effects

We present here a brief summary of the work by Moggi [Mog91, Mog89] on computational effects. This work has been very influencial, resulting in the development of the programming language Haskell, among others.

The grammar and typing rules for Moggi's metalanguage is found in Figure 1.4. Compared to the simply-typed λ-calculus, a type construction $\mathcal{T}(-)$ is added. Given a type A, the type $\mathcal{T}A$ is called a *monadic* type, and represents the computational effects of type A allowed in the language. The $\texttt{ret}$ constructor is to be seen as an introduction rule for monadic types, and embodies the fact that a *pure* or *non-effectful* computation can be seen as a monadic computation, but with no effect. The $\texttt{do}$ operation performs the sequencing of monadic computations.

The equational theory for monadic types, added on top of the equational theory for the simply-typed λ-calculus presented in Figure 1.2, is found in Figure 1.5.

$$\text{(Types)} \quad A, B \quad ::= 1 \mid A \to B \mid A \times B \mid \mathcal{T}A$$

$$\text{(Terms)} \quad M, N \quad ::= x \mid * \mid \lambda x^A.M \mid MN \mid \langle M, N\rangle \mid \pi_i M \mid \texttt{ret}\ M \mid \texttt{do}\ x \leftarrow M;\ N$$

$$\frac{}{\Gamma, x\colon A \vdash x\colon A} \qquad \frac{\Gamma \vdash M\colon A \to B \quad \Gamma \vdash N\colon A}{\Gamma \vdash MN\colon B}$$

$$\frac{}{\Gamma \vdash *\colon 1} \qquad \frac{\Gamma, x\colon A \vdash M\colon B}{\Gamma \vdash \lambda x^A.M\colon A \to B} \qquad \frac{\Gamma \vdash M\colon A_1 \times A_2}{\Gamma \vdash \pi_i M\colon A_i}$$

$$\frac{\Gamma \vdash M\colon A \quad \Gamma \vdash N\colon B}{\Gamma \vdash \langle M, N\rangle\colon A \times B} \qquad \frac{\Gamma \vdash M\colon A}{\Gamma \vdash \texttt{ret}\ M\colon \mathcal{T}A} \qquad \frac{\Gamma \vdash M\colon \mathcal{T}A \quad \Gamma, x\colon A \vdash N\colon \mathcal{T}B}{\Gamma \vdash \texttt{do}\ x \leftarrow M;\ N\colon \mathcal{T}B}$$

Figure 1.4 – Grammars and typing rules.

$$\frac{\Gamma \vdash M = N\colon A}{\Gamma \vdash \texttt{ret}\ M = \texttt{ret}\ N\colon \mathcal{T}A}\ (ret.eq) \qquad \frac{\Gamma \vdash M = M'\colon \mathcal{T}A \quad \Gamma, x\colon A \vdash N = N'\colon \mathcal{T}B}{\Gamma \vdash \texttt{do}\ x \leftarrow M;\ N = \texttt{do}\ x \leftarrow M';\ N'\colon \mathcal{T}B}\ (do.eq)$$

$$\frac{\Gamma \vdash M\colon A \quad \Gamma, x\colon A \vdash N\colon \mathcal{T}B}{\Gamma \vdash \texttt{do}\ x \leftarrow \texttt{ret}\ M;\ N = N[M/x]\colon \mathcal{T}B}\ (\mathcal{T}.\beta) \qquad \frac{\Gamma \vdash M\colon \mathcal{T}A}{\Gamma \vdash \texttt{do}\ x \leftarrow M;\ \texttt{ret}\ x = M\colon \mathcal{T}A}\ (\mathcal{T}.\eta)$$

Figure 1.5 – Moggi's equational rules for terms of monadic types.

Denotational semantics. The denotational semantics of Moggi's metalanguage is obtained in a cartesian closed category $\mathbf{C}$ equipped with a strong monad $\mathcal{T}$. Pure computations shall still be interpreted in as morphisms in the category $\mathbf{C}$; while monadic computations, *e.g.* $\Gamma \vdash M\colon \mathcal{T}A$, are interpreted as morphisms $[\![\Gamma]\!] \to \mathcal{T}\,[\![A]\!]$, thus living in the Kleisli category of $\mathcal{T}$. The interpretation of $\texttt{ret}$ is given by the unit of the monad $\mathcal{T}$, and the interpretation of of $\texttt{do}$ is obtained with the composition in the Kleisli category.

1.5.3 Premonoidal Structure of Strong Monads

Let $\mathcal{T}$ be a strong monad on a symmetric monoidal category $(\mathbf{C}, I, \otimes)$. Then, its Kleisli category $\mathbf{C}_\mathcal{T}$ does *not* necessarily have a canonical monoidal structure. However, it does have a canonical *premonoidal structure* as shown by Power and Robinson [PR97]. In fact, they show that this premonoidal structure is monoidal iff the monad $\mathcal{T}$ is commutative. Next, we briefly recall the premonoidal structure of $\mathbf{C}_\mathcal{T}$ as outlined by them.

For every two objects X and Y of $\mathbf{C}_\mathcal{T}$, their tensor product $X \otimes Y$ is also an object of $\mathbf{C}_\mathcal{T}$, but the monoidal product $\otimes$ of $\mathbf{C}$ does not necessarily induce a bifunctor on $\mathbf{C}_\mathcal{T}$. However, by using the left and right strengths of $\mathcal{T}$, we can define two families of functors as follows:

- for any object X, a functor $(- \otimes_l X) \colon \mathbf{C}_\mathcal{T} \to \mathbf{C}_\mathcal{T}$ whose action on objects sends Y to $Y \otimes X$, and sends $f : Y \to \mathcal{T}Z$ to $\tau'_{Z,X} \circ (f \otimes X) : Y \otimes X \to \mathcal{T}(Z \otimes X)$;
- for any object X, a functor $(X \otimes_r -) \colon \mathbf{C}_\mathcal{T} \to \mathbf{C}_\mathcal{T}$ whose action on objects sends Y to $X \otimes Y$, and sends $f : Y \to \mathcal{T}Z$ to $\tau_{X,Z} \circ (X \otimes f) : X \otimes Y \to \mathcal{T}(X \otimes Z)$.

This categorical data satisfies the axioms and coherence properties of *premonoidal categories* as explained in [PR97], but which we omit here because it is not essential for the development of our results. What is important is to note that in a premonoidal category, $f \otimes_l X'$ and $X \otimes_r g$ do not always commute. This leads us to the following definition, which plays a crucial role in the theory of premonoidal categories and has important links to our development.

Definition 1.102 (Premonoidal Centre [PR97])**.** Given a strong monad $(\mathcal{T}, \eta, \mu, \tau)$ on a symmetric monoidal category $(\mathbf{C}, I, \otimes)$, we say that a morphism $f : X \to Y$ in $\mathbf{C}_\mathcal{T}$ is *central* if for any morphism $f' : X' \to Y'$ in $\mathbf{C}_\mathcal{T}$, the diagram

$$\begin{array}{ccc} X \otimes X' & \xrightarrow{f \otimes_l X'} & Y \otimes X' \\ {\scriptstyle X \otimes_r f'}\downarrow & & \downarrow{\scriptstyle Y \otimes_r f'} \\ X \otimes Y' & \xrightarrow[f \otimes_l Y']{} & Y \otimes Y' \end{array}$$

commutes in $\mathbf{C}_\mathcal{T}$. The *premonoidal centre* of $\mathbf{C}_\mathcal{T}$ is the subcategory $Z(\mathbf{C}_\mathcal{T})$ which has the same objects as those of $\mathbf{C}_\mathcal{T}$ and whose morphisms are the central morphisms of $\mathbf{C}_\mathcal{T}$.

In [PR97], the authors prove that $Z(\mathbf{C}_\mathcal{T})$, is a symmetric *monoidal* subcategory of $\mathbf{C}_\mathcal{T}$. In particular, this means that Kleisli composition and the tensor functors $(- \otimes_l X)$ and $(X \otimes_r -)$ preserve central morphisms. However, it does not necessarily hold that the subcategory $Z(\mathbf{C}_\mathcal{T})$ is the Kleisli category for a monad over $\mathbf{C}$. Nevertheless, in this situation, the left adjoint of the Kleisli adjunction $\mathcal{J} \colon \mathbf{C} \to \mathbf{C}_\mathcal{T}$ always corestricts to $Z(\mathbf{C}_\mathcal{T})$. We write $\hat{\mathcal{J}} \colon \mathbf{C} \to Z(\mathbf{C}_\mathcal{T})$ to indicate this corestriction (which need not be a left adjoint).

Remark 1.103. In [PR97], the subcategory $Z(\mathbf{C}_\mathcal{T})$ is called the centre of $\mathbf{C}_\mathcal{T}$. However, we refer to it as the *premonoidal centre* of a premonoidal category to avoid confusion with the new notion of the centre of a monad that we introduce next. In the sequel, we show that the two notions are very strongly related to each other (Theorem 2.11).

Chapter 2
Monads and Commutativity

"Everyone likes monads." — Nima Motamed.

Abstract

Monads in category theory are algebraic structures that can be used to model computational effects in programming languages. We show how the notion of "*centre*", and more generally "*centrality*", *i.e.*, the property for an effect to commute with all other effects, may be formulated for strong monads acting on symmetric monoidal categories. We identify three equivalent conditions which characterise the existence of the centre of a strong monad (some of which relate it to the premonoidal centre of Power and Robinson) and we show that every strong monad on many well-known naturally occurring categories does admit a centre, thereby showing that this new notion is ubiquitous. More generally, we study *central submonads*, which are necessarily commutative, just like the centre of a strong monad. We provide a computational interpretation by formulating equational theories of lambda calculi equipped with central submonads, we describe categorical models for these theories and prove soundness, completeness and internal language results for our semantics.

References. This chapter, apart from some additions made by the author, is a paper [CLZ23] presented at LICS'2023, and coauthered with Titouan Carette and Vladimir Zamdzhiev.

2.1 Introduction

The importance of monads in programming semantics has been demonstrated in seminal work by Moggi [Mog89, Mog91]. The main idea is that monads allow us to introduce computational effects (e.g., state, input/output, recursion, probability, continuations) into pure type systems in a controlled way. The mathematical development surrounding monads has been very successful and it directly influenced modern programming language design through the

introduction of monads as a programming abstraction into languages such as Haskell, Scala and others (see [Ben15]). Inspired by this, we follow in the same spirit: we start with a mathematical question about monads, we provide the answer to it and we present a computational interpretation. The mathematical question that we ask is simple and it is inspired by the theory of monoids and groups:

Is there a suitable notion of "centre" that may be formulated for monads and what is a "central" submonad?

We show that, just as every monoid M (on $\mathbf{Set}$) has a centre, which is a *commutative* submonoid of M, so does every (canonically strong) monad $\mathcal{T}$ on $\mathbf{Set}$ and the centre of $\mathcal{T}$ is a *commutative* submonad of $\mathcal{T}$ (§2.2.1). A central [1] submonad of $\mathcal{T}$ is simply a submonad of the centre of $\mathcal{T}$ (Definition 2.30) and the analogy to the case of monoids and groups is completely preserved. Note that our construction has nothing to do with the folklore characterisation of monads as monoid objects in a functor category, wherein the notion of commutativity is unclear. The relevant analogy with monoids in $\mathbf{Set}$ is fully explained in Example 2.12. Generalising away from the category $\mathbf{Set}$, the answer is a little bit more complicated: not every monoid object M on a symmetric monoidal category $\mathbf{C}$ has a centre, and neither does every strong monad on $\mathbf{C}$ (§2.2.3). However, we show that under some reasonable assumptions, the centre does exist (Theorem 2.11) and we have not found any naturally occurring monads in the literature that are not centralisable (*i.e.*, monads other than the artificially constructed one we used as a counter-example). Furthermore, we show that for many categories of interest, all strong monads on them are centralisable (§2.3.1) and we demonstrate that the notion of centre is ubiquitous. The centre of a strong monad satisfies interesting universal properties (Theorem 2.11) which may be equivalently formulated in terms of our novel notion of *central cone* or via the *premonoidal centre* of Power and Robinson [PR97]. The notion of a central submonad is more general and it may be defined without using the centre. When the centre exists, a central submonad may be equivalently defined as a strong submonad of the centre (Theorem 2.29).

The computational significance of these ideas is easy to understand: given an effect, modelled by a strong monad, such that perhaps not *every* pair of effectful operations commute (*i.e.*, the order of monadic sequencing matters), identify only those effectful operations which do commute with any other possible effectful operation. The effectful operations that satisfy this property are called *central*. When the monad is centralisable, the collection of *all* central operations determine the centre of the monad (which is a commutative submonad). *Any* collection of central operations that may be organised into a strong submonad determines a central submonad (which also is commutative). We argue that central submonads have greater computational significance compared to the centre of a strong monad (§2.5.2) for two main reasons: (1) central submonads are strictly more general; (2) central submonads have a simpler and considerably more practical axiomatisation via an equational theory, whereas the centre of a monad requires an axiomatisation using a more complicated logical theory. We cement our categorical semantics by proving soundness, completeness and internal language results (See [MMDPR05] for a convincing argument why internal language results are important and why soundness and completeness *alone* might not be sufficient).

1. Given a group G, a *central subgroup* is a subgroup of the centre of G, equivalently, a subgroup whose elements commute with every element of G.

2.1.1 Related Work

A notion of commutants for enriched algebraic theories has been defined in [Luc18] from which the author derives a notion of centre of an enriched algebraic theory. In the case of enriched monads, in other words, strong monads arising from enriched algebraic theories, their notion of commutant extends to monad morphisms. While not explicitly stated in the paper, applying the commutant construction on the identity monad morphism from a monad to itself provides a notion of centre of a monad that appears to coincide with ours. However, enriched algebraic theories correspond to $\mathcal{J}$-ary $\mathcal{V}$-enriched monads (See [Luc18] for a definition of $\mathcal{J}$-ary monads w.r.t. a system of arities $\mathcal{J}$) on a symmetric monoidal *closed* category $\mathcal{V}$ (equivalently $\mathcal{J}$-ary strong monads on $\mathcal{V}$). In this chapter, we show that monoidal closure of $\mathcal{V}$ is not necessary to define the centre and neither is the $\mathcal{J}$-ary assumption on the monad. Other related work [GF16] considers a very general notion of commutativity in terms of certain kinds of duoidal categories. As a special case of their treatment, the authors are able to recover the commutativity of bistrong monads and with some additional effort (not outlined in the paper), it is possible to construct the centre of a bistrong monad acting on a monoidal *biclosed* category. Our construction of the centre appears to coincide with theirs in the special case of strong monads defined on symmetric monoidal *closed* categories, but as discussed above, our method does not require any kind of closure of the category. Therefore, compared to both works [GF16, Luc18], as far as symmetric monoidal (not necessarily closed) categories are concerned, our methods can be used to construct the centre for a larger class of strong monads and we establish our main results, together with our universal characterisation of the centre, under these assumptions. Furthermore, we also place a heavy emphasis on *central* submonads in this chapter and these kinds of monads are not discussed in either of these works and neither is there a computational interpretation (which is our main result in §2.5).

Another related work is [PR97], which introduces premonoidal categories. We have established important links between our development and the premonoidal centre (Theorem 2.11). While premonoidal categories have been influential in our understanding of effectful computation, it was less clear (to us) how to formulate an appropriate computational interpretation of the premonoidal centre for higher-order languages. We show that under some mild assumptions (which are easily satisfied see §2.3), the premonoidal centre of the Kleisli category of a strong monad induces an adjunction into the base category (Theorem 2.11) and this allows us to formulate a suitable computational interpretation by using monads, which are already well-understood [Mog91, Mog89] and well-integrated into many programming languages [Ben15].

Staton and Levy introduce the novel notion of *premulticategories* [SL13] in order to axiomatise impure/effectful computation in programming languages. The notion of centrality plays an important role in the development of the theory there as well. However, they do not focus, as we do, on providing suitable programming abstractions that identify both central and non-central computations (e.g., by separating them into different types like us) and from what we can tell from our reading, there are no universal properties stated for the collection of central morphisms. Also, our results provide a computational interpretation in terms of monads, which are standard and well-understood, so it is easier to incorporate them into existing languages.

Central morphisms in the context of computational effects have been studied in [Fü99], among other sorts of *varieties* of morphisms: thunkable, copyable, and discardable. The author links their notion of central morphisms with the ones from the premonoidal centre in Power and

Robinson [PR97], and also proves under some conditions that those varieties form a subcategory with similar properties to the original category. However, they do not mention that a central submonad or a centre can be constructed out of those central morphisms. More generally, the fact that monads could be derived from those varieties is not studied at all in that paper.

The work in [Fü99] has an impact in [MM22], where a Galois connection is established between call-by-value and call-by-name. In that paper, the order in which operations are done matters, and central computations are mentioned. Again, the central computations are not linked to submonads in there.

2.1.2 Work of the Author

The author has contributed to the following points.

- Equivalent characterisations for a monad to be centralisable (see Theorem 2.11).
- Subsection 2.3.3, which details the link with Lawvere theories.
- A language for central submonads, inspired from Moggi's metalanguage.
- A notion of equational theories for such a language.
- A completeness and internal language result, linking the categorical model to a syntactic notion of central submonad.

Compared to the paper [CLZ23], the proofs produced by the author are added, as well as a section (see §2.3.3) in which we show the link with Lawvere theories.

2.2 The Centre of a Strong Monad

We begin by showing that any (necessarily strong) monad on $\mathbf{Set}$ has a centre (§2.2.1) and we later show how to define the centre of a strong monad on an arbitrary symmetric monoidal category (§2.2.2). Unlike the former, the latter submonad does not always exist, but it does exist under mild assumptions and we show that the notion is ubiquitous.

2.2.1 The Centre of a Monad on $\mathbf{Set}$

The results we present next are a special case of our more general development from §2.2.2, but we choose to devote special attention to monads on $\mathbf{Set}$ for illustrative purposes.

Definition 2.1 (Centre)**.** Given a strong monad $(\mathcal{T}, \eta, \mu, \tau)$ on $\mathbf{Set}$ with right strength τ', we say that the *centre* of $\mathcal{T}$ at X, written $\mathcal{Z}X$, is the set

$$\mathcal{Z}X \stackrel{\text{def}}{=} \{t \in \mathcal{T}X \mid \forall Y \in \mathrm{Ob}(\mathbf{Set}).\forall s \in \mathcal{T}Y.\mu(\mathcal{T}\tau'(\tau(t,s))) = \mu(\mathcal{T}\tau(\tau'(t,s)))\}.$$

We write $\iota_X \colon \mathcal{Z}X \subseteq \mathcal{T}X$ for the indicated subset inclusion.

In other words, the centre of $\mathcal{T}$ at X is the subset of $\mathcal{T}X$ which contains all monadic elements for which (1.7) holds when the set X is fixed and the set Y ranges over all sets.

Notice that $\mathcal{Z}X \supseteq \eta_X(X)$, *i.e.*, the centre of $\mathcal{T}$ at X always contains all monadic elements which are in the image of the monadic unit. This follows easily from the axioms of strong monads. In fact, the assignment $\mathcal{Z}(-)$ extends to a *commutative submonad* of $\mathcal{T}$.

In particular, the assignment $\mathcal{Z}(-)$ extends to a functor $\mathcal{Z}\colon \mathbf{Set} \to \mathbf{Set}$ when we define $\mathcal{Z}f \stackrel{\text{def}}{=} \mathcal{T}f|_{\mathcal{Z}X}\colon \mathcal{Z}X \to \mathcal{Z}Y$, for any function $f : X \to Y$, where $\mathcal{T}f|_{\mathcal{Z}X}$ indicates the restriction of $\mathcal{T}f\colon \mathcal{T}X \to \mathcal{T}Y$ to the subset $\mathcal{Z}X$. Moreover, for any two sets X and Y, the monadic unit $\eta_X\colon X \to \mathcal{T}X$, the monadic multiplication $\mu_X : \mathcal{T}^2X \to \mathcal{T}X$, and the monadic strength $\tau_{X,Y}\colon X \times \mathcal{T}Y \to \mathcal{T}(X \times Y)$ (co)restrict respectively to functions $\eta^{\mathcal{Z}}_X\colon X \to \mathcal{Z}X$, $\mu^{\mathcal{Z}}_X\colon \mathcal{Z}^2X \to \mathcal{Z}X$ and $\tau^{\mathcal{Z}}_{X,Y}\colon X \times \mathcal{Z}Y \to \mathcal{Z}(X \times Y)$. That the above four classes of functions (co)restrict as indicated follows from our more general treatment presented in the next section. It then follows, as a special case of Theorem 2.11, that the data we just described constitutes a commutative submonad of $\mathcal{T}$.

Theorem 2.2. *The assignment $\mathcal{Z}(-)$ can be extended to a* commutative submonad *$(\mathcal{Z}, \eta^{\mathcal{Z}}, \mu^{\mathcal{Z}}, \tau^{\mathcal{Z}})$ of $\mathcal{T}$ with the inclusions $\iota_X\colon \mathcal{Z}X \subseteq \mathcal{T}X$ being the submonad morphism. Furthermore, there is a canonical isomorphism of categories $\mathbf{Set}_{\mathcal{Z}} \cong Z(\mathbf{Set}_{\mathcal{T}})$* [a].

a. Theorem 2.11 states precisely in what sense this isomorphism is canonical.

The final statement of Theorem 2.2 shows that the Kleisli category of $\mathcal{Z}$ is canonically isomorphic to the premonoidal centre of the Kleisli category of $\mathcal{T}$. Because of this, we are justified in saying that $\mathcal{Z}$ is not just *a* commutative submonad of $\mathcal{T}$, but rather it is *the centre* of $\mathcal{T}$, which is necessarily commutative (just like the centre of a monoid is a commutative submonoid). In §2.3.2 we provide concrete examples of monads on $\mathbf{Set}$ and their centres and we see that the construction of the centre aligns nicely with our intuition.

2.2.2 The General Construction of the Centre

Throughout the remainder of the section, we assume we are given a symmetric monoidal category $(\mathbf{C}, \otimes, I, \alpha, \lambda, \rho, \gamma)$ and a strong monad $(\mathcal{T}, \eta, \mu, \tau)$ on it with right strength τ'.

In $\mathbf{Set}$, the centre is defined pointwise through subsets of $\mathcal{T}X$ which only contain elements that satisfy the coherence condition for a commutative monad. However, $\mathbf{C}$ is an arbitrary symmetric monoidal category, so we cannot easily form subobjects in the required way. This leads us to the definition of a *central cone* which allows us to overcome this problem.

Definition 2.3 (Central Cone)**.** Let X be an object of $\mathbf{C}$. A *central cone* of $\mathcal{T}$ at X is given by a pair (Z, ι) of an object Z and a morphism $\iota\colon Z \to \mathcal{T}X$, such that for any object Y, the diagram

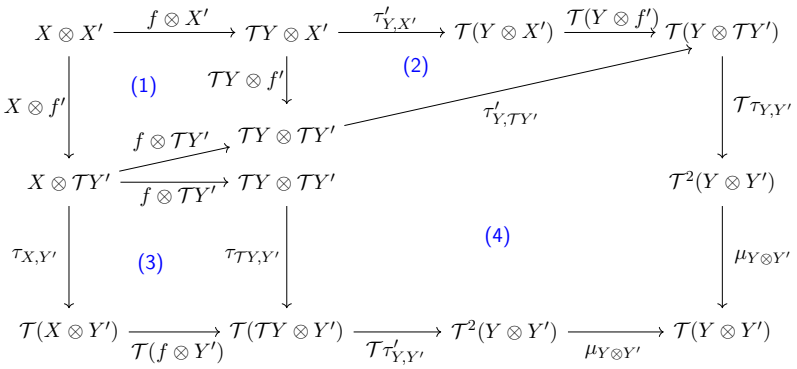
$$\begin{array}{ccccc}
Z \otimes \mathcal{T}Y & \xrightarrow{\iota \otimes \mathcal{T}Y} & \mathcal{T}X \otimes \mathcal{T}Y & \xrightarrow{\tau'_{X,\mathcal{T}Y}} & \mathcal{T}(X \otimes \mathcal{T}Y) \\
{\scriptstyle \iota \otimes \mathcal{T}Y}\downarrow & & & & \downarrow{\scriptstyle \mathcal{T}\tau_{X,Y}} \\
\mathcal{T}X \otimes \mathcal{T}Y & & & & \mathcal{T}^2(X \otimes Y) \\
{\scriptstyle \tau_{\mathcal{T}X,Y}}\downarrow & & & & \downarrow{\scriptstyle \mu_{X\otimes Y}} \\
\mathcal{T}(\mathcal{T}X \otimes Y) & \xrightarrow[\mathcal{T}\tau'_{X,Y}]{} & \mathcal{T}^2(X \otimes Y) & \xrightarrow[\mu_{X\otimes Y}]{} & \mathcal{T}(X \otimes Y)
\end{array}$$

commutes. If (Z, ι) and (Z', ι') are two central cones of $\mathcal{T}$ at X, then a *morphism of central cones* $\varphi\colon (Z', \iota') \to (Z, \iota)$ is a morphism $\varphi\colon Z' \to Z$, such that $\iota \circ \varphi = \iota'$. Thus central cones of $\mathcal{T}$ at X form a category. A *terminal central cone* of $\mathcal{T}$ at X is a central cone (Z, ι) for $\mathcal{T}$ at X, such that for any central cone (Z', ι') of $\mathcal{T}$ at X, there exists a unique morphism of central cones $\varphi\colon (Z', \iota') \to (Z, \iota)$. In other words, it is the terminal object in the category of central cones of $\mathcal{T}$ at X.

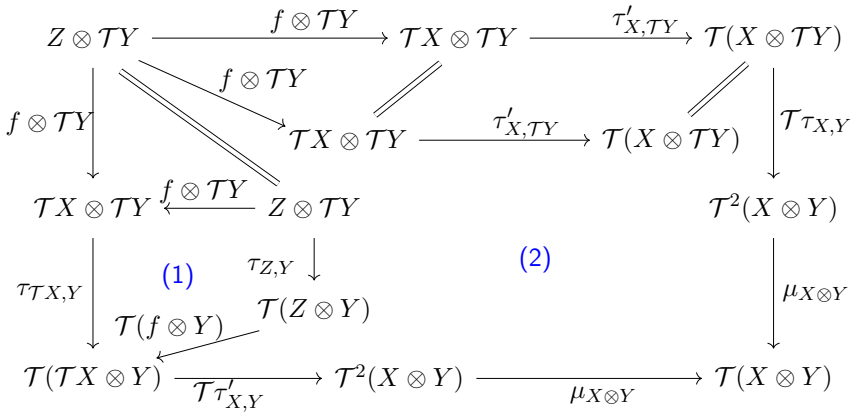
In particular, Definition 2.1 gives a terminal central cone for the special case of monads on $\mathbf{Set}$. The names "central morphism" (in the premonoidal sense, see §1.5.3) and "central cone" (above) also hint that there should be a relationship between them. In fact, the two notions are equivalent.

Proposition 2.4. *Let $f : X \to \mathcal{T}Y$ be a morphism in $\mathbf{C}$. The pair (X, f) is a central cone of $\mathcal{T}$ at Y iff f is central in $\mathbf{C}_\mathcal{T}$ in the premonoidal sense (Definition 1.102).*

Proof. Let (X, f) be a central cone and let $f'\colon X' \to \mathcal{T}Y'$ be a morphism. The following diagram:



commutes because: (1) $\mathbf{C}$ is monoidal; (2) τ' is natural; (3) τ is natural; and (4) the pair (X, f) is a central cone. Therefore, the morphism f is central in the premonoidal sense.
For the other direction, if f is central in $\mathbf{C}_\mathcal{T}$, the following diagram:



commutes because: (1) τ is natural; (2) f is a central morphism; all remaining subdiagrams commute trivially. This shows the pair (X, f) is a central cone. □

From now on, we rely heavily on the fact that central cones and central morphisms are equivalent notions, and we use Proposition 2.4 implicitly in the sequel. On the other hand, *terminal* central cones are crucial for our development, but it is unclear how to introduce a similar notion of "terminal central morphism" that is useful. For this reason, we prefer to work with (terminal) central cones.

It is easy to see that if a terminal central cone for $\mathcal{T}$ at X exists, then it is unique up to a unique isomorphism of central cones. Also, one can easily prove that if (Z, ι) is a terminal central cone, then ι is a monomorphism. The main definition of this subsection follows next and gives the foundation for constructing the centre of a strong monad.

Definition 2.5 (Centralisable Monad)**.** We say that the monad $\mathcal{T}$ is *centralisable* if, for any object X, a terminal central cone of $\mathcal{T}$ at X exists. In this situation, we write $(\mathcal{Z}X, \iota_X)$ for the terminal central cone of $\mathcal{T}$ at X.

In fact, for a centralisable monad $\mathcal{T}$, its terminal central cones induce a commutative submonad $\mathcal{Z}$ of $\mathcal{T}$, as the next theorem shows, and its proof reveals constructively how the monad structure arises from them.

Theorem 2.6. *If the monad $\mathcal{T}$ is centralisable, then the assignment $\mathcal{Z}(-)$ extends to a commutative monad $(\mathcal{Z}, \eta^{\mathcal{Z}}, \mu^{\mathcal{Z}}, \tau^{\mathcal{Z}})$ on $\mathbf{C}$. Moreover, $\mathcal{Z}$ is a commutative submonad of $\mathcal{T}$ and the morphisms $\iota_X \colon \mathcal{Z}X \to \mathcal{T}X$ constitute a monomorphism of strong monads $\iota \colon \mathcal{Z} \Rightarrow \mathcal{T}$.*

This theorem relies on several lemmas that are detailed below.

Lemma 2.7. *If $(X, f \colon X \to \mathcal{T}Y)$ is a central cone of $\mathcal{T}$ at Y, then for any $g \colon Z \to X$, it follows that $(Z, f \circ g)$ is a central cone of $\mathcal{T}$ at Y.*

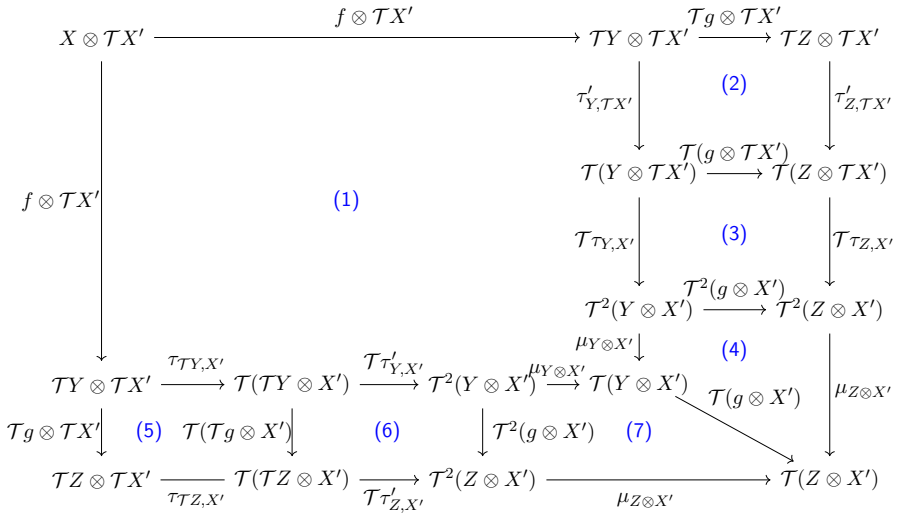
Proof. This is obtained by precomposing the definition of central cone by $g \otimes \mathrm{id}$.

$$\begin{array}{ccccc}
Z \otimes \mathcal{T}X' & \xrightarrow{g \otimes \mathcal{T}X'} & X \otimes \mathcal{T}X' & \xrightarrow{f \otimes \mathcal{T}X'} & \mathcal{T}Y \otimes \mathcal{T}X' \xrightarrow{\tau'_{Y,\mathcal{T}X'}} \mathcal{T}(Y \otimes \mathcal{T}X') \\
 & & f \otimes \mathcal{T}X' \downarrow & & \downarrow \mathcal{T}\tau_{Y,X'} \\
 & & \mathcal{T}Y \otimes \mathcal{T}X' & & \mathcal{T}^2(Y \otimes X') \\
 & & \tau_{\mathcal{T}Y,X'} \downarrow & & \downarrow \mu_{Y \otimes X'} \\
 & & \mathcal{T}(\mathcal{T}Y \otimes X') & \xrightarrow[\mathcal{T}\tau'_{Y,X'}]{} & \mathcal{T}^2(Y \otimes X') \xrightarrow[\mu_{Y \otimes X'}]{} \mathcal{T}(Y \otimes X')
\end{array}$$

commutes directly from the definition of central cone for f. □

Lemma 2.8. *If $(X, f\colon X \to \mathcal{T}Y)$ is a central cone of $\mathcal{T}$ at Y then for any $g\colon Y \to Z$, it follows that $(X, \mathcal{T}g \circ f)$ is a central cone of $\mathcal{T}$ at Z.*

Proof. The naturality of τ and μ allow us to push the application of g to the last postcomposition, in order to use the central property of f. In more details, the following diagram:



commutes, because: (1) f is a central cone, (2) τ' is natural, (3) τ is natural, (4) μ is natural (5) τ is natural, (6) τ' is natural, (7) μ is natural. □

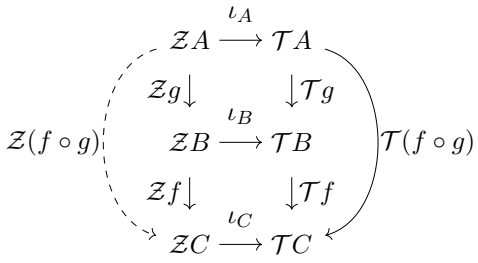
Lemma 2.9. *If (Z, ι) is a terminal central cone of $\mathcal{T}$ at X, then ι is a monomorphism.*

Proof. Let us consider $f, g : Y \to Z$ such that $\iota \circ f = \iota \circ g$; this morphism is a central cone at X (Lemma 2.7), and since (Z, ι) is a terminal central cone, it factors uniquely through ι. Thus $f = g$ and therefore ι is monic. □

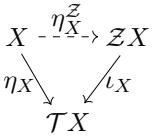
Proof of Theorem 2.6. First let us describe the functorial structure of $\mathcal{Z}$. Recall that $\mathcal{Z}$ maps every object X to its terminal central cone at X. Let $f : X \to Y$ be a morphism. We know that $\mathcal{T}f \circ \iota_X : \mathcal{Z}X \to \mathcal{T}Y$ is a central cone according to Lemma 2.8. Therefore, we define $\mathcal{Z}f$ as the unique map such that the following diagram commutes:

$$\begin{array}{ccc} \mathcal{Z}X & \overset{\mathcal{Z}f}{\dashrightarrow} & \mathcal{Z}Y \\ {\scriptstyle \iota_X}\downarrow & & \downarrow{\scriptstyle \iota_Y} \\ \mathcal{T}X & \underset{\mathcal{T}f}{\longrightarrow} & \mathcal{T}Y \end{array}$$

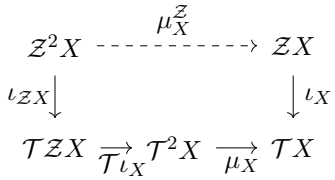
It follows directly that $\mathcal{Z}$ maps the identity to the identity, and that ι is natural. $\mathcal{Z}$ also preserves composition, which follows by the commutative diagram below.



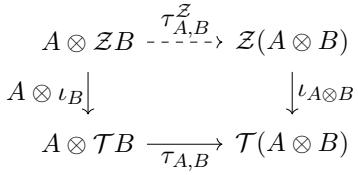
This proves that $\mathcal{Z}$ is a functor. Next, we describe its monad structure and after that we show that it is commutative.
The monadic unit η_X is central, because it is the identity morphism in $Z(\mathbf{C}_\mathcal{T})$, thus it factors through ι_X to define $\eta^\mathcal{Z}_X$.



Next, observe that, by definition, $\mu_X \circ \mathcal{T}\iota_X \circ \iota_{\mathcal{Z}X} = \iota_X \odot \iota_{\mathcal{Z}X}$, where $(- \odot -)$ indicates Kleisli composition. Since ι is central and Kleisli composition preserves central morphisms (see Definition 1.102, central morphisms form a subcategory of the Kleisli category), it follows that this morphism factors through ι_X and we use this to define $\mu^\mathcal{Z}_X$ as in the diagram below.

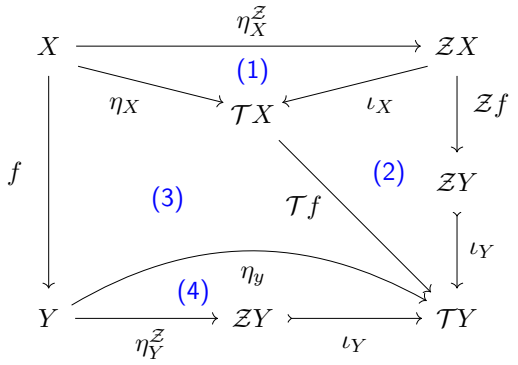


Again, by definition, $\tau_{A,B} \circ (A \otimes \iota_B) = A \otimes_r \iota_B$. Central morphisms are preserved by the premonoidal products (see 1.5.3) and therefore, this morphism factors through $\iota_{A\otimes B}$ which we use to define $\tau^\mathcal{Z}_{A,B}$ as in the diagram below.



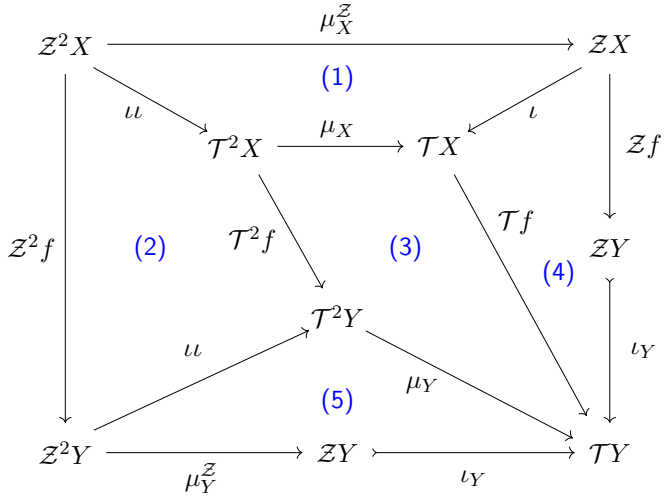
Note that the last three diagrams are exactly those of a morphism of strong monads (see Definition 1.99). Using the fact that ι is monic (see Lemma 2.9), the following commutative

diagram shows that $\eta^{\mathcal{Z}}$ is natural.

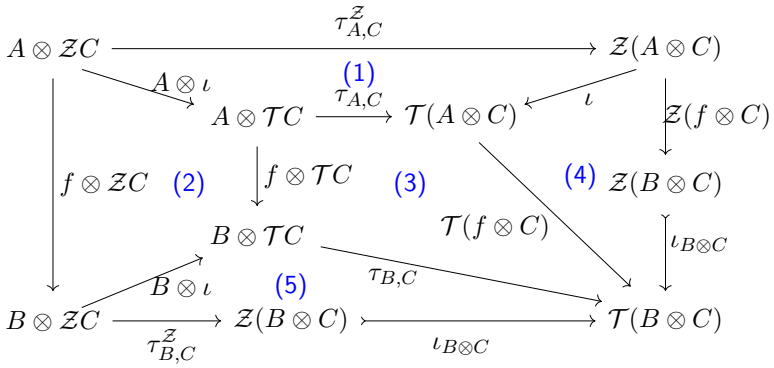


(1) definition of $\eta^{\mathcal{Z}}$, (2) ι is natural, (3) η is natural and (4) definition of $\eta^{\mathcal{Z}}$. Thus, we have proven that for any $f\colon X \to Y$, $\iota_Y \circ \mathcal{Z}f \circ \eta^{\mathcal{Z}}_X = \iota_Y \circ \eta^{\mathcal{Z}}_Y \circ f$. Besides, ι is monic, thus $\mathcal{Z}f \circ \eta^{\mathcal{Z}}_X = \eta^{\mathcal{Z}}_Y \circ f$ which proves that $\eta^{\mathcal{Z}}$ is natural. We will prove all the remaining diagrams with the same reasoning.

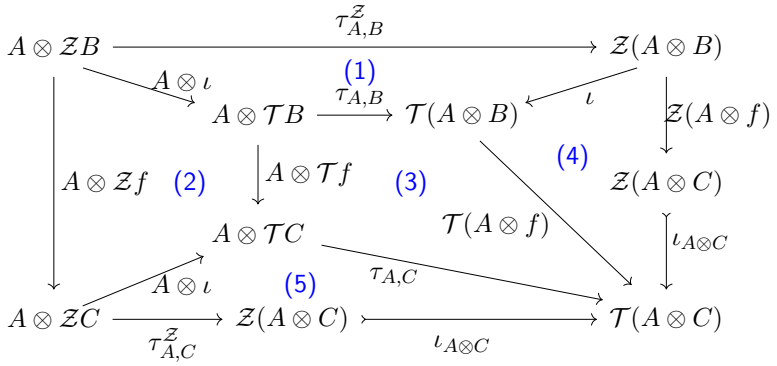
The following commutative diagram shows that $\mu^{\mathcal{Z}}$ is natural.



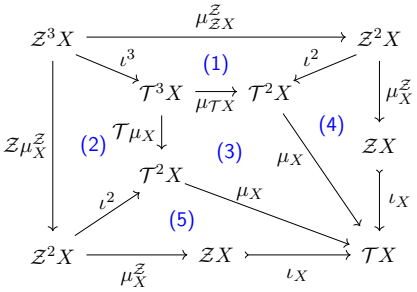
(1) definition of $\mu^{\mathcal{Z}}$, (2) ι is natural, (3) μ is natural, (4) ι is natural and (5) definition of $\mu^{\mathcal{Z}}$. The following commutative diagrams shows that $\tau^{\mathcal{Z}}$ is natural.

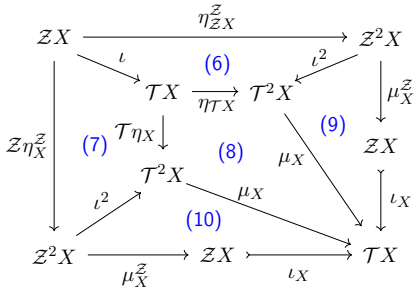


(1) definition of $\tau^{\mathcal{Z}}$, (2) ι is natural, (3) τ is natural, (4) ι is natural and (5) definition of $\tau^{\mathcal{Z}}$.



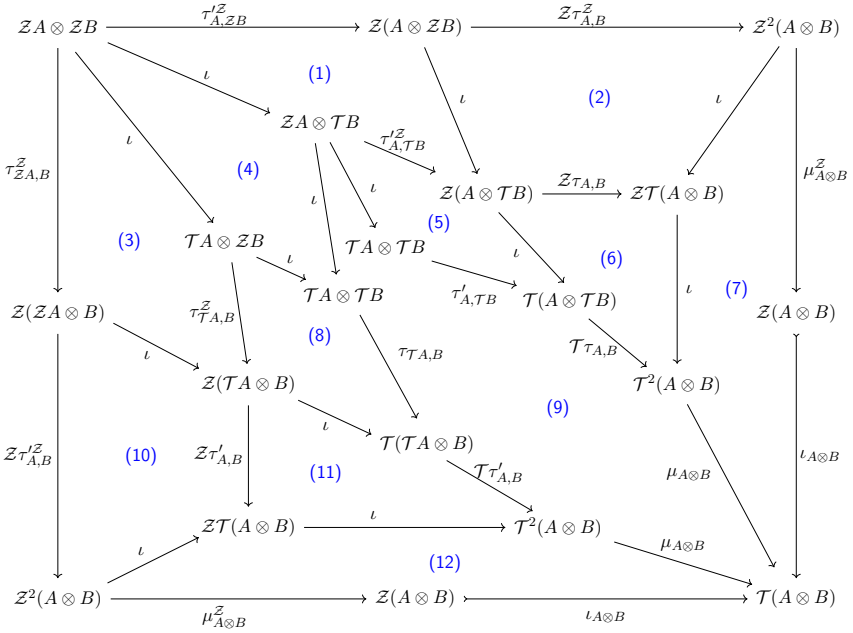
(1) definition of $\tau^{\mathcal{Z}}$, (2) ι is natural, (3) τ is natural, (4) ι is natural and (5) definition of $\tau^{\mathcal{Z}}$. The following commutative diagrams prove that $\mathcal{Z}$ is a monad.





(1) and (2) involve the definition of $\mu^{\mathcal{Z}}$ and the naturality of ι and $\mu^{\mathcal{Z}}$, (3) is by definition of monad, (4) definition of $\mu^{\mathcal{Z}}$ and (5) also. (6) and (7) involve the definition of $\eta^{\mathcal{Z}}$ and the naturality of ι and $\eta^{\mathcal{Z}}$, (8) is by definition of monad, (9) definition of $\mu^{\mathcal{Z}}$ and (10) also.

$\mathcal{Z}$ is proven strong with very similar diagrams. The commutative diagram:



(2.1)

proves that $\mathcal{Z}$ is a commutative monad, with (1) $\tau'^{\mathcal{Z}}$ is natural, (2) definition of $\tau^{\mathcal{Z}}$, (3) $\tau^{\mathcal{Z}}$ is natural, (4) $\mathbf{C}$ is monoidal, (5) definition of $\tau'^{\mathcal{Z}}$, (6) ι is natural, (7) definition of $\mu^{\mathcal{Z}}$, (8) definition of $\tau^{\mathcal{Z}}$, (9) ι is central, (10) definition of $\tau'^{\mathcal{Z}}$, (11) ι is natural and (12) definition of $\mu^{\mathcal{Z}}$. □

Theorem 2.6 shows that centralisable monads always induce a canonical commutative submonad. Next, we justify why this submonad should be seen as the centre of $\mathcal{T}$. Note that since $\mathcal{Z}$ is a submonad of $\mathcal{T}$, we know that $\mathbf{C}_{\mathcal{Z}}$ canonically embeds into $\mathbf{C}_{\mathcal{T}}$ (see Proposition 1.101). The next theorem shows that this embedding factors through the premonoidal centre of $\mathbf{C}_{\mathcal{T}}$, and moreover, the two categories are isomorphic.

Theorem 2.10. *In the situation of Theorem 2.6, the canonical embedding functor* $\mathcal{I}\colon \mathbf{C}_{\mathcal{Z}} \to \mathbf{C}_{\mathcal{T}}$ *corestricts to an isomorphism of categories* $\mathbf{C}_{\mathcal{Z}} \cong Z(\mathbf{C}_{\mathcal{T}})$.

Proof. $\mathcal{I}$ corestricts as indicated follows easily: for any morphism $f\colon X \to \mathcal{Z}Y$, we have that $\mathcal{I}f = \iota_Y \circ f$ which is central by Lemma 2.7. Let us write $\hat{\mathcal{I}}$ for the corestriction of $\mathcal{I}$ to $Z(\mathbf{C}_{\mathcal{T}})$. Next, to prove that $\hat{\mathcal{I}}\colon \mathbf{C}_{\mathcal{Z}} \to Z(\mathbf{C}_{\mathcal{T}})$ is an isomorphism, we define the inverse functor $G : Z(\mathbf{C}_{\mathcal{T}}) \to \mathbf{C}_{\mathcal{Z}}$.

On objects, we have $G(X) \stackrel{\text{def}}{=} X$. To define its mapping on morphisms, observe that if $f\colon X \to \mathcal{T}Y$ is a central morphism (in the premonoidal sense), then (X, f) is a central cone of $\mathcal{T}$ at Y (Proposition 2.4) and therefore there exists a unique morphism $f^{\mathcal{Z}}\colon X \to \mathcal{Z}Y$ such that $\iota_Y \circ f^{\mathcal{Z}} = f$; we define $Gf \stackrel{\text{def}}{=} f^{\mathcal{Z}}$. The proof that G is a functor is direct considering that any $f^{\mathcal{Z}}$ is a morphism of central cones and that all components of ι are monomorphisms.

To show that $\hat{\mathcal{I}}$ and G are mutual inverses, let $f : X \to \mathcal{T}Y$ be a morphism of $Z(\mathbf{C}_{\mathcal{T}})$, *i.e.*, a central morphism. Then, $\hat{\mathcal{I}}Gf = \iota_Y \circ f^{\mathcal{Z}} = f$ by definition of morphism of central

cones (see Definition 2.3). For the other direction, let $g : X \to \mathcal{Z}Y$ be a morphism in $\mathbf{C}$. Then, $\iota_Y \circ G\hat{\mathcal{I}}g = \iota_Y \circ (\iota_Y \circ g)^{\mathcal{Z}} = \iota_Y \circ g$ by Definition 2.3 and thus $G\hat{\mathcal{I}}g = g$ since ι_Y is a monomorphism (Lemma 2.9). □

It should now be clear that Theorem 2.6 and Theorem 2.10 show that we are justified in naming the submonad $\mathcal{Z}$ as *the* centre of $\mathcal{T}$. The existence of terminal central cones is not only sufficient to construct the centre (as we just showed), but it also is necessary and we show this next. Furthermore, we provide another equivalent characterisation in terms of the premonoidal structure of the monad.

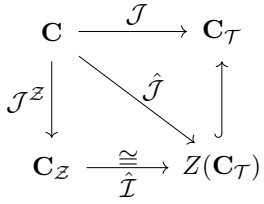
Theorem 2.11 (Centre). *Let $\mathbf{C}$ be a symmetric monoidal category and $\mathcal{T}$ a strong monad on it. The following are equivalent:*

1. *For any object X of $\mathbf{C}$, $\mathcal{T}$ admits a terminal central cone at X;*
2. *There exists a commutative submonad $\mathcal{Z}$ of $\mathcal{T}$ (which we call* the centre *of $\mathcal{T}$) such that the canonical embedding functor $\mathcal{I} : \mathbf{C}_{\mathcal{Z}} \to \mathbf{C}_{\mathcal{T}}$ corestricts to an isomorphism of categories $\mathbf{C}_{\mathcal{Z}} \cong Z(\mathbf{C}_{\mathcal{T}})$;*
3. *The corestriction of the Kleisli left adjoint $\mathcal{J} : \mathbf{C} \to \mathbf{C}_{\mathcal{T}}$ to the premonoidal centre $\hat{\mathcal{J}} : \mathbf{C} \to Z(\mathbf{C}_{\mathcal{T}})$ also is a left adjoint.*

Proof. We follow a circular strategy in order to prove that each of the points implies the others.

$(1 \Rightarrow 2)$: By Theorem 2.6 and Theorem 2.10.

$(2 \Rightarrow 3)$: Let us consider the Kleisli left adjoint $\mathcal{J}^{\mathcal{Z}}$ associated to the monad $\mathcal{Z}$. All our hypotheses can be summarised by the diagram



where $\hat{\mathcal{I}}\colon \mathbf{C}_{\mathcal{Z}} \cong Z(\mathbf{C}_{\mathcal{T}})$ is the corestriction of $\mathcal{I}$. This diagram commutes, because $\mathcal{Z}$ is a submonad of $\mathcal{T}$ (recall also that $\hat{\mathcal{J}}$ is the indicated corestriction of $\mathcal{J}$, see §1.5.3). Since $\hat{\mathcal{I}}$ is an isomorphism, then $\hat{\mathcal{J}} = \hat{\mathcal{I}} \circ \mathcal{J}^{\mathcal{Z}}$ is the composition of two left adjoints and it is therefore also a left adjoint.

$(3 \Rightarrow 1)$: Let $\mathcal{R}\colon Z(\mathbf{C}_{\mathcal{T}}) \to \mathbf{C}$ be the right adjoint of $\hat{\mathcal{J}}$ and let ε be the counit of the adjunction. We will show that the pair $(\mathcal{R}X, \varepsilon_X)$ is the terminal central cone of $\mathcal{T}$ at X.
First, since ε_X is a morphism in $Z(\mathbf{C}_{\mathcal{T}})$, it follows that it is central. Thus the pair $(\mathcal{R}X, \varepsilon_X)$ is a central cone of $\mathcal{T}$ at X. Next, let $\Phi\colon Z(\mathbf{C}_{\mathcal{T}})[\hat{\mathcal{J}}Y, X] \cong \mathbf{C}[Y, \mathcal{R}X]$ be the natural bijection induced by the adjunction. If $f\colon Y \to \mathcal{T}X$ is central, meaning a morphism of $Z(\mathbf{C}_{\mathcal{T}})$, the diagram below left commutes in $Z(\mathbf{C}_{\mathcal{T}})$, or equivalently, the diagram below right commutes in

$\mathbf{C}$:

Note that the pair (Y, f) is equivalently a central cone for $\mathcal{T}$ at X (by Proposition 2.4). Thus f uniquely factors through the counit $\varepsilon_X : \mathcal{R}X \to \mathcal{T}X$ and therefore $(\mathcal{R}X, \varepsilon_X)$ is the terminal central cone of $\mathcal{T}$ at X. □

This theorem shows that Definition 2.5 may be stated by choosing any one of the above equivalent criteria. We note that the first condition is the easiest to verify in practice. The second one is the most useful for providing a computational interpretation, as we do in the sequel. The third condition provides an important link to premonoidal categories.

Example 2.12. Given a monoid (M, e, m), consider the free monad induced by M, also known as the *writer monad*, which we write as $\mathcal{T} = (- \times M)\colon \mathbf{Set} \to \mathbf{Set}$. The centre $\mathcal{Z}$ of $\mathcal{T}$ is given by the commutative monad $(- \times Z(M))\colon \mathbf{Set} \to \mathbf{Set}$, where $Z(M)$ is the centre of the monoid M and where the monad data is given by the (co)restrictions of the monad data of $\mathcal{T}$. Note that $\mathcal{T}$ is a commutative monad iff M is a commutative monoid. See also Example 2.13.

2.2.3 A Non-centralisable Monad

In $\mathbf{Set}$, the terminal central cones used to define the centre are defined by taking appropriate subsets. One may wonder what happens if not every subset of a given set is an object of the category. The following example describes such a situation, which gives rise to a non-centralisable strong monad.

Example 2.13. Consider the Dihedral group $\mathbb{D}_4$, which has 8 elements. Its centre $Z(\mathbb{D}_4)$ is non-trivial and has 2 elements. Let $\mathbf{C}$ be the full subcategory of $\mathbf{Set}$ with objects that are finite products of the set $\mathbb{D}_4$ with itself. This category has a cartesian structure, and the terminal object is the singleton set (which is the empty product). Notice that every object in this category has a cardinality that is a power of 8. Therefore the cardinality of every homset of $\mathbf{C}$ is a power of 8. Since $\mathbf{C}$ has a cartesian structure and since $\mathbb{D}_4$ is a monoid, we can consider the writer monad $\mathcal{M} \stackrel{\text{def}}{=} (- \times \mathbb{D}_4)\colon \mathbf{C} \to \mathbf{C}$ induced by $\mathbb{D}_4$, which can be defined in the same way as in Example 2.12. It follows that $\mathcal{M}$ is a strong monad on $\mathbf{C}$. However, it is easy to show that this monad is not centralisable. Assume (for contradiction) that there is a monad $\mathcal{Z}\colon \mathbf{C} \to \mathbf{C}$ such that $\mathbf{C}_{\mathcal{Z}} \cong Z(\mathbf{C}_{\mathcal{M}})$ (see Theorem 2.11). Next, observe that the homset $Z(\mathbf{C}_{\mathcal{M}})[1, 1]$ has the same cardinality as the centre of the monoid $\mathbb{D}_4$, *i.e.*, its cardinality is 2. However, $\mathbf{C}_{\mathcal{Z}}$ cannot have such a homset since $\mathbf{C}_{\mathcal{Z}}[X, Y] = \mathbf{C}[X, \mathcal{Z}Y]$ which must have cardinality a power of 8. Therefore there exists no such monad $\mathcal{Z}$ and $\mathcal{M}$ is not centralisable.

Besides this example and any further attempts at constructing non-centralisable monads for this sole purpose, we do not know of any other strong monad in the literature that is not centralisable. In the next section, we present many examples of centralisable monads and classes of centralisable monads which show that our results are widely applicable.

2.3 Examples of Centres of Strong Monads

In this section, we show how we can make use of the mathematical results we already established in order to reason about the centres of monads of interest.

2.3.1 Categories whose Strong Monads are Centralisable

We saw earlier that every (strong) monad on $\mathbf{Set}$ is centralisable. In fact, this is also true for many other naturally occurring categories. For example, in many categories of interest, the objects of the category have a suitable notion of subobject (e.g., subsets in $\mathbf{Set}$, subspaces in $\mathbf{Vect}$) and the centre can be constructed in a similar way to the one in $\mathbf{Set}$.

Example 2.14. Let $\mathbf{DCPO}$ be the category whose objects are directed-complete partial orders and whose morphisms are Scott-continuous maps between them. Every strong monad on $\mathbf{DCPO}$ with respect to its cartesian structure is centralisable. The easiest way to see this is to use Theorem 2.11 (1). Writing $\mathcal{T}\colon \mathbf{DCPO} \to \mathbf{DCPO}$ for an arbitrary strong monad on $\mathbf{DCPO}$, the terminal central cone of $\mathcal{T}$ at X is given by the subdcpo $\mathcal{Z}X \subseteq \mathcal{T}X$ which has the underlying set $\mathcal{Z}X \stackrel{\text{def}}{=} \{t \in \mathcal{T}X \mid \forall Y \in \mathrm{Ob}(\mathbf{DCPO}).\forall s \in \mathcal{T}Y.\ \ \mu(\mathcal{T}\tau'(\tau(t,s))) = \mu(\mathcal{T}\tau(\tau'(t,s)))\}$. That $\mathcal{Z}X$ (with the inherited order) is a subdcpo of $\mathcal{T}X$ follows easily by using the fact that μ, τ, τ' and $\mathcal{T}$ are Scott-continuous. Therefore, the construction is fully analogous to the one in $\mathbf{Set}$.

Example 2.15. Let $\mathbf{Top}$ be the category whose objects are topological spaces, and whose morphisms are continuous maps between them. Every strong monad on $\mathbf{Top}$ with respect to its cartesian structure is centralisable. Using Theorem 2.11 (1) and writing $\mathcal{T} : \mathbf{Top} \to \mathbf{Top}$ for an arbitrary strong monad on $\mathbf{Top}$, the terminal central cone of $\mathcal{T}$ at X is given by the space $\mathcal{Z}X \subseteq \mathcal{T}X$ which has the underlying set $\mathcal{Z}X \stackrel{\text{def}}{=} \{t \in \mathcal{T}X \mid \forall Y \in \mathrm{Ob}(\mathbf{Top}).\forall s \in \mathcal{T}Y.\ \mu(\mathcal{T}\tau'(\tau(t,s))) = \mu(\mathcal{T}\tau(\tau'(t,s)))\}$ and whose topology is the subspace topology inherited from $\mathcal{T}X$.

Example 2.16. Every strong monad on the category $\mathbf{Meas}$ (whose objects are measurable spaces and the morphisms are measurable maps between them) is centralisable. The construction is fully analogous to the previous example, but instead of the subspace topology, we equip the underlying set with the subspace σ-algebra inherited from $\mathcal{T}X$ (which is the smallest σ-algebra that makes the subset inclusion map measurable).

Example 2.17. Let $\mathbf{Vect}$ be the category whose objects are vector spaces, and whose morphisms are linear maps between them. Every strong monad on $\mathbf{Vect}$ with respect to the usual symmetric monoidal structure is centralisable. One simply defines the subset $\mathcal{Z}X$ as in the other examples and shows that this is a linear subspace of $\mathcal{T}X$. That this is the terminal central cone is then obvious.

The above categories, together with the category $\mathbf{Set}$, are not meant to provide an exhaustive list of categories for which all strong monads are centralisable. Indeed, there are many more categories for which this is true. The purpose of these examples is to illustrate how we may use Theorem 2.11 (1) to construct the centre of a strong monad. Changing perspective, the proof of the next proposition uses Theorem 2.11 (3).

Proposition 2.18. *Let $\mathbf{C}$ be a symmetric monoidal closed category that is total* – i.e., *a locally small category whose Yoneda embedding has a left adjoint. Then all strong monads over $\mathbf{C}$ are centralisable.*

Proof. This proof was provided by Titouan Carette in [CLZ23]. □

Example 2.19. Any category which is the Eilenberg-Moore category of a commutative monad over $\mathbf{Set}$ is total [Kel86]. Furthermore it is symmetric monoidal closed [Kei78], thus all strong monads on it are centralisable. This includes: the category $\mathbf{Set}_*$ of pointed sets and point preserving functions (algebras of the lift monad); the category $\mathbf{CMon}$ of commutative monoids and monoid homomorphisms (algebras of the commutative monoid monad); the category $\mathbf{Conv}$ of convex sets and linear functions (algebras of the distribution monad); and the category $\mathbf{Sup}$ of complete semilattices and sup-preserving functions (algebras of the powerset monad).

Example 2.20. Any presheaf category $\mathbf{Set}^{\mathbf{C}^{\mathrm{op}}}$ over a small category $\mathbf{C}$ is total [Kel86] and cartesian closed, thus all strong monads on it (with respect to the cartesian structure) are centralisable. This includes: the category $\mathbf{Set}^{A^{\mathrm{op}}}$, where A is the category with two objects and two parallel arrows, which can be seen as the category of directed multi-graphs and graph homomorphisms; the category $\mathbf{Set}^{G^{\mathrm{op}}}$, where G is a group seen as a category, which can be seen as the category of G-sets (sets with an action of G) and equivariant maps; and the topos of trees $\mathbf{Set}^{\mathbb{N}^{\mathrm{op}}}$. If $\mathbf{C}$ is symmetric monoidal, then the Day convolution product makes $\mathbf{Set}^{\mathbf{C}^{\mathrm{op}}}$ symmetric monoidal closed [Day70], hence all strong monads on it with respect to the Day convolution monoidal structure also are centralisable.

Example 2.21. Any Grothendieck topos is cartesian closed and total, therefore it satisfies the conditions of Proposition 2.18.

2.3.2 Specific Examples of Centralisable Monads

In this subsection, we consider specific monads and construct their centres.

Example 2.22. Every commutative monad is naturally isomorphic to its centre.

Example 2.23. Let S be a set and consider the well-known continuation monad $\mathcal{T} = [[-, S], S] : \mathbf{Set} \to \mathbf{Set}$. Note that, if S is the empty set or a singleton set, then $\mathcal{T}$ is commutative, so we are in the situation of Example 2.22. Otherwise, when S is not trivial, one can prove (details omitted here) that $\mathcal{Z}X = \eta_X(X) \cong X$. Therefore, the centre of $\mathcal{T}$ is trivial and it is naturally isomorphic to the identity monad.

Example 2.24. Consider the well-known list monad $T : \mathbf{Set} \to \mathbf{Set}$ that is given by $TX = \bigsqcup_{n \geq 0} X^n$. Then, the centre of $\mathcal{T}$ is naturally isomorphic to the identity monad.

Example 2.23 shows that the centre of a monad may be trivial in the sense that it is precisely the image of the monadic unit and this is the least it can be. At the other extreme, Example 2.22 shows that the centre of a commutative monad coincides with itself, as one would expect. Thus, the monads that have interesting centres are those monads which are strong but not commutative, and which have non-trivial centres, such as the one in Example 2.12. Another interesting example of a strong monad with a non-trivial centre is provided next.

Example 2.25. Every semiring $(S, +, 0, \cdot, 1)$ induces a monad $\mathcal{T}_S : \mathbf{Set} \to \mathbf{Set}$ [JMS22]. This monad maps a set X to the set of finite formal sums of the form $\sum s_i x_i$, where s_i are elements of S and x_i are elements of X. The monad $\mathcal{T}_S$ is commutative iff S is commutative as a semiring. The centre $\mathcal{Z}$ of $\mathcal{T}_S$ is induced by the commutative semiring $Z(S)$, *i.e.*, by the centre of S in the usual sense. Therefore, $\mathcal{Z} = \mathcal{T}_{Z(S)}$.

Example 2.26. Any Lawvere theory $\mathbf{T}$ [HP07] induces a finitary monad on $\mathbf{Set}$. The centre of this monad is the monad induced by the centre of $\mathbf{T}$ in the sense of Lawvere theories [Wra70]. This is detailed in §2.3.3.

Example 2.27. The valuations monad $\mathcal{V} \colon \mathbf{DCPO} \to \mathbf{DCPO}$ [JP89, Jon90] is similar in spirit to the Giry monad on measurable spaces [Gir82]. It is an important monad in domain theory [GHK+12] that is used to combine probability and recursion for dcpo's. Given a dcpo X, the valuations monad $\mathcal{V}$ assigns the dcpo $\mathcal{V}X$ of all Scott-continuous *valuations* on X, which are Scott-continuous functions $\nu \colon \sigma(X) \to [0, 1]$ from the Scott-open sets of X into the unit interval that satisfy some additional properties that make them suitable to model probability (details omitted here, see [Jon90] for more information). The category $\mathbf{DCPO}$ is cartesian closed and the valuations monad $\mathcal{V} \colon \mathbf{DCPO} \to \mathbf{DCPO}$ is strong, but its commutativity on $\mathbf{DCPO}$ has been an open problem since 1989 [Jon90, JP89, JMZ21, JLMZ21b, GJT21]. The difficulty in (dis)proving the commutativity of $\mathcal{V}$ boils down to (dis)proving the following Fubini-style equation

$$\int_X \int_Y \chi_U(x, y) d\nu d\xi = \int_Y \int_X \chi_U(x, y) d\xi d\nu$$

holds for any dcpo's X and Y, any Scott-open subset $U \in \sigma(X \times Y)$ and any two valuations $\xi \in \mathcal{V}X$ and $\nu \in \mathcal{V}Y$. In the above equation, the notion of integration is given by the *valuation integral* (see [Jon90] for more information).

The *central valuations monad* [JMZ21], is the submonad $\mathcal{Z} \colon \mathbf{DCPO} \to \mathbf{DCPO}$ that maps a dcpo X to the dcpo $\mathcal{Z}X$ which has all *central valuations* as elements. Equivalently:

$$\mathcal{Z}X \stackrel{\text{def}}{=} \Big\{ \xi \in \mathcal{V}(X) \mid \forall Y \in \mathrm{Ob}(\mathbf{DCPO}). \forall U \in \sigma(X \times Y).$$
$$\forall \nu \in \mathcal{V}(Y). \int_X \int_Y \chi_U(x, y) d\nu d\xi = \int_Y \int_X \chi_U(x, y) d\xi d\nu \Big\}.$$

But this is precisely the centre of $\mathcal{V}$, which can be seen using Theorem 2.11 (1) after unpacking the definition of the monad data of $\mathcal{V}$. Therefore, we see that the main result of [JMZ21] is a special case of our more general categorical treatment. We wish to note, that the centre of $\mathcal{V}$ is quite large. It contains all three commutative submonads identified in [JLMZ21b] and all of them may be used to model lambda calculi with recursion and discrete probabilistic choice (see [JLMZ21b, JMZ21]).

2.3.3 Link with Lawvere theories

Commutants for Lawvere theories [HP07] were defined in Wraith's lecture notes [Wra70] but were only studied in details by Lucyshyn-Wright [LW18] later. The centre of a Lawvere theory is a special case of commutant.

In a Lawvere theory $\mathbf{T}$, we say that $f : A^n \to A^{n'}$ and $g : A^m \to A^{m'}$ commute if and only if $f^{m'} \circ g^n$ (also written $f \star g$) and $g^{n'} \circ f^m$ (also written $g \star f$) are equal, up to isomorphism. If $\mathbf{S}$ is a full subcategory of $\mathbf{T}$, one can define the commutant of $\mathbf{S}$ in $\mathbf{T}$, meaning a full subcategory of $\mathbf{T}$ whose morphisms commute with the morphisms of $\mathbf{S}$. This commutant is written $\mathbf{S}^{\perp}$, and is also a Lawvere subtheory of $\mathbf{T}$. Considering this, $\mathbf{T}^{\perp}$ is seen as the *centre* of the Lawvere theory $\mathbf{T}$ [Wra70]; and any subtheory of $\mathbf{T}^{\perp}$ is a *central subtheory* of $\mathbf{T}$.

What about monads? Models of a Lawvere theory $\mathbf{T}$ are finite-product-preserving functors $\mathbf{T} \to \mathbf{Set}$ and they form a category $\mathbf{Mod}(\mathbf{T}, \mathbf{Set})$. This category is adjoint to $\mathbf{Set}$ through a forgetful and free functors. Those adjunctions give birth to a monad. This monad is on $\mathbf{Set}$, thus it is strong, centralisable and finitary since it originates from a Lawvere theory. Thus given a Lawvere theory $\mathbf{T}$, we obtain a monad $\mathcal{T}$ whose centre $\mathcal{Z}$ is a commutative submonad of $\mathcal{T}$ and is finitary, which means that there exists a corresponding Lawvere theory. This Lawvere theory is a commutative subtheory of $\mathbf{T}$, as proven next.

The connection between Lawvere theories and finitary monads is extensively detailed in [Str72, Gar14, GP18]. To get a Lawvere theory out of a finitary monad $\mathcal{Z}$ on $\mathbf{Set}$, one needs to look at the opposite category of a skeleton of $\mathbf{Set}_{\mathcal{Z}}$ [HP07], noted here $\mathfrak{s}\mathbf{Set}^{op}_{\mathcal{Z}}$. This Lawvere theory is commutative because $\mathbf{Set}_{\mathcal{Z}}$ is monoidal. Moreover, $\mathbf{Set}_{\mathcal{Z}}$ is embedded in $\mathbf{Set}_{\mathcal{T}}$, then $\mathfrak{s}\mathbf{Set}^{op}_{\mathcal{Z}}$ is embedded in $\mathfrak{s}\mathbf{Set}^{op}_{\mathcal{T}}$; the latter being equivalent to $\mathbf{T}$.

Theorem 2.28. *Given a Lawvere theory $\mathbf{T}$, its $\mathbf{Set}$-monad $\mathcal{T}$ is centralisable and its centre $\mathcal{Z}$ has a corresponding Lawvere theory $\mathfrak{s}\mathbf{Set}^{op}_{\mathcal{Z}}$ that is equivalent to $\mathbf{T}^{\perp}$.*

Proof. This is a direct application of the point (2) of Theorem 2.11. □

This connection helps motivate a similar theory for commutants in the general context of strong monads. However, the litterature on Lawvere theories is not enough to grasp all those monads on symmetric monoidal category: in this subsection, we have only given the example for the category $\mathbf{Set}$, and in general, in the literature, the category is often required to be closed.

2.4 Central Submonads

So far, we focused primarily on *the* centre of a strong monad. Now we focus our attention on *central submonads* of a strong monad which we define by taking inspiration from the notion of central subgroup in group theory. Just like central subgroups, central submonads are more general compared to the centre. The centre of a strong monad, whenever it exists, can be intuitively understood as the largest central submonad, so the two notions are strongly related. We will later see that central submonads are more interesting computationally.

Theorem 2.29 (Centrality)**.** *Let $\mathbf{C}$ be a symmetric monoidal category and $\mathcal{T}$ a strong monad on it. Let $\mathcal{S}$ be a strong submonad of $\mathcal{T}$ with $\iota : \mathcal{S} \Rightarrow \mathcal{T}$ the strong submonad monomorphism. The following are equivalent:*

1) *For any object X of $\mathbf{C}$, $(\mathcal{S}X, \iota_X)$ is a central cone for $\mathcal{T}$ at X;*
2) *the canonical embedding functor $\mathcal{I} : \mathbf{C}_{\mathcal{S}} \to \mathbf{C}_{\mathcal{T}}$ corestricts to an embedding of*

categories $\hat{\mathcal{I}} : \mathbf{C}_{\mathcal{S}} \to Z(\mathbf{C}_{\mathcal{T}})$.

Furthermore, these conditions imply that $\mathcal{S}$ *is a commutative submonad of* $\mathcal{T}$*. Under the additional assumption that* $\mathcal{T}$ *is centralisable, these conditions also are equivalent to:*

3) $\mathcal{S}$ *is a submonad of the centre of* $\mathcal{T}$*, and thus is commutative.*

Proof.

$(1 \Rightarrow 2)$: The proof of Th. 2.10 contains the necessary elements for this proof. In details, we know that all the components of ι are central, and we also know that precomposing a central morphism keeps being central (see Lemma 2.7).

$(2 \Rightarrow 1)$: The hypothesis ensures that $\hat{\mathcal{I}}(id_X) = \iota_X$ is central.

The diagram in (2.1) proves that the centre of a centralisable monad is commutative. Assuming (1) – or (2) – is true, then the same diagram replacing $\mathcal{Z}$ by $\mathcal{S}$ proves that $\mathcal{S}$ is a commutative monad.

$(1 \Rightarrow 3)$: Moreover, each $\iota^{\mathcal{S}}_X : \mathcal{S}X \Rightarrow \mathcal{T}X$ factorises through the terminal central cone $\iota^{\mathcal{Z}}_X$. A strong monad morphism $\mathcal{S} \Rightarrow \mathcal{Z}$ arises from those factorisations.

$(3 \Rightarrow 1)$: Let us write $\mathcal{Z}$ the centre of $\mathcal{T}$, $\iota^{\mathcal{S}} : \mathcal{S} \Rightarrow \mathcal{Z}$ and $\iota^{\mathcal{Z}} : \mathcal{Z} \Rightarrow \mathcal{T}$ the submonad morphisms. The components of $\iota^{\mathcal{Z}}$ are terminal central cones, and are in particular central, so $\iota^{\mathcal{Z}} \circ \iota^{\mathcal{S}}$ is also central by Lemma 2.7. Thus the components of the submonad morphism from $\mathcal{S}$ to $\mathcal{T}$ are central. □

Definition 2.30 (Central Submonad)**.** Given a strong submonad $\mathcal{S}$ of $\mathcal{T}$, we say that $\mathcal{S}$ is a *central submonad* of $\mathcal{T}$ if it satisfies any one of the above equivalent criteria from Theorem 2.29.

Just like the centre of a strong monad, any central submonad also is commutative and the above theorem (Theorem 2.30) shows that central submonads have a similar structure to the centre of a strong monad. The final statement shows that we may see the centre (whenever it exists) as the largest central submonad of $\mathcal{T}$. The centre of a strong monad often does exist (as we already argued), so the last criterion also provides a simple way to determine whether a submonad is central or not.

Example 2.31. By the above theorem, every centre described in §2.3 is a central submonad.

Example 2.32. Let $\mathcal{T}$ be a strong monad on a symmetric monoidal category $\mathbf{C}$, such that all unit maps $\eta_X : X \to \mathcal{T}X$ are monomorphisms (this is often the case in practice). Then, the identity monad on $\mathbf{C}$ is a central submonad of $\mathcal{T}$.

Example 2.33. Given a monoid M, let $\mathcal{T} = (M \times -)$ be the monad on $\mathbf{Set}$ from Example 2.12. Any submonoid S of $Z(M)$ induces a central submonad $(S \times -)$ of $\mathcal{T}$.

Example 2.34. Given a semiring R, consider the monad $\mathcal{T}_R$ from Example 2.25. Any subsemiring S of $Z(R)$ induces a central submonad $\mathcal{T}_S$ of $\mathcal{T}_R$.

Example 2.35. A notion of *central Lawvere subtheory* can be introduced in an obvious way. It induces a central submonad of the monad induced by the original Lawvere theory.

Example 2.36. The three commutative submonads identified in [JLMZ21b] are central submonads of the valuations monad $\mathbf{V}$ from Example 2.27, because each one of them is a commutative submonad of the centre of $\mathbf{V}$ [JMZ21].

Remark 2.37. Given an arbitrary monoid M (on $\mathbf{Set}$), there could be a commutative submonoid S of M that is not central (*i.e.*, its elements do not commute with all elements of M). The same holds for strong monads. For instance, let $M = \mathbb{D}_4$ (see Example 2.13) and let S be the submonoid of M that contains only the rotations (of which there are four). Then, S is a commutative submonoid that is not central. By taking the free monads induced by these monoids (see Example 2.12) on $\mathbf{Set}$, we get an example of a commutative submonad that is not central. Moreover, if we take $\mathbf{D}$ to be the full subcategory of $\mathbf{Set}$ whose objects have cardinality that is different from two, then $\mathbf{D}$ has a cartesian structure and the writer monads induced by S and M on $\mathbf{D}$ give an example of a non-centralisable strong monad that admits a commutative non-central submonad. In this situation, the identity monad on $\mathbf{D}$ gives an example of a central (commutative) submonad even though the ambient monad (induced by M) is not centralisable.

2.5 Computational Interpretation

In this section, we provide a computational interpretation of our ideas. We consider a simply-typed lambda calculus together with a strong monad $\mathcal{T}$ and a *central submonad* $\mathcal{S}$ of $\mathcal{T}$. We call this system the *Central Submonad Calculus (CSC)*. We describe its equational theories, formulate appropriate categorical models for it and we prove soundness, completeness and internal language results for our semantics.

2.5.1 Syntactic Structure of the Central Submonad Calculus

We begin by describing the types we use. The grammar of types (see Figure 2.1) are just the usual ones with one addition – we extend the grammar by adding the family of types $\mathcal{S}A$. The type $\mathcal{T}A$ represents the type of monadic computations for our monad $\mathcal{T}$ that produce values of type A (together with a potential side effect described by $\mathcal{T}$). The type $\mathcal{S}A$ represents the type of *central* monadic computations for our monad $\mathcal{T}$ that produce values of type A (together with a potential *central* side effect that is in the submonad $\mathcal{S}$). Some terms and formation rules can be expressed in the same way for types of the form $\mathcal{S}A$ or $\mathcal{T}A$ and in this case we simply write $\mathcal{X}A$ to indicate that $\mathcal{X}$ may range over $\{\mathcal{S}, \mathcal{T}\}$.

The grammar of terms and their formation rules are described in Figure 2.1. The first six rules in Figure 2.1 are just the usual formation rules for a simply-typed lambda calculus with pair types. Contexts are considered up to permutation and without repetition and all judgements we consider are implicitly closed under weakening (which is important when adding constants). The $\mathtt{ret}_{\mathcal{X}}\ M$ term is used as an introduction rule for the monadic types and it allows us to see the pure (*i.e.*, non-effectful) computation described by the term M as a monadic one. The term ιM allows us to view a *central* monadic computation as a monadic (not necessarily central) one. Semantically, it corresponds to applying the ι submonad inclusion we saw in previous sections. Finally, we have two terms for monadic sequencing that use the familiar `do`-notation. The monadic sequencing of two central computations remains central, which is represented via the $\mathtt{do}_{\mathcal{S}}$ terms; the $\mathtt{do}_{\mathcal{T}}$ terms are used for monadic sequencing of (not necessarily central) computations.

(Types) $A, B \ ::= 1 \mid A \to B \mid A \times B \mid \mathcal{S}A \mid \mathcal{T}A$

(Terms) $M, N \ ::= x \mid * \mid \lambda x^A.M \mid MN \mid \langle M, N\rangle$
$\mid \pi_i M \mid \mathtt{ret}_{\mathcal{S}}\ M \mid \iota M \mid \mathtt{ret}_{\mathcal{T}}\ M$
$\mid \mathtt{do}_{\mathcal{S}}\ x \leftarrow M;\ N \mid \mathtt{do}_{\mathcal{T}}\ x \leftarrow M;\ N$

$$\frac{}{\Gamma, x : A \vdash x : A} \qquad \frac{\Gamma \vdash M : A \to B \quad \Gamma \vdash N : A}{\Gamma \vdash MN : B}$$

$$\frac{}{\Gamma \vdash * : 1} \qquad \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x^A.M : A \to B} \qquad \frac{\Gamma \vdash M : A_1 \times A_2}{\Gamma \vdash \pi_i M : A_i}$$

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash N : B}{\Gamma \vdash \langle M, N\rangle : A \times B} \qquad \frac{\Gamma \vdash M : A}{\Gamma \vdash \mathtt{ret}_{\mathcal{X}}\ M : \mathcal{X}A}$$

$$\frac{\Gamma \vdash M : \mathcal{S}A}{\Gamma \vdash \iota M : \mathcal{T}A} \qquad \frac{\Gamma \vdash M : \mathcal{X}A \quad \Gamma, x : A \vdash N : \mathcal{X}B}{\Gamma \vdash \mathtt{do}_{\mathcal{X}}\ x \leftarrow M;\ N : \mathcal{X}B}$$

Figure 2.1 – Grammars and formation rules.

2.5.2 Equational Theories of the Central Submonad Calculus

Next, we describe equational theories for our calculus. We follow the vocabulary and the terminology in [MMDPR05] in order to formulate an appropriate notion of CSC-theory.

Definition 2.38 (CSC-theory)**.** A CSC-theory is an extension of the Central Submonad Calculus (see §2.5.1) with new ground types, new term constants (which we assume are well-formed in any context, including the empty one) and new equalities between types and between terms.

In a CSC-theory, we have four types of judgements: the judgement $\vdash A : \mathtt{type}$ indicates that A is a (simple) type; the judgement $\vdash A = B : \mathtt{type}$ indicates that types A and B are equal; the judgement $\Gamma \vdash M : A$ indicates that M is a well-formed term of type A in context Γ, as usual; finally, the judgement $\Gamma \vdash M = N : A$ indicates that the two well-formed terms M and N are equal.

Type judgements and term judgements are described in Figure 2.1 and type equality judgements in Figure 2.2. Following the principle of judgemental equality, we add type conversion rules in Figure 2.3. The rules in Figure 1.2 are the usual rules that describe the equational theory of the simply-typed lambda calculus. As often done by many authors, we implicitly identify terms that are α-equivalent. The rules for β-equivalence and η-equivalence are explicitly specified.

In Figure 2.4, we present the equational rules for monadic computation. The rules on the first three lines – *(ret.eq)*, *(do.eq)*, $(\mathcal{X}.\beta)$, $(\mathcal{X}.\eta)$, *($\mathcal{X}$.assoc)* – axiomatise the structure of a strong monad. Because of this, these rules are stated for both monads $\mathcal{T}$ and $\mathcal{S}$. The rules *(ι.mono)*, *($\iota\mathcal{S}$.ret)* and *($\iota\mathcal{S}$.comp)* are used to axiomatise the structure of $\mathcal{S}$ as a submonad of $\mathcal{T}$. Intuitively, these rules can be understood as specifying that central monadic computations can be seen as (general) monadic computations of the ambient monad $\mathcal{T}$. The remainder of the rules are used to axiomatise the behaviour of $\mathcal{S}$ as a *central* submonad of $\mathcal{T}$. The rule $(\mathcal{S}.central)$

$$\frac{\vdash A : \mathtt{type}}{\vdash A = A : \mathtt{type}} \qquad \frac{\vdash A = B : \mathtt{type}}{\vdash B = A : \mathtt{type}} \qquad \frac{\vdash A = B : \mathtt{type} \quad \vdash B = C : \mathtt{type}}{\vdash A = C : \mathtt{type}}$$

$$\frac{\vdash A = A' : \mathtt{type} \quad \vdash B = B' : \mathtt{type}}{\vdash A \times B = A' \times B' : \mathtt{type}} \qquad \frac{\vdash A = A' : \mathtt{type} \quad \vdash B = B' : \mathtt{type}}{\vdash A \to B = A' \to B' : \mathtt{type}}$$

$$\frac{\vdash A = B : \mathtt{type}}{\vdash \mathcal{X}A = \mathcal{X}B : \mathtt{type}}$$

Figure 2.2 – Equational rules for types.

$$\frac{\vdash A = B : \mathtt{type} \quad \vdash C = D : \mathtt{type} \quad \Gamma, x : A \vdash M : C}{\Gamma, x : B \vdash M : D}$$

$$\frac{\vdash A = B : \mathtt{type} \quad \vdash C = D : \mathtt{type} \quad \Gamma, x : A \vdash M = N : C}{\Gamma, x : B \vdash M = N : D}$$

Figure 2.3 – Type conversion rules.

is undoubtedly the most important one, because it ensures that central computations commute with any other (not necessarily central) computation when performing monadic sequencing with the $\mathcal{T}$ monad.

Example 2.39. Let us consider an example of a CSC-theory. Given a monoid (M, e, m) we now axiomatise the writer monad induced by M. A theory for this monad does not add any new types, but it adds constants for each element c of M: $\Gamma \vdash \mathtt{act}_{\mathcal{T}}(c) : \mathcal{T}1$. In this specific theory, we may think of the side-effect computed by monadic sequencing as being simply an element of M. The term $\mathtt{act}_{\mathcal{T}}(c)$ can be understood as performing the monoid multiplication on the right with argument c, *i.e.*, it applies the function $m(-, c)$ to whatever is the current state of the program.

Let S be a submonoid of the centre $Z(M)$ of M. This makes S a *central* submonoid of M (this can be defined in a similar way to central subgroups). We enrich the theory with the following constant and rule for each s in S:

$$\frac{}{\Gamma \vdash \mathtt{act}_{\mathcal{S}}(s) : \mathcal{S}1} \qquad \frac{}{\Gamma \vdash \iota\ \mathtt{act}_{\mathcal{S}}(s) = \mathtt{act}_{\mathcal{T}}(s) : \mathcal{T}1}$$

The application of $\mathtt{ret}_{\mathcal{X}}$ is equivalent to acting on the monoid data with the neutral element:

$$\frac{}{\Gamma \vdash \mathtt{ret}_{\mathcal{X}}\ * = \mathtt{act}_{\mathcal{X}}(e) : \mathcal{S}1}$$

Of course, the actions compose:

$$\frac{\Gamma \vdash M : \mathcal{X}A}{\begin{array}{c}\Gamma \vdash \mathtt{do}_{\mathcal{X}}\ * \leftarrow \mathtt{act}_{\mathcal{X}}(c);\ \mathtt{do}_{\mathcal{X}}\ * \leftarrow \mathtt{act}_{\mathcal{X}}(c');\ M \\ = \mathtt{do}_{\mathcal{X}}\ * \leftarrow \mathtt{act}_{\mathcal{X}}(m(c, c'));\ M : \mathcal{X}A\end{array}}$$

where we have used some (hopefully obvious) syntactic sugar. We write $\mathfrak{T}_M$ to refer to this theory.

$$\frac{\Gamma \vdash M = N \colon A}{\Gamma \vdash \mathtt{ret}_{\mathcal{X}}\ M = \mathtt{ret}_{\mathcal{X}}\ N \colon \mathcal{X}A}\ (ret.eq) \qquad \frac{\Gamma \vdash M = M' \colon \mathcal{X}A \quad \Gamma, x : A \vdash N = N' \colon \mathcal{X}B}{\Gamma \vdash \mathtt{do}_{\mathcal{X}}\ x \leftarrow M;\ N = \mathtt{do}_{\mathcal{X}}\ x \leftarrow M';\ N' \colon \mathcal{X}B}\ (do.eq)$$

$$\frac{\Gamma \vdash M \colon A \quad \Gamma, x : A \vdash N \colon \mathcal{X}B}{\Gamma \vdash \mathtt{do}_{\mathcal{X}}\ x \leftarrow \mathtt{ret}_{\mathcal{X}}\ M;\ N = N[M/x] \colon \mathcal{X}B}\ (\mathcal{X}.\beta) \qquad \frac{\Gamma \vdash M \colon \mathcal{X}A}{\Gamma \vdash \mathtt{do}_{\mathcal{X}}\ x \leftarrow M;\ \mathtt{ret}_{\mathcal{X}}\ x = M \colon \mathcal{X}A}\ (\mathcal{X}.\eta)$$

$$\frac{\Gamma \vdash M \colon \mathcal{X}A \quad \Gamma \vdash N \colon \mathcal{X}B \quad \Gamma, x : A, y : B \vdash P \colon \mathcal{X}C}{\Gamma \vdash \mathtt{do}_{\mathcal{X}}\ y \leftarrow (\mathtt{do}_{\mathcal{X}}\ x \leftarrow M;\ N);\ P = \mathtt{do}_{\mathcal{X}}\ x \leftarrow M;\ \mathtt{do}_{\mathcal{X}}\ y \leftarrow N;\ P \colon \mathcal{X}C}\ (\mathcal{X}.assoc)$$

$$\frac{\Gamma \vdash M : \mathcal{S}A \quad \Gamma \vdash N \colon \mathcal{T}B \quad \Gamma, x : A, y : B \vdash P : \mathcal{T}C}{\Gamma \vdash \mathtt{do}_{\mathcal{T}}\ x \leftarrow \iota M;\ \mathtt{do}_{\mathcal{T}}\ y \leftarrow N;\ P = \mathtt{do}_{\mathcal{T}}\ y \leftarrow N;\ \mathtt{do}_{\mathcal{T}}\ x \leftarrow \iota M;\ P \colon \mathcal{T}C}\ (\mathcal{S}.central)$$

$$\frac{\Gamma \vdash M = N : \mathcal{S}A}{\Gamma \vdash \iota M = \iota N : \mathcal{T}A}\ (\iota.mono) \qquad \frac{\Gamma \vdash M \colon A}{\Gamma \vdash \iota\ \mathtt{ret}_{\mathcal{S}}\ M = \mathtt{ret}_{\mathcal{T}}\ M \colon \mathcal{T}A}\ (\iota\mathcal{S}.ret)$$

$$\frac{\Gamma \vdash M \colon \mathcal{S}A \quad \Gamma, x : A \vdash N \colon \mathcal{S}B}{\Gamma \vdash \mathtt{do}_{\mathcal{T}}\ x \leftarrow \iota M;\ \iota N = \iota\ \mathtt{do}_{\mathcal{S}}\ x \leftarrow M;\ N \colon \mathcal{T}B}\ (\iota\mathcal{S}.comp)$$

Figure 2.4 – Equational rules for terms of monadic types of CSC.

Remark 2.40. As we have now seen, the equational theories of central submonads admit a presentation that is similar in spirit to that of the simply-typed λ-calculus. However, that is not the case with *the* centre of a strong monad. The reason is that the theory $\mathfrak{T}$ can introduce a central effect – one that commutes with all others – as a constant c that is not assigned the type $\mathcal{S}A$, but the type $\mathcal{T}A$, for some A. However, the centre, being the largest central submonad, must contain all such effects, so the constant c has to be equal to a term of the form $\iota c'$. One solution to this problem would be to use a more expressive logic and introduce a rule as follows (writing inline because of space): given $c \colon \mathcal{T}A$ and $x : A, y : B \vdash P \colon \mathcal{T}C$, such that $\forall N : \mathcal{T}B.\ \vdash \mathtt{do}_{\mathcal{T}}\ x \leftarrow c;\ \mathtt{do}_{\mathcal{T}}\ y \leftarrow N;\ P = \mathtt{do}_{\mathcal{T}}\ y \leftarrow N;\ \mathtt{do}_{\mathcal{T}}\ x \leftarrow c;\ P \colon \mathcal{T}C$ then $\exists c' : \mathcal{S}A.\ \vdash c = \iota c' \colon \mathcal{T}A$. However, the addition of such a rule seems unnecessary to prove our main point and it increases the complexity of the logic. Because of this, our choice is to focus on central submonads. Another reason to prefer central submonads over the centre is that they are more general and it is not required to identify *all* central effects (which would be the case for the centre). Overall, our choice for central submonads is motivated by the advantages they provide in terms of generality, simplicity and practicality of their equational theories compared to the centre.

Now that we have introduced theories, we explain how they can be translated into one another in an appropriate way.

Definition 2.41 (CSC-translation)**.** A *translation* V between two CSC-theories $\mathfrak{T}$ and $\mathfrak{T}'$ is a function that maps types of $\mathfrak{T}$ to types of $\mathfrak{T}'$ and terms of $\mathfrak{T}$ to terms of $\mathfrak{T}'$ that preserves the provability of all type judgements, term judgements, type equality judgements and term equality judgements. Moreover, such a translation is required to satisfy the following structural requirements on types:

$$V(1) = 1 \qquad V(\mathcal{T}A) = \mathcal{T}V(A) \qquad V(\mathcal{S}A) = \mathcal{S}V(B)$$
$$V(A \to B) = V(A) \to V(B) \qquad V(A \times B) = V(A) \times V(B)$$

and on terms:

$$V(*) = *$$
$$V(\lambda x^A.M) = \lambda x^A.V(M) \qquad V(MN) = V(M)V(N)$$
$$V(\langle M, N\rangle) = \langle V(M), V(N)\rangle \qquad V(\pi_i M) = \pi_i V(M)$$
$$V(\iota M) = \iota V(M) \qquad V(\mathtt{ret}_{\mathcal{X}}\ M) = \mathtt{ret}_{\mathcal{X}}\ V(M)$$
$$V(\mathtt{do}_{\mathcal{X}}\ x \leftarrow M;\ N) = \mathtt{do}_{\mathcal{X}}\ x \leftarrow V(M);\ V(N)$$

Remark 2.42. The above equations do not imply preservation of the relevant judgements for *constants*. Because of this, the first part of the definition also is necessary.

Of course, it is easy to see that CSC-theories and CSC-translations form a category. However, in order to precisely state our main result, we have to consider the 2-categorical structure of CSC-theories. Intuitively, we may view every CSC-theory as a category itself (with types as objects and terms as morphisms) and every CSC-translation as a functor that strictly preserves the relevant structure. Then, intuitively, an appropriate notion of a 2-morphism would be a natural transformation between such functors. This is made precise (in non-categorical terms) by our next definition.

Definition 2.43 (CSC-translation Transformation)**.** Given two CSC-theories $\mathfrak{T}$ and $\mathfrak{T}'$, and two CSC-translations V and V' between them, a CSC-*translation transformation* $\alpha : V \Rightarrow V'$ is a type-indexed family of term judgements $x : V(A) \vdash \alpha_A \colon V'(A)$ such that, for any valid judgement $x : A \vdash f : B$ in $\mathfrak{T}$

$$x : V(A) \vdash \alpha_B[V(f)/x] = V'(f)[\alpha_A/x] \colon V'(B)$$

also is derivable in $\mathfrak{T}'$.

> **Proposition 2.44.** CSC-*theories,* CSC-*translations and* CSC-*translation transformations form a* 2-*category* $\mathbf{Th}(\mathrm{CSC})$.

Proof. Direct with Definition 2.43. □

2.5.3 Categorical Models of CSC

Now we describe what are the appropriate categorical models for providing a semantic interpretation of our calculus.

Definition 2.45 (CSC-model)**.** A CSC-*model* is a cartesian closed category $\mathbf{C}$ equipped with both a strong monad $\mathcal{T}$ and a central submonad $\mathcal{S}^{\mathcal{T}}$ of $\mathcal{T}$ with submonad monomorphism written as $\iota^{\mathcal{T}} \colon \mathcal{S}^{\mathcal{T}} \Rightarrow \mathcal{T}$. We often use a quadruple $(\mathbf{C}, \mathcal{T}, \mathcal{S}^{\mathcal{T}}, \iota^{\mathcal{T}})$ to refer to a CSC-model.

We will soon show that CSC-models correspond to CSC-theories in a precise way. This correspondence covers CSC-translations too and for this we introduce our next definition.

Definition 2.46 (CSC-model Morphism)**.** Given two CSC-models $(\mathbf{C}, \mathcal{T}, \mathcal{S}^{\mathcal{T}}, \iota^{\mathcal{T}})$ and $(\mathbf{D}, \mathcal{M}, \mathcal{S}^{\mathcal{M}}, \iota^{\mathcal{M}})$, a CSC-*model morphism* is a strict cartesian closed functor $F : \mathbf{C} \to \mathbf{D}$ that satisfies the following additional coherence properties:

$$F(\mathcal{T}X) = \mathcal{M}(FX) \qquad F(\mathcal{S}^{\mathcal{T}}X) = \mathcal{S}^{\mathcal{M}}(FX)$$
$$F\iota^{\mathcal{T}}_X = \iota^{\mathcal{M}}_{FX} \qquad F\eta^{\mathcal{T}}_X = \eta^{\mathcal{M}}_{FX}$$
$$F\mu^{\mathcal{T}}_X = \mu^{\mathcal{M}}_{FX} \qquad F\tau^{\mathcal{T}}_{X,Y} = \tau^{\mathcal{M}}_{FX,FY}.$$

Notice that a CSC-model morphism *strictly* preserves all of the relevant categorical structure. This is done on purpose so that we can establish an exact correspondence with CSC-translations, which also strictly preserve the relevant structure. To match the notion of a CSC-translation transformation, we just have to consider natural transformations between CSC-model morphisms.

Proposition 2.47. *CSC-models, CSC-model morphisms and natural transformations between them form a 2-category* $\mathbf{Mod}(\mathrm{CSC})$.

Proof. Direct. □

2.5.4 Semantic Interpretation

Now we explain how to introduce a denotational semantics for our theories using our models. An interpretation of a CSC-theory $\mathfrak{T}$ in a CSC-model $\mathbf{C}$ is a function $[\![-]\!]$ that maps types of $\mathfrak{T}$ to objects of $\mathbf{C}$ and well-formed terms of $\mathfrak{T}$ to morphisms of $\mathbf{C}$. We provide the details below.

For each ground type G, we assume there is an appropriate corresponding object $[\![G]\!]$ of $\mathbf{C}$. The remaining types are interpreted as objects in $\mathbf{C}$ as follows: $[\![1]\!] \stackrel{\text{def}}{=} 1; [\![A \to B]\!] \stackrel{\text{def}}{=} [\![B]\!]^{[\![A]\!]}; [\![A \times B]\!] \stackrel{\text{def}}{=} [\![A]\!] \times [\![B]\!]; [\![\mathcal{S}A]\!] \stackrel{\text{def}}{=} \mathcal{S}\,[\![A]\!]; [\![\mathcal{T}A]\!] \stackrel{\text{def}}{=} \mathcal{T}\,[\![A]\!]$. Variable contexts $\Gamma = x_1 \colon A_1 \ldots x_n \colon A_n$ are interpreted as usual as $[\![\Gamma]\!] \stackrel{\text{def}}{=} [\![A_1]\!] \times \cdots \times [\![A_n]\!]$. Terms are interpreted as morphisms $[\![\Gamma \vdash M : A]\!] : [\![\Gamma]\!] \to [\![A]\!]$ of $\mathbf{C}$. When the context and the type of a term M are understood, then we simply write $[\![M]\!]$ as a shorthand for $[\![\Gamma \vdash M : A]\!]$. The interpretation of term constants and the terms of the simply-typed λ-calculus is defined in the usual way (details omitted). The interpretation of the monadic terms is given by:

$$
\begin{aligned}
[\![\Gamma \vdash \mathtt{ret}_{\mathcal{X}}\ M : \mathcal{X}A]\!] &= \eta^{\mathcal{X}}_{[\![A]\!]} \circ [\![M]\!] \\
[\![\Gamma \vdash \iota M : \mathcal{T}A]\!] &= \iota_{[\![A]\!]} \circ [\![M]\!] \\
[\![\Gamma \vdash \mathtt{do}_{\mathcal{X}}\ x \leftarrow M;\ N \colon \mathcal{X}B]\!] &= \mu^{\mathcal{X}}_{[\![B]\!]} \circ \mathcal{X}\,[\![N]\!] \circ \tau^{\mathcal{X}}_{[\![\Gamma]\!],[\![A]\!]} \circ \langle \mathrm{id}, [\![M]\!] \rangle
\end{aligned}
$$

where we use $\mathcal{X}$ to range over $\mathcal{T}$ or its central submonad $\mathcal{S}$.

Definition 2.48 (Soundness and Completeness)**.** An interpretation $[\![-]\!]$ of a CSC-theory $\mathfrak{T}$ in a CSC-model $\mathbf{C}$ is said to be *sound* if for any type equality judgement $\vdash A = B : \mathtt{type}$ in $\mathfrak{T}$, we have that $[\![A]\!] = [\![B]\!]$ in $\mathbf{C}$, and for any equality judgement $\Gamma \vdash M = N : A$ in $\mathfrak{T}$, we have that $[\![\Gamma \vdash M \colon A]\!] = [\![\Gamma \vdash N : A]\!]$ in $\mathbf{C}$. An interpretation $[\![-]\!]$ is said to be *complete* when $\vdash A = B : \mathtt{type}$ iff $[\![A]\!] = [\![B]\!]$ and $\Gamma \vdash M = N \colon A$ iff $[\![\Gamma \vdash M \colon A]\!] = [\![\Gamma \vdash N \colon A]\!]$. If, moreover, the interpretation is clear from context, then we may simply say that the model $\mathbf{C}$ itself is sound and complete for the CSC-theory $\mathfrak{T}$.

Remark 2.49. There are different definitions of what constitutes a "model" in the literature. For example, a "model" in [Cro94] corresponds to a sound interpretation in our sense.

Example 2.50. A categorical model for the CSC-theory $\mathfrak{T}_M$ of Example 2.39 is given by the category $\mathbf{Set}$ together with the writer monad $\mathcal{T} \stackrel{\text{def}}{=} (- \times M) \colon \mathbf{Set} \to \mathbf{Set}$ and the central

submonad $\mathcal{S} \stackrel{\text{def}}{=} (- \times S) \colon \mathbf{Set} \to \mathbf{Set}$. More specifically, the monad data for $\mathcal{T}$ is given by:

$$\begin{aligned}
&\eta_A \colon A \to A \times M :: a \mapsto (a, e) \\
&\mu_A \colon (A \times M) \times M \to A \times M :: ((a, c), c') \mapsto (a, m(c, c')) \\
&\tau_{A,B} \colon A \times (B \times M) \to (A \times B) \times M :: \\
&\qquad (a, (b, c)) \mapsto ((a, b), c)
\end{aligned}$$

and the monad data for $\mathcal{S}$ is defined in the same way by (co)restricting to the submonoid S. The interpretation of the term constants is given by:

$$\begin{aligned}
[\![\Gamma \vdash \mathtt{act}_{\mathcal{T}}(c) : \mathcal{T}1]\!] &\colon [\![\Gamma]\!] \to 1 \times M :: \gamma \mapsto (*, c) \\
[\![\Gamma \vdash \mathtt{act}_{\mathcal{S}}(c) : \mathcal{S}1]\!] &\colon [\![\Gamma]\!] \to 1 \times S :: \gamma \mapsto (*, c)
\end{aligned}$$

This interpretation of the theory $\mathfrak{T}_M$ is sound and complete.

2.5.5 Equivalence between Theories and Models

Our final result in this chapter is to show that CSC-theories and CSC-models are strongly related. To do this, we define the *syntactic* CSC*-model* $S(\mathfrak{T})$ of CSC-theory $\mathfrak{T}$, and the *internal language* $L(\mathbf{C})$ that maps a CSC-model $\mathbf{C}$ to its internal language viewed as a CSC-theory. These two assignments give rise to our desired equivalence (Theorem 2.60).

The Syntactic CSC-model. Assume throughout the subsection that we are given a CSC-theory $\mathfrak{T}$. We show how to construct a sound and complete model $S(\mathfrak{T})$ of $\mathfrak{T}$ by building its categorical data using the syntax provided by $\mathfrak{T}$.

Definition 2.51 (Syntactic Category)**.** Let $S(\mathfrak{T})$ be the category whose objects are the types of $\mathfrak{T}$ modulo type equality, *i.e.*, the objects are equivalence classes $[A]$ of types with $A' \in [A]$ iff $\vdash A' = A \colon \mathtt{type}$ in $\mathfrak{T}$. The morphisms $S(\mathfrak{T})([A], [B])$ are equivalence classes of judgements $[x : A \vdash f : B]$, where $(x : A' \vdash f' \colon B') \in [x : A \vdash f \colon B]$ iff $\vdash A' = A \colon \mathtt{type}$ and $\vdash B' = B \colon \mathtt{type}$ and $x : A \vdash f = f' \colon B$. Identities are given by $[x \colon A \vdash x \colon A]$ and composition is defined by

$$[y \colon B \vdash g \colon C] \circ [x \colon A \vdash f \colon B'] = [x \colon A \vdash g[f/y] \colon C],$$

with $B' \in [B]$.

Lemma 2.52. *The above definition is independent of the choice of representatives and the syntactic category $S(\mathfrak{T})$ is a well-defined cartesian closed category.*

Proof. Suppose given two morphisms $f \colon A \to B, g \colon B \to C$, and a choice $[x \colon A' \vdash f' \colon B'_f] = f$ and $[y \colon B'_g \vdash g' \colon C'] = g$. Note that $B = [B'_f] = [B'_g]$, and in particular $y \colon B'_f \vdash g' \colon C'$ is derivable with $[y \colon B'_g \vdash g' \colon C'] = [y \colon B'_f \vdash g' \colon C']$. Thus, $x \colon A' \vdash g'[f'/y] \colon C'$ is derivable. We then prove that the choice $[x \colon A' \vdash f' \colon B'_f] = f$ and $[y \colon B'_f \vdash g' \colon C'] = g$ does not matter. We consider now new term judgments for some terms f'' and g'' such that $[x \colon A' \vdash f' \colon B'_f] = [x \colon A'' \vdash f'' \colon B''_f]$ and $[y \colon B'_f \vdash g' \colon C'] = [y \colon B''_f \vdash g'' \colon C'']$. By definition, $[A'] = [A'']$,

$[B'_f] = [B''_f]$ and $[C'] = [C'']$, and we wish to prove that $[x\colon A' \vdash g'[f'/y]\colon C'] = [x\colon A'' \vdash g''[f''/y]\colon C'']$.

$$\Pi_2 = \left\{ \dfrac{\dfrac{x\colon A, y\colon B' \vdash g' = g''\colon C'}{x\colon A' \vdash \lambda y^{B'_f}.g' = \lambda y^{B''_f}.g''\colon B'_f \to C'} \quad x\colon A' \vdash f' = f''\colon C'}{x\colon A' \vdash (\lambda y^{B'_f}.g')f' = (\lambda y^{B''_f}.g'')f''\colon C'} \; (\lambda.eq) \right.$$

$$\Pi_1 = \left\{ \dfrac{\Pi_2 \quad \dfrac{x\colon A', y\colon B''_f \vdash g''\colon C' \quad x\colon A' \vdash f''\colon B''_f}{x\colon A' \vdash (\lambda y^{B''_f}.g'')f'' = g''[f''/y]\colon C'} \; (\lambda.\beta)}{x\colon A' \vdash (\lambda y^{B'_f}.g')f' = g''[f''/y]\colon C'} \; (trans) \right.$$

$$\dfrac{\dfrac{x\colon A', y\colon B'_f \vdash g'\colon C' \quad x\colon A' \vdash f'\colon C'}{x\colon A' \vdash g'[f'/y] = (\lambda y^{B'_f}.g')f'\colon C'} \; (\lambda.\beta) \qquad \Pi_1}{x\colon A' \vdash g'[f'/y] = g''[f''/y]\colon C'} \; (trans)$$

Thus, it is safe to define $g \circ f$ as $[x\colon A' \vdash g'[f'/y]\colon C']$.

Given a choice of A' in $[A]$, $[x\colon A' \vdash x\colon A']$ is the identity morphism for the type $[A]$. Considering $[x\colon A' \vdash f\colon B']$ and $[y\colon C' \vdash g\colon A']$, we have:

$$[x\colon A' \vdash f\colon B'] \circ [x\colon A' \vdash x\colon A'] = [x\colon A' \vdash f[x/x]\colon B'] = [x\colon A' \vdash f\colon B'],$$

and

$$[x\colon A' \vdash x\colon A'] \circ [y\colon C' \vdash g\colon A'] = [y\colon C' \vdash x[g/x]\colon A'] = [y\colon C' \vdash g\colon A'].$$

One can notice that, for example, $x\colon A' \vdash f\colon B'$ has conveniently be chosen with the right type A'. It is authorised, because we have proven above that the choice of representative does not matter in composition matters.

The cartesian closure is a usual result for a syntactic category from a simply-typed λ-calculus, and it is preserved in our context. □

Remark 2.53. Note that by using *Scott's trick* [Sco55] we can take quotients without having to go up higher in the class hierarchy, so foundational issues can be avoided.

Lemma 2.54 ([AB23]). *The following assignments:*

$$\begin{aligned}
\mathcal{T}([A]) &= [\mathcal{T}A] \\
\mathcal{T}([x\colon A \vdash f\colon B]) &= [y\colon \mathcal{T}A \vdash \mathtt{do}_{\mathcal{T}}\ x \leftarrow y;\ \mathtt{ret}_{\mathcal{T}}\ f\colon \mathcal{T}B] \\
\eta_{[A]} &= [x\colon A \vdash \mathtt{ret}_{\mathcal{T}}\ x\colon \mathcal{T}A] \\
\mu_{[A]} &= [x\colon \mathcal{T}\mathcal{T}A \vdash \mathtt{do}_{\mathcal{T}}\ y \leftarrow x;\ y\colon \mathcal{T}A] \\
\tau_{[A],[B]} &= [x\colon A \times \mathcal{T}B \vdash \mathtt{do}_{\mathcal{T}}\ y \leftarrow \pi_2 x;\ \mathtt{ret}_{\mathcal{T}}\ \langle \pi_1 x, y\rangle\colon \mathcal{T}(A \times B)]
\end{aligned}$$

are independent of the choice of representatives and define a strong monad $(\mathcal{T}, \eta, \mu, \tau)$ *on* $S(\mathfrak{T})$.

Lemma 2.55. *In a similar way to Lemma 2.52, we can define a strong monad $(\mathcal{S}, \eta^{\mathcal{S}}, \mu^{\mathcal{S}}, \tau^{\mathcal{S}})$ on $S(\mathfrak{T})$ by using the corresponding monadic primitives. Then, the assignment:*

$$\iota_{[A]} = [x\colon \mathcal{S}A \vdash \iota x\colon \mathcal{T}A]$$

is independent of the choice of representative and gives a strong submonad monomorphism $\iota\colon \mathcal{S} \Rightarrow \mathcal{T}$ that makes $\mathcal{S}$ a central submonad of $\mathcal{T}$.

Proof. In all the following proofs, we consider convenient members of equivalence classes, because the choice of representative does not change the result, thanks to Lemma 2.52.

We prove that ι is a submonad morphism:

$$\begin{aligned}
& \iota_A \circ \eta^{\mathcal{S}}_A \\
\overset{def.}{=}\ & [y\colon \mathcal{S}A \vdash \iota y\colon \mathcal{T}A] \circ [x\colon A \vdash \mathtt{ret}_{\mathcal{S}}\ x\colon \mathcal{S}A] \\
\overset{comp.}{=}\ & [x\colon A \vdash \iota\ \mathtt{ret}_{\mathcal{S}}\ x\colon \mathcal{T}A] \\
\overset{(\iota\mathcal{S}.ret)}{=}\ & [x\colon A \vdash \mathtt{ret}_{\mathcal{T}}\ x\colon \mathcal{T}A] \\
\overset{def.}{=}\ & \eta^{\mathcal{T}}_A
\end{aligned}$$

$$\begin{aligned}
& \mu^{\mathcal{T}}_A \circ \mathcal{T}\iota_A \circ \iota_{\mathcal{S}A} \\
\overset{def.}{=}\ & [z\colon \mathcal{T}\mathcal{T}A \vdash \mathtt{do}_{\mathcal{T}}\ y \leftarrow z;\ y\colon \mathcal{T}A] \\
& \circ [y'\colon \mathcal{T}\mathcal{S}A \vdash \mathtt{do}_{\mathcal{T}}\ x \leftarrow y';\ \mathtt{ret}_{\mathcal{T}}\ \iota x\colon \mathcal{T}\mathcal{T}A] \circ [x'\colon \mathcal{S}\mathcal{S}A \vdash \iota x'\colon \mathcal{T}\mathcal{S}A] \\
\overset{comp.}{=}\ & [x'\colon \mathcal{S}\mathcal{S}A \vdash \mathtt{do}_{\mathcal{T}}\ y \leftarrow (\mathtt{do}_{\mathcal{T}}\ x \leftarrow \iota x';\ \mathtt{ret}_{\mathcal{T}}\ \iota x);\ y\colon \mathcal{T}A] \\
\overset{(\mathcal{T}.assoc)}{=}\ & [x'\colon \mathcal{S}\mathcal{S}A \vdash \mathtt{do}_{\mathcal{T}}\ x \leftarrow \iota x';\ \mathtt{do}_{\mathcal{T}}\ y \leftarrow \mathtt{ret}_{\mathcal{T}}\ \iota x;\ y\colon \mathcal{T}A] \\
\overset{(\mathcal{T}.\beta)}{=}\ & [x'\colon \mathcal{S}\mathcal{S}A \vdash \mathtt{do}_{\mathcal{T}}\ x \leftarrow \iota x';\ \iota x\colon \mathcal{T}A] \\
\overset{(\iota\mathcal{S}.comp)}{=}\ & [x'\colon \mathcal{S}\mathcal{S}A \vdash \iota\ \mathtt{do}_{\mathcal{S}}\ x \leftarrow x';\ x\colon \mathcal{T}A] \\
\overset{comp.}{=}\ & [y\colon \mathcal{S}A \vdash \iota y\colon \mathcal{T}A] \circ [x'\colon \mathcal{S}\mathcal{S}A \vdash \mathtt{do}_{\mathcal{S}}\ x \leftarrow x';\ x\colon \mathcal{S}A] \\
\overset{def.}{=}\ & \iota_A \circ \mu^{\mathcal{S}}_A
\end{aligned}$$

$$\begin{aligned}
& \iota_{A\times B} \circ \tau^{\mathcal{S}}_{A,B} \\
\overset{def.}{=}\ & [x\colon \mathcal{S}(A\times B) \vdash \iota x\colon \mathcal{T}(A\times B)] \\
& \circ [z\colon A \times \mathcal{S}B \vdash \mathtt{do}_{\mathcal{S}}\ y \leftarrow \pi_2 z;\ \mathtt{ret}_{\mathcal{S}}\ \langle \pi_1 z, y\rangle\colon \mathcal{S}(A\times B)] \\
\overset{comp.}{=}\ & [z\colon A \times \mathcal{S}B \vdash \iota\,(\mathtt{do}_{\mathcal{S}}\ y \leftarrow \pi_2 z;\ \mathtt{ret}_{\mathcal{S}}\ \langle \pi_1 z, y\rangle) : \mathcal{T}(A\times B)] \\
\overset{(\iota\mathcal{S}.comp)}{=}\ & [z\colon A \times \mathcal{S}B \vdash \mathtt{do}_{\mathcal{T}}\ y \leftarrow \iota\ \pi_2 z;\ \iota\ \mathtt{ret}_{\mathcal{S}}\ \langle \pi_1 z, y\rangle\colon \mathcal{T}(A\times B)] \\
\overset{(\iota\mathcal{S}.ret)}{=}\ & [z\colon A \times \mathcal{S}B \vdash \mathtt{do}_{\mathcal{T}}\ y \leftarrow \iota\ \pi_2 z;\ \mathtt{ret}_{\mathcal{T}}\ \langle \pi_1 z, y\rangle\colon \mathcal{T}(A\times B)] \\
\overset{(\times.\beta)}{=}\ & [z\colon A \times \mathcal{S}B \vdash \mathtt{do}_{\mathcal{T}}\ y \leftarrow \pi_2\langle \pi_1 z, \iota\pi_2 z\rangle;\ \mathtt{ret}_{\mathcal{T}}\ \langle \pi_1\langle \pi_1 z, \iota\pi_2 z\rangle, y\rangle\colon \mathcal{T}(A\times B)] \\
\overset{comp.}{=}\ & [x\colon A \times \mathcal{T}B \vdash \mathtt{do}_{\mathcal{T}}\ y \leftarrow \pi_2 x;\ \mathtt{ret}_{\mathcal{T}}\ \langle \pi_1 x, y\rangle\colon \mathcal{T}(A\times B)] \\
& \circ [z\colon A \times \mathcal{S}B \vdash \langle \pi_1 z, \iota\pi_2 z\rangle\colon A \times \mathcal{T}B] \\
\overset{def.}{=}\ & \tau^{\mathcal{T}}_{A,B} \circ (A \times \iota_B)
\end{aligned}$$

Moreover, ι is a monomorphism because of the $(\iota.mono)$ rule.

Finally, $\mathcal{Z}$ is a central submonad of $\mathcal{T}$:

$$
\begin{aligned}
& dst_{A,B} \circ (\iota \times \mathcal{T}B) \\
\overset{def.+comp.}{=}\ & [z \colon \mathcal{S}A \times \mathcal{T}B \vdash \\
& \quad \mathtt{do}_{\mathcal{T}}\ x \leftarrow (\mathtt{do}_{\mathcal{T}}\ y \leftarrow \iota\ \pi_1 z;\ \mathtt{ret}_{\mathcal{T}}\ (\mathtt{do}_{\mathcal{T}}\ y' \leftarrow \pi_2 z;\ \mathtt{ret}_{\mathcal{T}}\ \langle y, y' \rangle))\ ;\ x \colon \mathcal{T}(A \times B)] \\
\overset{(\mathcal{T}.assoc)}{=}\ & [z \colon \mathcal{S}A \times \mathcal{T}B \vdash \\
& \quad \mathtt{do}_{\mathcal{T}}\ y \leftarrow \iota\ \pi_1 z;\ \mathtt{do}_{\mathcal{T}}\ x \leftarrow \mathtt{ret}_{\mathcal{T}}\ (\mathtt{do}_{\mathcal{T}}\ y' \leftarrow \pi_2 z;\ \mathtt{ret}_{\mathcal{T}}\ \langle y, y' \rangle)\ ;\ x \colon \mathcal{T}(A \times B)] \\
\overset{(\mathcal{T}.\beta)}{=}\ & [z \colon \mathcal{S}A \times \mathcal{T}B \vdash \mathtt{do}_{\mathcal{T}}\ y \leftarrow \iota\ \pi_1 z;\ \mathtt{do}_{\mathcal{T}}\ y' \leftarrow \pi_2 z;\ \mathtt{ret}_{\mathcal{T}}\ \langle y, y' \rangle \colon \mathcal{T}(A \times B)] \\
\overset{(\mathcal{S}.central)}{=}\ & [z \colon \mathcal{S}A \times \mathcal{T}B \vdash \mathtt{do}_{\mathcal{T}}\ y' \leftarrow \pi_2 z;\ \mathtt{do}_{\mathcal{T}}\ y \leftarrow \iota\ \pi_1 z;\ \mathtt{ret}_{\mathcal{T}}\ \langle y, y' \rangle \colon \mathcal{T}(A \times B)] \\
\overset{(\mathcal{T}.\beta)}{=}\ & [z \colon \mathcal{S}A \times \mathcal{T}B \vdash \\
& \quad \mathtt{do}_{\mathcal{T}}\ y' \leftarrow \pi_2 z;\ \mathtt{do}_{\mathcal{T}}\ x \leftarrow \mathtt{ret}_{\mathcal{T}}\ (\mathtt{do}_{\mathcal{T}}\ y \leftarrow \iota\ \pi_1 z;\ \mathtt{ret}_{\mathcal{T}}\ \langle y, y' \rangle)\ ;\ x \colon \mathcal{T}(A \times B)] \\
\overset{(\mathcal{T}.assoc)}{=}\ & [z \colon \mathcal{S}A \times \mathcal{T}B \vdash \\
& \quad \mathtt{do}_{\mathcal{T}}\ x \leftarrow (\mathtt{do}_{\mathcal{T}}\ y' \leftarrow \pi_2 z;\ \mathtt{ret}_{\mathcal{T}}\ (\mathtt{do}_{\mathcal{T}}\ y \leftarrow \iota\ \pi_1 z;\ \mathtt{ret}_{\mathcal{T}}\ \langle y, y' \rangle))\ ;\ x \colon \mathcal{T}(A \times B)] \\
\overset{comp.+def.}{=}\ & dst'_{A,B} \circ (\iota \times \mathcal{T}B)
\end{aligned}
$$

□

Now we can prove our completeness result.

Theorem 2.56 (Completeness). *The quadruple* $(S(\mathfrak{T}), \mathcal{T}, \mathcal{S}, \iota)$ *is a sound and complete* CSC*-model for the* CSC*-theory* $\mathfrak{T}$*.*

Proof. There exists an (obvious) interpretation $[\![-]\!]$ of $\mathfrak{T}$ into $S(\mathfrak{T})$ which follows the structure outlined in §2.5.4. Standard arguments then show that $\Gamma \vdash M = N \colon A$ in $\mathfrak{T}$ iff $[\![\Gamma \vdash M \colon A]\!] = [\![\Gamma \vdash N \colon A]\!]$ in $S(\mathfrak{T})$. □

Remark 2.57. Note that the obvious canonical interpretation of $\mathfrak{T}$ in $S(\mathfrak{T})$ is initial as one may expect: any sound interpretation of $\mathfrak{T}$ in a CSC-model $\mathbf{C}$ factorises uniquely through the canonical interpretation via a CSC-model morphism.

Internal Language. With completeness proven, we now wish to establish an internal language result.

Definition 2.58 (Internal Language)**.** Given a CSC-model $\mathbf{C}$, we define a CSC-theory $L(\mathbf{C})$ as follows:

- For each object A of $\mathbf{C}$ we add a ground type which we name A^*.
- Every ground type A^* is interpreted in $\mathbf{C}$ by setting $[\![A^*]\!] \overset{\text{def}}{=} A$. This uniquely determines an interpretation on all types.
- If A and B are two (not necessarily ground) types, we add a type equality $\vdash A = B \colon \mathtt{type}$ iff $[\![A]\!] = [\![B]\!]$.
- For every morphism $f \colon A \to B$ in $\mathbf{C}$, we add a term constant $\vdash c_f \colon A^* \to B^*$. Its interpretation in $\mathbf{C}$ is defined to be $[\![c_f]\!] \overset{\text{def}}{=} \mathrm{curry}(f \circ \cong) \colon 1 \to B^A$, *i.e.*, it is defined by currying the morphism f in the obvious way. This uniquely determines an interpretation on all well-formed terms.
- New term equality axioms $\Gamma \vdash M = N \colon B$ iff $[\![\Gamma \vdash M \colon B]\!] = [\![\Gamma \vdash N \colon B]\!]$.

Theorem 2.59. *For any* CSC*-model* $\mathbf{C}$ *the above definition gives a well-defined* CSC*-theory* $L(\mathbf{C})$*. Moreover, the model* $\mathbf{C}$ *is sound and complete for* $L(\mathbf{C})$*.*

Proof. Well-definedness is straightforward and follows by a simple induction argument using the fact that the semantic interpretation $[\![-]\!]$ defined in §2.5.4 is always sound. Completeness is then immediate by the last condition in Definition 2.58. □

Equivalence Theorem. Finally, we show that both the construction of the syntactic category and the assignment of the internal language give rise to appropriate equivalences.

Theorem 2.60. *The relationship between the internal language and the syntactic model enjoys the following properties in the 2-categories* $\mathbf{Mod}(\mathrm{CSC})$ *and* $\mathbf{Th}(\mathrm{CSC})$*, respectively:*

1. *For any* CSC*-model* $\mathbf{C}$*, we have that* $\mathbf{C} \simeq SL(\mathbf{C})$*,* i.e.*, there exist* CSC*-model morphisms* $F\colon \mathbf{C} \to SL(\mathbf{C})$ *and* $G\colon SL(\mathbf{C}) \to \mathbf{C}$ *such that* $F \circ G \cong \mathrm{id}$ *and* $\mathrm{id} \cong G \circ F$*.*
2. *For any* CSC*-theory* $\mathfrak{T}$*, we have that* $\mathfrak{T} \simeq LS(\mathfrak{T})$*,* i.e.*, there exist* CSC*-translations* $V\colon \mathfrak{T} \to LS(\mathfrak{T})$ *and* $W\colon LS(\mathfrak{T}) \to \mathfrak{T}$ *such that* $V \circ W \cong \mathrm{id}$ *and* $\mathrm{id} \cong W \circ V$*.*

Proof. Given $\mathbf{C}$ an object of $\mathbf{Mod}(\mathrm{CSC})$, we wish to prove that $\mathbf{C}$ is equivalent to $SL(\mathbf{C})$. To do so, we introduce two strict cartesian closed functors $F\colon \mathbf{C} \to SL(\mathbf{C})$ and $G\colon SL(\mathbf{C}) \to \mathbf{C}$, such that there are isomorphisms $\mathrm{id} \Rightarrow GF$ and $FG \Rightarrow \mathrm{id}$.

- F maps an object A of $\mathbf{C}$ to $[A^*]$. It maps a morphism $f\colon A \to B$ to $[x : A^* \vdash c_f x\colon B^*]$.
- G maps an object $[A]$ to $[\![A]\!]$, the interpretation of the type A in $\mathbf{C}$, because the choice of representative of $[A]$ does not change the interpretation. G maps a morphism $[x\colon A \vdash g\colon B]$ to $[\![x\colon A \vdash g\colon B]\!]$.

Then it is easy to check that $GF = \mathrm{id}$ and $FG = \mathrm{id}$. Therefore $\mathbf{C}$ is isomorphic to $SL(\mathbf{C})$. Furthermore, given a CSC-theory $\mathfrak{T}$, we wish to prove that $\mathfrak{T}$ is equivalent to $LS(\mathfrak{T})$. To do so, we introduce two CSC-translations $V\colon \mathfrak{T} \to LS(\mathfrak{T})$ and $W\colon LS(\mathfrak{T}) \to \mathfrak{T}$ such that there are isomorphic CSC-translation transformations $VW \Rightarrow \mathrm{id}$ and $\mathrm{id} \Rightarrow WV$.

- V maps a type A in $\mathfrak{T}$ to $[A]^*$, and term judgements $x\colon A \vdash f\colon B$ to $x\colon [A]^* \vdash c_{[x\colon A \vdash f\colon B]} x\colon [B]^*$.
- Observe that for each type A in $LS(\mathfrak{T})$, there is a type of the form $[B]^*$ such that $\vdash A = [B]^*\colon \mathtt{type}$ in $LS(\mathfrak{T})$. We define $W(A) \stackrel{\mathrm{def}}{=} B$ (the choice of B does not matter). Then, for term constants we define $W(\vdash c_{[x\colon A \vdash f\colon B]}\colon B^*) \stackrel{\mathrm{def}}{=} (\vdash \lambda x.f\colon A \to B)$ and this uniquely determines the action of W on the remaining terms (the choice of f does not matter).

Given a type A in $\mathfrak{T}$, $x\colon W(V(A)) \vdash x\colon A$ is derivable in $\mathfrak{T}$ because $\vdash W(V(A)) = A\colon \mathtt{type}$, and $\alpha_A\colon x\colon W(V(A)) \vdash x\colon A$ defines an isomorphic CSC-translation transformation: post-composing (resp. composing) it with $x\colon A \vdash x\colon W(V(A))$ gives $x\colon W(V(A)) \vdash x\colon W(V(A))$ (resp. $x\colon A \vdash x\colon A$). Given a type A' in $LS(\mathfrak{T})$, the same is true for $\beta_{A'} = x\colon A' \vdash x\colon V(W(A'))$. Thus, for every CSC-theory, $\mathfrak{T}$ is equivalent to $LS(\mathfrak{T})$. □

Remark 2.61. We introduced type equalities so that we can prove Theorem 2.60. This is also the approach taken in [MMDPR05] and without this, technical difficulties arise. Theory translations

are defined strictly (up to equality, not up to isomorphism) and in order to match this with the corresponding notion of model morphism, we use type equalities. Without type equalities, the symmetry within Theorem 2.60 can only be established if we make further changes. One potential solution would be to weaken the notion of theory translation by requiring that it preserves types up to type isomorphism (*i.e.*, make it strong instead of strict), but this is technically cumbersome.

2.6 Conclusion and Future Work

We showed that, under some mild assumptions, strong monads indeed admit a centre, which is a commutative submonad, and we provided three equivalent characterisations for the existence of this centre (Theorem 2.11) which also establish important links to the theory of premonoidal categories. In particular, every (canonically strong) monad on $\mathbf{Set}$ is centralisable (§2.2.1) and we showed that the same is true for many other categories of interest (§2.3.1) and we identified specific monads with interesting centres (§2.3.2). More generally, we considered central submonads and we provided a computational interpretation of our ideas (§2.5) which has the added benefit of allowing us to easily keep track of which monadic operations are central, *i.e.*, which effectful operations commute under monadic sequencing with any other (not necessarily central) effectful operation. We cemented our semantics by proving soundness, completeness and internal language results.

One direction for future work is to consider a theory of *commutants* or *centralisers* for monads (in the spirit of [Luc18, GF16]) and to develop a computational interpretation with the expected properties (soundness, completeness and internal language). Another opportunity for future work includes studying the relationship between the centres of strong monads and distributive laws. In particular, given two strong monads and a strong/commutative distributive law between them, can we show that the distributive law also holds for their centres (or for some central submonads)? If so, this would allow us to use the distributive law to combine not just the original monads, but their centres/central submonads as well. Moreover, the interaction of the centre with operations on monadic theories can be investigated.

Our definition of central submonads makes essential use of the notion of monomorphism of strong monads. Another possibility for future work is to investigate an alternative approach where we consider an appropriate class of factorisation systems instead of monomorphisms to define central submonads. Yet another possibility for future work is to investigate if central submonads of a given strong monad have some interesting poset structure.

A natural generalisation of monads is the notion of *arrows* – or *strong promonads*. A promonad is a monoid in the category of profunctors, and profunctors are to functors what relations are to functions. Arrows give then a more general framework to study computational effects, and are particularly meaningful for effects in reversible computing [ASvW+05, HKK18a]. A final direction for future work is the study of equational theories and internal language for arrows.

Chapter 3
Simply-typed Quantum Control

"Okay, this is a quantum computer, right? We barely know how it works, it's basically magic." — Black Mirror, S06E01.

Abstract

Quantum control is a recent notion in the literature, and many of its facets are still poorly understood, especially in terms of programming languages. Our goal is to build up solid foundations for the study of quantum control, syntactically and semantically. We provide syntax and semantics for a simply-typed calculus based on pattern-matching, developed to represent quantum reversible operations, and quantum control is ensured with the help of a quantum algebraic effect. To enforce reversibility, a syntactic notion of orthonormal basis is introduced, called here *orthogonal decomposition*. A denotational semantics and an equational theory are developed, and we prove that the former is complete with regard to the latter.

References. This work has been the focus of many conversations between Kostia Chardonnet, Robin Kaarsgaard, Benoît Valiron and the author. It has then been enriched for a paper coauthored with Kinnari Dave, Romain Péchoux and Vladimir Zamdzhiev, where this language aimed at quantum control is intergrated in a larger language, that also handles classical control. That paper is under submission.

3.1 Introduction

Quantum superposition is an important computational resource that is utilised by many quantum algorithms and protocols. Therefore, when designing quantum programming languages and quantum type systems, it is natural to consider how to introduce quantum superposition into the languages and systems under consideration. One approach, that we investigate in this chapter, aims to introduce quantum superposition as a principal feature of the language.

That is, we provide language features that allow us to form superpositions of terms (*i.e.* programs) in the language. In particular, instead of adopting a gate-level view of quantum computation, wherein the computational process is described through a series of low-level atomic gates, we take an approach that allows us to abstract away from such details and that allows us to focus on the linear-algebraic structure of the computation.

Quantum computing embodies several types of operations. The principal operations are the unitary maps, which are reversible transformations of quantum states – *e.g.*, when working with quantum circuits, *quantum gates* are reversible. State preparation is another kind of operation, that initialises one or several qubits. Finally, measurement *breaks* quantum superposition to obtain a classical result with a certain probability. For example, when measuring a quantum bit $\alpha\left|0\right\rangle + \beta\left|1\right\rangle$, the output can be 0 with probability $|\alpha|^2$ and 1 with probability $|\beta|^2$. It is known that, in the design of a quantum program, measurement can be deferred to the end of the execution [Cro12, NC10]. Using this principle of deferred measurement, each program is then divided into two separate parts: the first is entirely reversible, followed by a measurement at the end. It is then sensible to focus on *reversible* programming to design a quantum programming language.

The idea of reversible computation comes from Landauer and Bennett [Lan61, Ben73] with the analysis of its expressivity, and the relationship between irreversible computing and dissipation of energy. This leads to an interest in reversible computation [Ben00, ACG+20], both with a low-level approach [Car12, WSSD16, SM13], and from a high-level perspective [Lut86, YG07, YAG16, JS12, JS14, SVV18, YAG12, TA15, JKT18].

Reversible programming lies on the latter side of the spectrum, and two main approaches have been followed. Embodied by Janus [Lut86, YG07, Yok10, YAG16] and later R-CORE and R-WHILE [GKY19], the first one focuses on imperative languages whose control flow is inherently reversible – the main issue with this aspect being tests and loops. The other approach is concerned with the design of functional languages with structured data and related case-analysis, or *pattern-matching* [YAG12, TA15, JS14, SVV18, JKT18]. To ensure reversibility, strong constraints have to be established on the pattern-matching in order to maintain reversibility.

Those developments were utilised to introduce a programming language aimed at reversible quantum programming [SVV18], which is the work this chapter builds upon. The goal of that paper is more specific: it aims to formalise a programming language that performs *quantum control*. Quantum control, as opposed to classical control, is the ability to realise the control schemes – such as *if* statements or *while* loops – with quantum data, which means, a superposition of states. Informally, one can say that quantum control is allowing not only superposition of states, but also superposition of programs. A practical example of quantum control is the *quantum switch* [CDPV13], which works as follows: given two quantum states x and y, and two unitary operations U, V, the quantum switch performs the operation UV on $\left|y\right\rangle$ when x is in the state $\left|0\right\rangle$, and VU when x is in the state $\left|1\right\rangle$. In general, the obtained operation sends the state $(\alpha\left|0\right\rangle + \beta\left|1\right\rangle) \otimes \left|y\right\rangle$ to $\alpha\left|0\right\rangle \otimes (UV\left|y\right\rangle) + \beta\left|1\right\rangle \otimes (VU\left|y\right\rangle)$. Besides being mathematically feasible, the quantum switch is also doable in a lab [PMA+14, RRF+17].

The literature on quantum control is fairly recent, because it was thought to be not feasible in a realistic quantum computer. Since the introduction of the quantum switch, quantum control is starting to be studied with a programming language point of view [SVV18, VLRH23,

AM22, DCHPS23]. In some cases, the presentation lacks a proper denotational study; in some others, it lacks an operational account. One can summarise by saying that quantum control in programming languages lacks solid foundations, as much syntactically as mathematically. The work in this chapter is an early attempt to resolve this issue.

3.1.1 Related work

Reversible computation. Two successive papers [CLV21, CLV23] – that are also the focus of the next chapter – provide a categorical semantics of a reversible programming language based on Theseus. That language is closely related to the one presented in our work, however it only handles reversibility on classical data. More generally, [KAG17] details the structure of inverse categories used to interpret reversible programming. However, a category interpreting a quantum programming language also has to take into account quantum states, usually represented by isometries; this is why it makes sense to work with contractive maps as morphisms. The category with Hilbert spaces as objects and contraction as morphisms is not an inverse category. This shows that the work on classical reversible programming languages cannot be applied to our goal. Furthermore, reversible quantum programming is not a monadic effect over reversible programming: the sensible way of going from an inverse category to Hilbert spaces is the functor ℓ^2 [Heu13], which is not an adjoint functor, and thus cannot provide a monad.

Quantum control. Our work is based on the paper of Sabry, Valiron and Vizzotto [SVV18] where a functional reversible programming language is introduced and extended to a quantum programming language handling quantum control with recursive functions over lists. However, the denotational semantics given in that paper is not compositional, nor it is proven sound, nor adequate with regard to the operational semantics. In general, that paper lays out great ideas on how to work with quantum control, but with few proven statements. This chapter aims to provide stronger foundations, syntactical and mathematical. This will hopefully help tackle the question of the denotational semantics of quantum structural recursion, as introduced in the paper cited.

PUNQ [DCHPS23] is a programming language which is close to ours in some aspects: it relies on a notion of orthogonality between terms to form linear combinations, and its goal is to work with unitaries, to ensure quantum control. There are some differences: its design is based on linear logic and is closer to a linear λ-calculus; the base type is the one of *bits* – which is quickly generalised to *qubits* –, even the authors still work with a specific basis, while one would expect that working with qubits directly solves this issue (see discussion in §3.6). Finally, to ensure normalisation of well-typed terms, the authors introduce an orthogonality predicate, akin to one in [SVV18] and in this chapter. However, the former requires that the terms are computed to check whether they are orthogonal. This means that the type checking is not static, and therefore not necessarily efficient.

Similarly, there are many approaches to quantum control, but that do not ensure unitarity of operations, usually related to the λ-calculus [DCM22b, DCM22a, DCGMV19, AG05, CdVVV22]. These developments set themselves in the long list of papers revolving around algebraic λ-calculi [ADCV17, AD17, Vau09, SV09]. Alejandro Díaz-Caro has written a nice paper on that topic [DC22]. Note that these approaches struggle to scale to infinite dimensions.

Other programming languages handling quantum control have been introduced, such as Qunity [VLRH23]. Qunity is based on reversible pattern-matching [SVV18], like this chapter. However, Qunity assumes built-in unitary operations, and has then λ-abstractions that are not proven to be unitary operations. This last point is tackled with an *error handling* scheme, which does not appear to be suitable, and there is no semantics to show how this scheme would behave operationally.

3.1.2 Contribution

We provide a reversible programming language with simple types, inspired by the direct sum and tensor product of vector spaces, and natural numbers, to show that it is possible to work with infinite data types. This language relies on an orthogonality notion between values to define *linear combinations* of values to represent quantum superposition. A notion of orthonormal basis in the syntax is introduced, which helps prove that the abstractions are unitary operations.

Regarding the syntax, the presentation has been improved, some lemmas were fixed compared to previous presentations [SVV18, Cha23], and many more lemmas have been proven concerning the orthogonal decomposition.

This chapter is organised as follows: the first section (see §3.2) outlines the syntax of the language (see Figure 3.1, the grammar of terms, their typing rules, utilising orthogonality (see Definition 3.3) and orthogonal decomposition (see Definition 3.14). Then, substitutions are introduced (§3.2.3), described in a way that fits the language, allowing us to write our equivalent of β-reduction in a comprehensible manner. We then introduce an equational theory (§3.3.1), in the vein of the equational theory of the simply-typed λ-calculus, and we prove that it verifies a normalisation property. A later section (§3.4) is dedicated to introducing the mathematical notions and definition required to establish the denotational semantics, given in §3.5. Finally, we prove completeness of the denotational semantics with regard to this equational theory (§3.5.2).

3.1.3 Work of the author

Within this chapter, the author has contributed to the following points.

- A new kind of linear combinations of terms, which is more fit to ensure normalisation (see Figure 3.1). This required the further development of the type system with different conditions than are already present in the literature.
- A new definition of orthogonality (see Definition 3.3), close to the ones in [SVV18, Cha23, CLV21, CLV23, CSV23], but it fits a larger collection of terms of the language, including linear combinations and function application.
- Formalised *well-formed* substitutions (see Definition 3.25), to help the denotational semantics.
- An equational theory for the language (see §3.3.1), establishing stronger foundations to quantum control than in [SVV18] where there is a mix of equations and operational semantics.

- A compositional denotational semantics (see §3.5), which is sound and complete with regard to the equational theory.

3.2 The Language

In this section, we present the syntax of the programming language studied in this chapter dedicated to simply-typed quantum control. The quantum aspect of the language is provided as an algebraic effect with the introduction of linear combinations with the combinator Σ.

3.2.1 Syntax of the Language

$$
\begin{array}{llcl}
\text{(Ground types)} & A, B & ::= & \mathtt{I} \mid A \oplus B \mid A \otimes B \mid \mathtt{Nat} \\
\text{(Unitary types)} & T & ::= & A \leftrightarrow B \\
\text{(Basis Values)} & b & ::= & * \mid x \mid \mathtt{inj}_l\, b \mid \mathtt{inj}_r\, b \mid b \otimes b \mid \mathtt{zero} \mid \mathtt{S}\, b \\
\text{(Values)} & v & ::= & \Sigma_{i \in I}(\alpha_i \cdot b_i) \\
\text{(Expressions)} & e & ::= & * \mid x \mid \mathtt{inj}_l\, e \mid \mathtt{inj}_r\, e \mid e \otimes e \mid \mathtt{zero} \mid \mathtt{S}\, e \\
 & & & \quad \mid \Sigma_{i \in I}(\alpha_i \cdot e_i) \\
\text{(Unitaries)} & \omega & ::= & \{ \mid b_1 \leftrightarrow e_1 \mid \ldots \mid b_n \leftrightarrow e_n \mid \} \\
 & & & \quad \mid \omega \otimes \omega \mid \omega \oplus \omega \mid \omega \circ \omega \mid \omega^{-1} \mid \mathtt{ctrl}\, \omega \\
\text{(Terms)} & t & ::= & * \mid x \mid \mathtt{inj}_l\, t \mid \mathtt{inj}_r\, t \mid t \otimes t \mid \mathtt{zero} \mid \mathtt{S}\, t \\
 & & & \quad \mid \omega\ t \mid \Sigma_{i \in I}(\alpha_i \cdot t_i)
\end{array}
$$

Figure 3.1 – Syntax of simply-typed quantum control.

The syntax of the programming language studied in this chapter is described in a usual way, with grammars, such as the ones in Chapter 1 (see the grammars for the simply-typed λ-calculus in (1.1) and (1.2)). It is given in Figure 3.1.

Types. The ground types are given by a unit type $\mathtt{I}$ and the usual connectives $\oplus$ and $\otimes$, which are respectively called *direct sum* and *tensor product*. We also have the inductive type $\mathtt{Nat}$, as a witness that it is possible to work with infinite data types, and thus infinite-dimensional spaces in the model. We equip functions, called *unitaries* in this chapter, with a separate type, written $A \leftrightarrow B$ when A and B are two ground types. This double arrow notation is inherited from [SVV18], as a way of picturing that the operations are indeed reversible.

Terms. The terms of the language are given as follows:

- variables $x, y, z, \ldots$, given as elements of a set of variables $\mathtt{Var}$, assumed to be totally ordered;
- a term $*$ called the *unit* corresponding to the unit type $\mathtt{I}$;
- usual connectives for the direct sum, $\mathtt{inj}_l$ and $\mathtt{inj}_r$ which are respectively called the *left injector* and *right injector*;

- a connective corresponding to the tensor product, that is also written $\otimes$;
- terms for natural numbers, `zero` and the connective `S` that gives the successor of a term;
- the application of a unitary to a term, written $\omega\ t$ when ω is a unitary and t a term;
- finally, given a finite set of indices I – which could be a finite set of numbers $\{0, 1, 2, \ldots, n\}$ –, assumed to be totally ordered, a family of complex numbers $(\alpha_i)_{i\in I}$ and a family of terms $(t_i)_{i\in I}$, one can form the term $\Sigma(I, (\alpha_i)_i, (t_i)_i)$, representing the linear combination of the terms with complex scalars. This last term construction embodies the quantum effect of the language. In the rest of the chapter, we write $\Sigma_{i\in I}(\alpha_i \cdot t_i)$ for $\Sigma(I, (\alpha_i)_i, (t_i)_i)$ to make it more readable.

In quantum theory, a linear combination of vectors $\sum_{i\in I} \alpha_i \left|i\right\rangle$ is normalised if $\sum_{i\in I} \left|\alpha_i\right|^2 = 1$. The family of real numbers $(\left|\alpha_i\right|^2)_{i\in I}$ is then seen as a probability distribution. This work does not focus on the probabilistic aspect of quantum theory; however, we want to work with well-formed states, and thus normalised states. This is why we ensure later that a linear combination of terms is normalised. Throughout the chapter, a term $\Sigma_{i\in\{1,2\}}(\alpha_i \cdot t_i)$ might be written $\alpha_1 \cdot t_1 + \alpha_2 \cdot t_2$, regarded as syntactic sugar. In some examples, a term $\Sigma_{i\in\{*\}} 1 \cdot t$ can be written $1 \cdot t$ or even t for readability; but note that $\Sigma_{i\in\{*\}} 1 \cdot t$ and t are different terms in the syntax.

Remark 3.1. Since we are working with a programming language which handles complex numbers, we might want to ensure that the complex numbers are *computable* [Tur37]; this is not a terrible assumption, since the set of computable complex numbers keeps the structure of a field [Ric54]. However, we do not wish to focus on this point; we then assume that we work with complex numbers achievable in a given quantum hardware, and that those still form a field.

The terms that are unitary-free – in the sense that they do not contain any function application – and that do not involve linear combinations are called *basis values*. These terms are naturally classical, as opposed to quantum. Their name comes from the fact that we use them as a syntactic representation of the canonical basis of a Hilbert space, as introduced in §1.4.3. Note that basis values are totally ordered.

Values, on the other hand, are linear combinations of basis values. In a value $\sum_{i\in I}(\alpha_i \cdot b_i)$, we assume that the family $(b_i)_{i\in I}$ is an increasing sequence, and that none of the scalars are equal to 0; this allows us to work with unique normal forms later in the chapter. Since this definition is restrictive, we need to introduce a more general piece of syntax, which still does not include unitary application, called an *expression*. They are used as outputs of functions, that we introduce below.

Example 3.2 (Qubits)**.** In our presentation, the type of *qubits* is $\mathtt{I} \oplus \mathtt{I}$. The term $\mathtt{inj}_l\ *$ represents the quantum state $\left|0\right\rangle$ and $\mathtt{inj}_r\ *$, the other element of the canonical basis $\left|1\right\rangle$. The general state of a qubit is given by the term $\alpha \cdot (\mathtt{inj}_l\ *) + \beta \cdot (\mathtt{inj}_r\ *)$.This type $\mathtt{I} \oplus \mathtt{I}$ can also be seen as the type of quantum booleans. The basis value $\mathtt{inj}_l\ *$ represents *false* and $\mathtt{inj}_r\ *$, *true*.

Unitaries. Unitaries are firstly obtained by what we call *unitary abstractions*, written as a set of clauses $\{\ \mid\ b_1 \leftrightarrow e_1 \mid \ldots \mid\ b_n \leftrightarrow e_n\ \}$ or $\{\ \mid\ b_i \leftrightarrow e_i\ \}_{i\in I}$, given a family of basis values and a family of expressions, both indexed by a set I. We will see that several conditions have to be verified to ensure that this unitary actually performs a unitary operation (in other words,

a reversible operation between normalised states). The grammar for unitaries also contains operations such as the direct sum $\oplus$, the tensor product $\otimes$, the inverse $(-)^{-1}$ and the qubit control `ctrl`. These operations can be seen as syntactic sugar, since they can all be performed within unitary abstractions. Their presence in the grammar is an instance of operations at the level of unitaries, which are at a *higher order* than the operations at the ground level. We will see in the next chapter that, provided some conditions, any calculus can be put on top of unitaries.

Unitary abstractions can be seen as a mix of λ-abstractions and pattern-matching; the latter is sometimes written `case` or `match` in a functional programming language. We give some examples to make this intuition clearer.

$$\{|\, x \leftrightarrow x\} \qquad \{|\, x \otimes y \leftrightarrow y \otimes x\} \qquad \left\{ \begin{array}{lcl} |\, \mathtt{inj}_l\, x & \leftrightarrow & \mathtt{inj}_r\, x \\ |\, \mathtt{inj}_r\, y & \leftrightarrow & \mathtt{inj}_l\, y \end{array} \right\} \tag{3.1}$$

The example on the left performs the identity; the one in the middle swaps the two elements of a tensor product; and the last one swaps the two parts of a direct sum. Part of the conditions for a unitary abstraction to be well-typed, is to have the same variables on each side of a clause. This is verified by the examples above. We will see that all three examples are well-typed, with the typing rules described in the next section. Note that, in those examples, the terms on the right-hand side of the abstraction should be values, in the form of a sum. We have simplified notations here, because the sum would involve only one element.

A simple example that involves quantum superposition is the following:

$$\left\{ \begin{array}{lcl} |\, \mathtt{inj}_l\, * & \leftrightarrow & \frac{1}{\sqrt{2}} \cdot (\mathtt{inj}_l\, *) + \frac{1}{\sqrt{2}} \cdot (\mathtt{inj}_r\, *) \\ |\, \mathtt{inj}_r\, * & \leftrightarrow & \frac{1}{\sqrt{2}} \cdot (\mathtt{inj}_l\, *) - \frac{1}{\sqrt{2}} \cdot (\mathtt{inj}_r\, *) \end{array} \right\}$$

which operates the well-known *Hadamard* operator on a qubit, given below.

$$\begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{pmatrix} \tag{3.2}$$

This operation is significant in quantum computing: it is the one that allows one to introduce quantum superposition in a system, as well as entanglement, when followed by a controlled *not* operator. The Hadamard operator performs a *change of basis*. It is an important notion in linear algebra, and is an equally important notion in this chapter, where unitary abstractions are precisely defined as a change between two bases; the latter are introduced syntactically, and are more general than linear algebraic bases.

3.2.2 Types and Typing Rules

As usual, a *typing context* consists of a set of pairs of a variable and a type, written Δ and generated by $\Delta ::= \emptyset \mid \{x : A\} \cup \Delta$. A comma between two contexts represents the union of the contexts, *i.e.* $\Delta, \Delta' = \Delta \cup \Delta'$. The variables in a context Δ need to be all different to one another. We have two levels of judgements: the one for terms, where sequents are written $\Delta \vdash t : A$ and a typing judgement for isos, noted $\vdash_\omega \omega : A \leftrightarrow B$. Variables in a context Δ are strictly linear: given $\Delta \vdash t : A$, every element of Δ has to occur *exactly once* in the term t.

Orthogonality. We have seen in §1.4.3 that a quantum state has to be normalised. The first focus of this section to ensure normalisation through the type system. Let us recall that, in quantum computing, there are two necessary conditions for a superposition of states $\alpha\,|\varphi\rangle + \beta\,|\psi\rangle$ to be normalised. The first one is concerned with the probability distribution condition, $|\alpha|^2 + |\beta|^2 = 1$. Secondly, the vectors $|\varphi\rangle$ and $|\psi\rangle$ need to be orthogonal. Indeed, the following vector:

$$\frac{1}{\sqrt{2}}\,|0\rangle + \frac{1}{\sqrt{2}}\,|0\rangle = \sqrt{2}\,|0\rangle$$

is not normalised. Therefore, its corresponding term

$$\frac{1}{\sqrt{2}} \cdot (\mathtt{inj}_l\ *) + \frac{1}{\sqrt{2}} \cdot (\mathtt{inj}_l\ *)$$

should not be accepted by our type system. To do so, we introduce a notion of orthogonality for terms in our syntax (see Definition 3.3), in order to express later a typing rule ensuring normalisation.

Our orthogonality predicate is primarily based on direct sums and injections. Given two Hilbert spaces H_1 and H_2 and vectors $x_1 \in H_1$ and $x_2 \in H_2$, their respective injections into $H_1 \oplus H_2$, namely $(x_1, 0)$ and $(0, x_2)$, are orthogonal. Therefore, given two terms t_1 and t_2, we fix that their respective projections $\mathtt{inj}_l\ t_1$ and $\mathtt{inj}_r\ t_2$ are orthogonal.

Our orthogonality predicate is then generalised on all terms through congruence rules. The final rule presented in Definition 3.3 outlines the orthogonality obtained with a *change of basis*. For example, in the Hilbert space $\mathbb{C}^2$, the two vectors:

$$|+\rangle = \frac{1}{\sqrt{2}}\,|0\rangle + \frac{1}{\sqrt{2}}\,|1\rangle \quad \text{and} \quad |-\rangle = \frac{1}{\sqrt{2}}\,|0\rangle - \frac{1}{\sqrt{2}}\,|1\rangle$$

are orthogonal.

In the next definition, we introduce the predicate of orthogonality, written $\perp$. In particular, it means that given $t_1 \perp t_2$, a normalised linear combination of the two terms can be formed.

Definition 3.3 (Orthogonality)**.** We introduce a symmetric binary relation $\perp$ on terms. Given two terms t_1, t_2, we have $t_1 \perp t_2$ if it can be derived inductively with the rules below; when $t_1 \perp t_2$ can be derived, we say that t_1 and t_2 are orthogonal. The relation $\perp$ is defined as the smallest symmetric relation such that:

$$\frac{}{\mathtt{inj}_l\ t_1 \perp \mathtt{inj}_r\ t_2} \quad \frac{}{\mathtt{zero} \perp \mathtt{S}\ t} \quad \frac{t_1 \perp t_2}{\mathtt{S}\ t_1 \perp \mathtt{S}\ t_2} \quad \frac{t_1 \perp t_2}{t \otimes t_1 \perp t' \otimes t_2} \quad \frac{t_1 \perp t_2}{t_1 \otimes t \perp t_2 \otimes t'}$$

$$\frac{t_1 \perp t_2}{\mathtt{inj}_l\ t_1 \perp \mathtt{inj}_l\ t_2} \quad \frac{t_1 \perp t_2}{\mathtt{inj}_r\ t_1 \perp \mathtt{inj}_r\ t_2} \quad \frac{t_1 \perp t_2}{\omega\ t_1 \perp \omega\ t_2} \quad \frac{\forall i \in I, t \perp t_i}{t \perp \Sigma_{i\in I}(\alpha_i \cdot t_i)}\ (\star)$$

$$\frac{\forall i \in I, t \perp t_i \quad \alpha_* = 0 \quad t_* = t}{t \perp \Sigma_{i\in I\cup\{*\}}(\alpha_i \cdot t_i)} \quad \frac{\forall i \neq j \in I, t_i \perp t_j \quad J, K \subseteq I \quad \sum_{i\in J\cap K} \bar{\alpha_i}\beta_i = 0}{\Sigma_{j\in J}(\alpha_j \cdot t_j) \perp \Sigma_{k\in K}(\beta_k \cdot t_k)}\ (\star)$$

Remark 3.4. Among the two inference rules marked by a $\star$ above, it could seem like the last one implies the first one. However, we remind that Σ is a contructor, and therefore t and $\Sigma_{j\in J}(\alpha_j \cdot t_j)$ are distinct cases of the grammar.

Remark 3.5 (Orthogonality of variables). Given two variables x and y, the terms x and y are not orthogonal. The main reason is that they could be instantiated with the same value. On the other hand, $\mathtt{inj}_l\ x$ and $\mathtt{inj}_r\ y$ are orthogonal, for example.

Remark 3.6. We recall the third unitary presented in (3.1), which swaps $\mathtt{inj}_l$ and $\mathtt{inj}_r$:

$$\left\{ \begin{array}{lcl} \mid \mathtt{inj}_l\ x & \leftrightarrow & \mathtt{inj}_r\ x \\ \mid \mathtt{inj}_r\ y & \leftrightarrow & \mathtt{inj}_l\ y \end{array} \right\}$$

and we call this unitary ω. The term $\omega\ (\mathtt{inj}_l\ *)$ represents the application of the unitary ω to $\mathtt{inj}_l\ *$. We later show within our equational theory (see §3.3), that the terms $\omega\ (\mathtt{inj}_l\ *)$ and $\mathtt{inj}_r\ *$ are equal, as expected. However, we cannot derive that $\omega\ (\mathtt{inj}_l\ *)$ and $\mathtt{inj}_l\ *$ are orthogonal with the rules of orthogonality given above (see Definition 3.3), even if $\mathtt{inj}_l\ *$ and $\mathtt{inj}_r\ *$ are orthogonal. This is because we wish to be able to derive orthogonality *statically*.

Lemma 3.7. *If $t \perp t'$, then t and t' are different terms.*

Note that orthogonality holds without any typing rules or notion of type. Figure 3.2 introduces the typing rules for expressions, and therefore for basis values and values, thanks to the notion of orthogonality.

$$\frac{}{\emptyset \vdash *\colon \mathtt{I},} \qquad \frac{}{x\colon A \vdash x\colon A,} \qquad \frac{\Delta_1 \vdash e_1\colon A \quad \Delta_2 \vdash e_2\colon B}{\Delta_1, \Delta_2 \vdash e_1 \otimes e_2\colon A \otimes B,}$$

$$\frac{\Delta \vdash e\colon A}{\Delta \vdash \mathtt{inj}_l\ e\colon A \oplus B,} \qquad \frac{\Delta \vdash e\colon B}{\Delta \vdash \mathtt{inj}_r\ e\colon A \oplus B,}$$

$$\frac{}{\vdash \mathtt{zero}\colon \mathtt{Nat},} \qquad \frac{\Delta \vdash e\colon \mathtt{Nat}}{\Delta \vdash \mathtt{S}\ e\colon \mathtt{Nat},}$$

$$\frac{\Delta \vdash e_i\colon A \quad \Sigma_i |\alpha_i|^2 = 1 \quad \forall i \neq j, e_i \perp e_j}{\Delta \vdash \Sigma_i(\alpha_i \cdot e_i)\colon A.}$$

Figure 3.2 – Typing rules of (basis) values and expressions.

Once we know how basis values are formed in a certain type, we can discuss some orthogonality properties among specific types. In linear algebra, given a vector space and an orthogonal basis B of that space, if two elements of the basis are not orthogonal, then they are equal. Our case is more subtle, because, for example, $\mathtt{inj}_r\ (\mathtt{inj}_l\ *)$ and $\mathtt{inj}_r\ x$ are not equal, but also not orthogonal. We will see, later in this chapter, that they are linked by substitution.

Example 3.8. The values $\mathtt{inj}_l\ *$ and $\mathtt{inj}_r\ *$ are orthogonal, but of course, they are both not orthogonal to $\frac{1}{\sqrt{2}} \cdot (\mathtt{inj}_l\ *) - \frac{1}{\sqrt{2}} \cdot (\mathtt{inj}_r\ *)$.

Orthogonal decomposition. This motivates a syntactic definition for a basis, containing expressions of the language. Given a set of orthogonal expressions, we want to ensure that this set spans the whole type, so that it can be seen as an orthonormal basis. This is given by the

notion of *orthogonal decomposition*. We first introduce this notion in Definition 3.9 with basis values only, and we then extend it to expressions in general in Definition 3.14.

Before outlying the definitions of orthogonal decompositions, we introduce some notations. Given a set $S = \{e_1 \otimes e'_1, \ldots, e_n \otimes e'_m\}$, we define $\pi_1(S) = \{e_1, \ldots, e_n\}$ and $\pi_2(S) = \{e'_1, \ldots, e'_m\}$. Finally, we define S^1_e and S^2_e respectively as $\{e' \mid e \otimes e' \in S\}$ and $\{e' \mid e' \otimes e \in S\}$.

Definition 3.9 (Orthogonal Decomposition)**.** We introduce a predicate $\mathrm{OD}_A(-)$ on finite sets of basis values. Given a finite set of values S, $\mathrm{OD}_A S$ holds if it can be derived with the following rules. The predicate is defined inductively as the smallest predicate such that:

$$\frac{}{\mathrm{OD}_A(\{x\})} \qquad \frac{}{\mathrm{OD}_{\mathtt{I}}(\{*\})} \qquad \frac{\mathrm{OD}_A S \qquad \mathrm{OD}_B T}{\mathrm{OD}_{A\oplus B}(\{\mathtt{inj}_l\ s \mid s \in S\} \cup \{\mathtt{inj}_r\ t \mid t \in T\})}$$

$$\frac{\mathrm{OD}_{\mathtt{Nat}} S}{\mathrm{OD}_{\mathtt{Nat}}(\{\mathtt{zero}\} \cup \{\mathtt{S}\ s \mid s \in S\})} \qquad \frac{\mathrm{OD}_B(\pi_2(S)) \text{ and } \forall b \in \pi_2(S), \mathrm{OD}_A(S^2_b)}{\mathrm{OD}_{A\otimes B} S}$$

$$\frac{\mathrm{OD}_A(\pi_1(S)) \text{ and } \forall b \in \pi_1(S), \mathrm{OD}_B(S^1_b)}{\mathrm{OD}_{A\otimes B} S}$$

To simplify the notations in coming proofs, we will write $S \boxplus T$ for the set $\{\mathtt{inj}_l\ s \mid s \in S\} \cup \{\mathtt{inj}_r\ t \mid t \in T\}$, and $S^{\oplus 0}$ for the set $\{\mathtt{zero}\} \cup \{\mathtt{S}\ s \mid s \in S\}$. We adopt functional programming convention regarding parentheses: when readable, we write $\mathrm{OD}_A S$ instead of $\mathrm{OD}_A(S)$. Also, we call OD the predicate introduced above in general, without precision of type, to facilitate the later discussions.

Example 3.10. Following Example 3.2, we have $\mathrm{OD}_{\mathtt{I}\oplus\mathtt{I}}\{\mathtt{inj}_l *, \mathtt{inj}_r *\}$. The two qubits $|0\rangle$ and $|1\rangle$ are indeed an orthonormal basis for qubit states.

Remark 3.11. Note that the precondition to derive $\mathrm{OD}_{A\otimes B} S$ cannot be simplified: there are sets S such that $\mathrm{OD}_A(\pi_1(S))$ and for all $b \in \pi_1(S)$, $\mathrm{OD}_B(S^1_b)$ but not all $b \in \pi_2(S)$ is such that $\mathrm{OD}_B(S^2_b)$, *e.g.* $S = \{(\mathtt{inj}_l *) \otimes y, (\mathtt{inj}_r\ x) \otimes (\mathtt{inj}_l *), (\mathtt{inj}_r\ x) \otimes (\mathtt{inj}_r *)\}$ with the type $(\mathtt{I} \oplus (\mathtt{I} \oplus \mathtt{I})) \otimes (\mathtt{I} \otimes \mathtt{I})$.

Example 3.12. Given any type A, we have $\mathrm{OD}_A\{x\}$, where x is a variable. Since it is a variable, it can be substituted with any terms of type A, therefore it *parses* the whole type. We make precise the notion of subsitution for our syntax in §3.2.3.

Example 3.13. We have $\mathrm{OD}_{\mathtt{Nat}}\{\mathtt{zero}, \mathtt{S}\ \mathtt{zero}, \mathtt{S}\ \mathtt{S}\ x\}$. Indeed, any closed basis value of type `Nat` is either `zero`, or `S zero`, or `S S` b.

The predicate OD_A defined above ensures that a finite set S of values represents an orthonormal basis of A, that we can view as the canonical basis. Note that even for `Nat`, the set representing the basis is finite, *e.g.* $\{\mathtt{zero}, \mathtt{S}\ \mathtt{zero}, \mathtt{S}\ (\mathtt{S}\ x)\}$. With knowledge of linear algebra, one can say that the bases represented by OD are not all the possible orthonormal bases. A change of basis through a unitary matrix also provides an orthonormal basis. This is the purpose of the next definition.

Definition 3.14. We extend the previous definition to general values. $\mathrm{OD}^{ext}_A(-)$ is a predicate on finite sets of values, and $\mathrm{OD}^{ext}_A S$ holds if it can be derived from the following rule.

$$\frac{}{\mathrm{OD}^{ext}_A(\{x\})} \qquad \frac{}{\mathrm{OD}^{ext}_{\mathtt{I}}(\{*\})} \qquad \frac{\mathrm{OD}^{ext}_A S \qquad \mathrm{OD}^{ext}_B T}{\mathrm{OD}^{ext}_{A\oplus B}(\{\mathtt{inj}_l\ e \mid e \in S\} \cup \{\mathtt{inj}_r\ e \mid e \in T\})}$$

$$\frac{\mathrm{OD}^{ext}_{\mathtt{Nat}} S}{\mathrm{OD}^{ext}_{\mathtt{Nat}}(\{\mathtt{zero}\} \cup \{\mathtt{S}\ v \mid v \in S\})} \qquad \frac{\mathrm{OD}^{ext}_B(\pi_2(S)) \text{ and } \forall e \in \pi_2(S), \mathrm{OD}^{ext}_A(S^2_e)}{\mathrm{OD}^{ext}_{A\otimes B} S}$$

$$\frac{\mathrm{OD}^{ext}_A(\pi_1(S)) \text{ and } \forall e \in \pi_1(S), \mathrm{OD}^{ext}_B(S^1_e)}{\mathrm{OD}^{ext}_{A\otimes B} S} \qquad \frac{\mathrm{OD}^{ext}_A S \quad (\alpha_{e,e'})_{(e,e')\in S\times S} \text{ is a unitary matrix}}{\mathrm{OD}^{ext}_A(\{\Sigma_{e'\in S}(\alpha_{e,e'} \cdot e') \mid e \in S\})}$$

Given $\mathrm{OD}^{ext}_A S$, we say that S is an *orthogonal decomposition* of type A.

To simplify notations, we write S^α for the set $\{\Sigma_{e'\in S}(\alpha_{e,e'} \cdot e') \mid e \in S\}$.

Example 3.15. Following Example 3.2, we have

$$\mathrm{OD}^{ext}_{\mathtt{I}\oplus\mathtt{I}}\left\{\frac{1}{\sqrt{2}} \cdot \mathtt{inj}_l * + \frac{1}{\sqrt{2}} \cdot \mathtt{inj}_r *, \quad \frac{1}{\sqrt{2}} \cdot \mathtt{inj}_l * - \frac{1}{\sqrt{2}} \cdot \mathtt{inj}_r *\right\}.$$

The two qubits $|+\rangle$ and $|-\rangle$ are indeed an orthonormal basis for qubit states.

Example 3.16. Following Example 3.13 and the previous example, we have:

$$\mathrm{OD}^{ext}_{\mathtt{Nat}}\left\{\frac{1}{\sqrt{2}}\mathtt{zero} + \frac{1}{\sqrt{2}}(\mathtt{S}\ \mathtt{zero}), \quad \frac{1}{\sqrt{2}}\mathtt{zero} - \frac{1}{\sqrt{2}}(\mathtt{S}\ \mathtt{zero}), \quad \mathtt{S}\ \mathtt{S}\ x\right\}.$$

The predicate OD is defined without the help of orthogonality (see Definition 3.3), but there is a link: the elements of an orthogonal decomposition, are in particular pairwise orthogonal.

Lemma 3.17 (OD implies $\perp$). *Given* $\mathrm{OD}^{ext}_A S$, *for all* $t_1 \neq t_2 \in S$, $t_1 \perp t_2$.

Proof. The proof is done by induction on OD.

- $\mathrm{OD}^{ext}_A\{x\}$. There is no pair of different terms in $\{x\}$.
- $\mathrm{OD}^{ext}_{\mathtt{I}}\{*\}$. There is no pair of different terms in $\{*\}$.
- $\mathrm{OD}^{ext}_{A\oplus B} S \boxplus T$. There are several cases: either both terms are of the form $\mathtt{inj}_l\ -$, namely $t_1 = \mathtt{inj}_l\ t'_1$ and $t_2 = \mathtt{inj}_l\ t'_2$, with t'_1 and t'_2 in S, in which case the induction hypothesis on $\mathrm{OD}^{ext}_A S$ gives that $t'_1 \perp t'_2$, and thus $\mathtt{inj}_l\ t'_1 \perp \mathtt{inj}_l\ t'_2$; or both are of the $\mathtt{inj}_r\ -$, the case is similar with the induction hypothesis on $\mathrm{OD}^{ext}_B T$; or $t_1 = \mathtt{inj}_l\ t'_1$ and $t_2 = \mathtt{inj}_r\ t'_2$, then we have directly $\mathtt{inj}_l\ t'_1 \perp \mathtt{inj}_r\ t'_2$.
- $\mathrm{OD}^{ext}_{A\otimes B} S$. Both cases are similar. Suppose that $\mathrm{OD}^{ext}_A \pi_1(S)$. We know that $t_1 = t'_1 \otimes t''_1$ and $t'_2 \otimes t''_2$. The induction hypothesis on $\mathrm{OD}^{ext}_A \pi_1(S)$ gives that $t'_1 \perp t'_2$ and thus $t'_1 \otimes t''_1 \perp t'_2 \otimes t''_2$.
- $\mathrm{OD}^{ext}_{\mathtt{Nat}} S^{\oplus 0}$. If one of t_1 or t_2 is $\mathtt{zero}$, the conclusion is direct; else, $t_1 = \mathtt{S}\ t'_1$ and $t_2 = \mathtt{S}\ t'_2$ and the induction hypothesis on $\mathrm{OD}^{ext}_{\mathtt{Nat}} S$ gives that $t'_1 \perp t'_2$ and thus $\mathtt{S}\ t'_1 \perp \mathtt{S}\ t'_2$.
- $\mathrm{OD}^{ext}_A S^\alpha$. By induction hypothesis, all the terms in S are pairwise orthogonal; and the matrix α is unitary, which means that the inner product of its columns is zero when the columns are different, which ensures that two different terms in S^α are orthogonal.

□

Note that while orthogonality ensures non-overlapping, it does not ensure exhaustivity, only OD_A does. The next lemma details the exhaustivity of OD_A, and is a consequence of a later result, that uses the notion of substitution (see §3.2.3 and Lemma 3.33).

Proposition 3.18. *If* $\mathrm{OD}_A^{ext} S$ *and* $\Delta \vdash e : A$*, then there exists* $e' \in S$ *such that* $\neg(e \perp e')$.

This notion of orthogonal decomposition allows us to introduce *unitary* abstractions in our syntax. A basic unitary has the form $\{ \mid b_1 \leftrightarrow e_1 \mid \ldots \mid b_n \leftrightarrow e_n \}$ and is well-typed only if $(b_i)_i$ forms a basis and $(e_i)_i$ also forms a basis; we then retrieve the intuition that a unitary should be equivalent to a change of orthonormal basis. The typing rules for deriving unitaries and unitary operations are detailed in Figure 3.3.

$$\frac{\begin{array}{c}\Delta_i \vdash b_i : A \quad \mathrm{OD}_A\{b_1, \ldots, b_n\} \\ \Delta_i \vdash e_i : B \quad \mathrm{OD}_B^{ext}\{e_1, \ldots, e_n\}\end{array}}{\vdash_\omega \{ \mid b_1 \leftrightarrow e_1 \mid \ldots \mid b_n \leftrightarrow e_n \} : A \leftrightarrow B,} \qquad \frac{\vdash_\omega \omega : A \leftrightarrow B}{\vdash_\omega \omega^{-1} : B \leftrightarrow A,}$$

$$\frac{\vdash_\omega \omega_1 : A \leftrightarrow B \quad \vdash_\omega \omega_2 : B \leftrightarrow C}{\vdash_\omega \omega_2 \circ \omega_1 : A \leftrightarrow C,} \qquad \frac{\vdash_\omega \omega_1 : A_1 \leftrightarrow B_1 \quad \vdash_\omega \omega_2 : A_2 \leftrightarrow B_2}{\vdash_\omega \omega_1 \otimes \omega_2 : A_1 \otimes A_2 \leftrightarrow B_1 \otimes B_2,}$$

$$\frac{\vdash_\omega \omega_1 : A_1 \leftrightarrow B_1 \quad \vdash_\omega \omega_2 : A_2 \leftrightarrow B_2}{\vdash_\omega \omega_1 \oplus \omega_2 : A_1 \oplus A_2 \leftrightarrow B_1 \oplus B_2,} \qquad \frac{\vdash_\omega \omega : A \leftrightarrow A}{\vdash_\omega \mathtt{ctrl}\ \omega : (\mathtt{I} \oplus \mathtt{I}) \otimes A \leftrightarrow (\mathtt{I} \oplus \mathtt{I}) \otimes A.}$$

Figure 3.3 – Typing rules of unitaries.

Terms in our syntax are either expressions or an application of a unitary to a term, in a similar style to the λ-calculus; however, we have seen that abstractions are considered separately in the grammar. The typing rules are the same as the one for values given in Figure 3.2, with the addition of a rule that enables the application of a unitary to a term. The details are in Figure 3.4.

$$\frac{}{\emptyset \vdash * : \mathtt{I},} \qquad \frac{}{x : A \vdash x : A,} \qquad \frac{\Delta_1 \vdash t_1 : A \quad \Delta_2 \vdash t_2 : B}{\Delta_1, \Delta_2 \vdash t_1 \otimes t_2 : A \otimes B,}$$

$$\frac{\Delta \vdash t : A}{\Delta \vdash \mathtt{inj}_l\ t : A \oplus B,} \qquad \frac{\Delta \vdash t : B}{\Delta \vdash \mathtt{inj}_r\ t : A \oplus B,}$$

$$\frac{}{\vdash \mathtt{zero} : \mathtt{Nat},} \qquad \frac{\Delta \vdash t : \mathtt{Nat}}{\Delta \vdash \mathtt{S}\ t : \mathtt{Nat},}$$

$$\frac{\Delta \vdash t_i : A \quad \sum_i |\alpha_i|^2 = 1 \quad \forall i \neq j, t_i \perp t_j}{\Delta \vdash \Sigma_i(\alpha_i \cdot t_i) : A,} \qquad \frac{\vdash_\omega \omega : A \leftrightarrow B \quad \Delta \vdash t : A}{\Delta \vdash \omega\ t : B.}$$

Figure 3.4 – Typing rules of terms.

The application of a unitary to a term is what carries the computational power of the language. We have seen that in the λ-calculus, the β-reduction reduces an application to a

term where a substitution is performed. A similar mechanism is at play in this syntax; however, we have seen that the left-hand side in a unitary abstraction can contain several variables. Hence the introduction of *valuations*, which we use to perform substitution and to define our equivalent of β-reduction.

3.2.3 Valuations and Substitution

We recall the formalisation proposed in [SVV18], with the notion of valuation: a partial map from a finite set of variables (the support) to a set of values. Given two basis values b and b', we build the smallest valuation σ such that the patterns of b and b' match and such that the application of the substitution to b, written $\sigma(b)$, is equal to b'. We denote the matching of a basis value b' against a pattern b and its associated valuation σ as the predicate $\mathrm{match}(\sigma, b, b')$. Thus, $\mathrm{match}(\sigma, b, b')$ means that b' matches with b and gives a smallest valuation σ, while $\sigma(b)$ is the substitution performed. The predicate $\mathrm{match}(\sigma, b, b')$ is defined as follows, with $\mathtt{inj}_i\ b$ being either $\mathtt{inj}_l\ b$ or $\mathtt{inj}_r\ b$.

$$\frac{}{\mathrm{match}(\sigma, *, *)} \quad \frac{\sigma = \{x \mapsto b'\}}{\mathrm{match}(\sigma, x, b')} \quad \frac{\mathrm{match}(\sigma, b, b')}{\mathrm{match}(\sigma, \mathtt{inj}_i\ b, \mathtt{inj}_i\ b')}$$

$$\frac{\mathrm{match}(\sigma, b_1, b_1') \quad \mathrm{match}(\sigma, b_2, b_2') \quad \mathsf{supp}(\sigma_1) \cap \mathsf{supp}(\sigma_2) = \emptyset \quad \sigma = \sigma_1 \cup \sigma_2}{\mathrm{match}(\sigma, b_1 \otimes b_2, b_1' \otimes b_2')}$$

$$\frac{}{\mathrm{match}(\sigma, \mathtt{zero}, \mathtt{zero})} \quad \frac{\mathrm{match}(\sigma, b, b')}{\mathrm{match}(\sigma, \mathtt{S}\ b, \mathtt{S}\ b')}$$

Besides basis values, we authorise valuations to replace variables with any expression, *e.g.* $\{x \mapsto e\}$. Whenever σ is a valuation whose support contains the variables of t, we write $\sigma(t)$ for the value where the variables of t have been replaced with the corresponding terms in σ, as follows:

- $\sigma(x) = e$ if $\{x \mapsto e\} \subseteq \sigma$,
- $\sigma(*) = *$,
- $\sigma(\mathtt{inj}_l\ t) = \mathtt{inj}_l\ \sigma(t)$,
- $\sigma(\mathtt{inj}_r\ t) = \mathtt{inj}_r\ \sigma(t)$,
- $\sigma(t_1 \otimes t_2) = \sigma(t_1) \otimes \sigma(t_2)$,
- $\sigma(\mathtt{S}\ t) = \mathtt{S}\ \sigma(t)$,
- $\sigma(\Sigma_i(\alpha_i \cdot t_i)) = \Sigma_i(\alpha_i \cdot \sigma(t_i))$,
- $\sigma(\omega\ t) = \omega\ \sigma(t)$.

Remark 3.19. If $\mathrm{match}(\sigma, b, b')$, then $\sigma(b) = b'$.

Example 3.20. Given a valuation σ such that $\{x \mapsto \mathtt{inj}_l\ \mathtt{inj}_r\ *\} \subseteq \sigma$, then $\sigma(x)$ is the expression $\mathtt{inj}_l\ \mathtt{inj}_r\ *$.

Example 3.21. Given a valuation σ such that $\{x \mapsto \mathtt{inj}_l\ *, y \mapsto \mathtt{inj}_r\ *\} \subseteq \sigma$, then $\sigma(x \otimes y)$ is the expression $(\mathtt{inj}_l\ *) \otimes (\mathtt{inj}_r\ *)$.

We can now show the soundness of orthogonality with regard to pattern-matching: in other words, orthogonality is stable by substitution, and thus the previous remark ensures there cannot be any match between two basis values if they are orthogonal.

Lemma 3.22. *Given two terms t_1 and t_2, if $t_1 \perp t_2$, then for all valuations σ_1 and σ_2, $\sigma_1(t_1) \perp \sigma_2(t_2)$.*

Proof. Observe that $\sigma_1(\mathtt{inj}_l\, t_1) = \mathtt{inj}_l\, \sigma_1(t_1)$ and $\sigma_2(\mathtt{inj}_r\, t_2) = \mathtt{inj}_r\, \sigma_2(t_2)$ and thus, whatever the valuations are, those two terms are orthogonal. The rest of the proof falls directly by induction on the definition of $\perp$. □

If one of the basis values is closed, we observe that there is an equivalence between matching the pattern and not being orthogonal; which also implies that different patterns are orthogonal.

Proposition 3.23. *Given two well-typed basis values $\Delta \vdash b \colon A$ and $\vdash b' \colon A$, $\neg(b \perp b')$ iff there exists σ such that* $\mathrm{match}(\sigma, b, b')$.

Proof. This is proven by a direct induction on $\Delta \vdash b \colon A$. □

The next two lemmas provide a strong link between orthogonal decompositions and substitutions.

Lemma 3.24 (Exhaustivity and non-overlapping)**.** *Assume that $\mathrm{OD}_A S$; then for all closed basis values $\vdash b' \colon A$, there exists a unique $b \in S$ and a unique σ such that* $\mathrm{match}(\sigma, b, b')$.

Proof. This is proven by induction on the derivation of $\mathrm{OD}_A S$.

- If $\mathrm{OD}_A\{x\}$. There is only x in S and $\{x \mapsto b'\}$ is the only possible substitution.
- If $\mathrm{OD}_{\mathtt{I}}\{*\}$, we have $b' = *$ and there is nothing to do.
- If $\mathrm{OD}_{A \oplus B} S \uplus T$, there are two cases:
  - either $b' = \mathtt{inj}_l\, b'_A$, in which case the induction hypothesis gives a unique $b_A \in S$ and a unique σ such that $\mathrm{match}(\sigma, b_A, b'_A)$, and thus $\mathrm{match}(\sigma, \mathtt{inj}_l\, b_A, b)$ in a unique way,
  - or $b' = \mathtt{inj}_r\, b'_B$, and a similar argument gives a unique match $\mathrm{match}(\sigma, \mathtt{inj}_r\, b_B, b')$.
- If $\mathrm{OD}_{A \otimes B} S$, $b' = b'_A \otimes b'_B$, in both cases to derive OD, we get unique b_A, b_B, σ_A and σ_B such that $\mathrm{match}(\sigma_A \cup \sigma_B, b_A \otimes b_B, b')$.
- If $\mathrm{OD}_{\mathtt{Nat}} S^{\oplus 0}$, there are two cases: either $b' = \mathtt{zero}$, in which case there is nothing to do, or $b' = \mathtt{S}\, b''$, and the induction hypothesis gives a unique b and a unique σ such that $\mathrm{match}(\sigma, b, b'')$ and thus $\mathrm{match}(\sigma, \mathtt{S}\, b, b')$.

□

Observe that some of the results in this section focus on *basis values*, and threfore do not involve linear combinations. This is because unitary abstractions are formed as a set of clauses such as $b_i \leftrightarrow e_i$, where the terms on the left can only be basis values, and this allows us to narrow down the pattern-matching to basis values only.

The definition of valuation σ does not involve any condition on types or type judgements. However, we need this sort of condition to formulate a substitution lemma, hence the next definition.

Definition 3.25. A valuation σ is said to be well-formed with regard to a context Δ if for all $(x_i \colon A_i) \in \Delta$, we have $\{x_i \mapsto e_i\} \subseteq \sigma$ and $\vdash e_i \colon A_i$. We write $\Delta \Vdash \sigma$ for a well-formed valuation with regard to Δ.

Remark 3.26. The valuation σ obtained in Lemma 3.24 is well-formed iff b is well-typed.

Lemma 3.27. *Given a well-typed term $\Delta \vdash t : A$ and a well-formed valuation $\Delta \Vdash \sigma$, then we have $\vdash \sigma(t) : A$.*

Proof. The proof is done by induction on $\Delta \vdash t : A$.

- $\vdash * : \mathtt{I}$. Direct.
- $x : A \vdash x : A$. Since $x : A \Vdash \sigma$, there is a well-typed term t such that $\{x \mapsto e\} \subseteq \sigma$ and thus $\sigma(x)$ is well-formed.
- $\Delta_1, \Delta_2 \vdash t_1 \otimes t_2 : A \otimes B$. By induction hypothesis, $\sigma(t_1)$ and $\sigma(t_2)$ are well-formed, and thus $\sigma(t_1 \otimes t_2) = \sigma(t_1) \otimes \sigma(t_2)$ is also well-typed.
- $\Delta \vdash \mathtt{inj}_i\ t : A_1 \oplus A_2$. By induction hypothesis, $\sigma(t)$ is well-typed, and thus $\mathtt{inj}_i\ \sigma(t) = \sigma(\mathtt{inj}_i\ t)$ is also well-typed.
- $\vdash \mathtt{zero} : \mathtt{Nat}$. Direct.
- $\Delta \vdash \mathtt{S}\ t : \mathtt{Nat}$. By induction hypothesis, $\sigma(t)$ is well-typed, and thus $\mathtt{S}\ \sigma(t) = \sigma(\mathtt{S}\ t)$ is also well-typed.
- $\Delta \vdash \Sigma_i(\alpha_i \cdot t_i) : A$. By induction hypothesis, $\sigma(t_i)$ is well-typed for all i, and thus $\Sigma_i(\alpha_i \cdot \sigma(t_i)) = \sigma(\Sigma_i(\alpha_i \cdot t_i))$ is also well-typed, thanks to Lemma 3.22.
- $\Delta \vdash \omega\ t : B$. By induction hypothesis, $\sigma(t)$ is well-typed, and thus $\omega\ \sigma(t) = \sigma(\omega\ t)$ is also well-typed.

□

Substitutions, now properly defined, help us formalise how the language handles operations. In the paper this work is based on [SVV18], the computational behaviour of the language is presented through an operational semantics, while terms are considered up to linear algebraic equalities. In this chapter, we choose to work entirely with an *equational theory*, similar to the ones introduced in the previous chapters (see §1.1.1, §1.5.2 and §2.5.2), whose details are outlined in the next section. This equational theory contains both linear algebraic considerations and the computational aspects of the language.

3.3 Equational Theory

In this section, we define an equational theory for our language, akin to the ones presented in the previous chapters for the λ-calculus (see Figure 1.2), for Moggi's metalanguage (see Figure 1.5) and for the Central Submonad Calculus (see Figure 2.4). An equation judgement is written $\Delta \vdash t_1 = t_2 : A$, where Δ is a context, A is a type and t_1 and t_2 are terms. We do not need to assume that t_1 and t_2 are well-typed, it is derived (see Proposition 3.30). We provide the main equational theory of our language in Figure 3.8, with the rules of reflexivity, symmetry and transitivity in Figure 3.5, linear algebraic identities in Figure 3.6, and congruence identities in Figure 3.7.

Remark 3.28. Among the equational rules presented in this section, only the equations in Figure 3.8 provide an operational account of the language. They can be seen as a reduction system from left to right. On the other hand, the equations in Figure 3.6 show that the algebraic power of Hilbert spaces and isometries can be captured within the type system.

$$\frac{\Delta \vdash t : A}{\Delta \vdash t = t : A}\ (refl) \qquad \frac{\Delta \vdash t_1 = t_2 : A}{\Delta \vdash t_2 = t_1 : A}\ (symm)$$

$$\frac{\Delta \vdash t_1 = t_2 : A \quad \Delta \vdash t_2 = t_3 : A}{\Delta \vdash t_1 = t_3 : A}\ (trans)$$

Figure 3.5 – Basic equational rules.

$$\frac{\ell : \{1,\dots,n\} \to \{1,\dots,n\} \text{ is a bijection} \quad \Delta \vdash \Sigma_i(\alpha_i \cdot t_i) : A}{\Delta \vdash \Sigma_i(\alpha_i \cdot t_i) = \Sigma_i(\alpha_{\ell(i)} \cdot t_{\ell(i)}) : A}\ (perm)$$

$$\frac{\Delta \vdash \Sigma_{i=1}^{n}(\alpha_i \cdot t_i) : A \quad \alpha_n = 0}{\Delta \vdash \Sigma_{i=1}^{n}(\alpha_i \cdot t_i) = \Sigma_{i=1}^{n-1}(\alpha_i \cdot t_i) : A}\ (0.scal) \qquad \frac{\Delta \vdash t : A}{\Delta \vdash \Sigma_{i=1}^{1}(1 \cdot t) = t : A}\ (1.scal)$$

$$\frac{\Delta \vdash \Sigma_i\left(\alpha_i \cdot \Sigma_j(\beta_{ij} \cdot t_j)\right) : A}{\Delta \vdash \Sigma_i\left(\alpha_i \cdot \Sigma_j(\beta_{ij} \cdot t_j)\right) = \Sigma_j((\sum_i \alpha_i\beta_{ij}) \cdot t_j) : A}\ (fubini)$$

$$\frac{\Delta \vdash \Sigma_{ij}(\alpha_i\beta_{ij} \cdot t_{ij}) : A}{\Delta \vdash \Sigma_i\left(\alpha_i \cdot \Sigma_j(\beta_{ij} \cdot t_{ij})\right) = \Sigma_{ij}(\alpha_i\beta_{ij} \cdot t_{ij}) : A}\ (double)$$

$$\frac{\vdash_\omega \omega : A \leftrightarrow B \quad \Delta \vdash \Sigma_i(\alpha_i \cdot t_i) : A}{\Delta \vdash \omega\ \Sigma_i(\alpha_i \cdot t_i) = \Sigma_i(\alpha_i \cdot \omega\ t_i) : B}\ (\omega.linear)$$

$$\frac{\Delta \vdash \Sigma_i(\alpha_i \cdot t_i) : A}{\Delta \vdash \mathtt{inj}_l\ \Sigma_i(\alpha_i \cdot t_i) = \Sigma_i(\alpha_i \cdot \mathtt{inj}_l\ t_i) : A \oplus B}\ (\iota.linear_1)$$

$$\frac{\Delta \vdash \Sigma_i(\alpha_i \cdot t_i) : B}{\Delta \vdash \mathtt{inj}_r\ \Sigma_i(\alpha_i \cdot t_i) = \Sigma_i(\alpha_i \cdot \mathtt{inj}_r\ t_i) : A \oplus B}\ (\iota.linear_2)$$

$$\frac{\Delta_1 \vdash t : A \quad \Delta_2 \vdash \Sigma_i(\alpha_i \cdot t_i) : B}{\Delta_1, \Delta_2 \vdash t \otimes (\Sigma_i(\alpha_i \cdot t_i)) = \Sigma_i(\alpha_i \cdot t \otimes t_i) : A \otimes B}\ (\otimes.linear_1)$$

$$\frac{\Delta_1 \vdash t : B \quad \Delta \vdash \Sigma_i(\alpha_i \cdot t_i) : A}{\Delta_1, \Delta_2 \vdash (\Sigma_i(\alpha_i \cdot t_i)) \otimes t = \Sigma_i(\alpha_i \cdot t_i \otimes t) : A \otimes B}\ (\otimes.linear_2)$$

$$\frac{\Delta \vdash \Sigma_i(\alpha_i \cdot t_i) : \mathtt{Nat}}{\Delta \vdash \mathtt{S}\ \Sigma_i(\alpha_i \cdot t_i) = \Sigma_i(\alpha_i \cdot \mathtt{S}\ t_i) : \mathtt{Nat}}\ (S.linear)$$

Figure 3.6 – Vector space and linear applications equational rules.

3.3.1 Equations and typing

We start by proving that the equational theory presented is sound with the typing rules of the language. In other words, we show that if two terms are equal in our theory, they are both well-typed. To do so, we need to show that equality between terms preserve orthogonality.

Lemma 3.29. *Given two terms t_1 and t_2 such that $t_1 \perp t_2$ and $\Delta \vdash t_1 = t_1' : A$, then $t_1' \perp t_2$.*

Proof. By induction on the rules of the equational theory. □

$$\frac{\Delta \vdash t_1 = t_2 \colon A}{\Delta \vdash \mathtt{inj}_l\ t_1 = \mathtt{inj}_l\ t_2 \colon A \oplus B}\ (\iota.eq_1) \qquad \frac{\Delta \vdash t_1 = t_2 \colon B}{\Delta \vdash \mathtt{inj}_r\ t_1 = \mathtt{inj}_r\ t_2 \colon A \oplus B}\ (\iota.eq_2)$$

$$\frac{\Delta \vdash t_1 = t_2 \colon A \quad \Delta' \vdash t \colon B}{\Delta, \Delta' \vdash t_1 \otimes t = t_2 \otimes t \colon A \otimes B}\ (\otimes.eq_1) \qquad \frac{\Delta \vdash t_1 = t_2 \colon B \quad \Delta' \vdash t \colon A}{\Delta, \Delta' \vdash t \otimes t_1 = t \otimes t_2 \colon A \otimes B}\ (\otimes.eq_2)$$

$$\frac{\Delta \vdash t_1 = t_2 \colon \mathtt{Nat}}{\Delta \vdash \mathtt{S}\ t_1 = \mathtt{S}\ t_2 \colon \mathtt{Nat}}\ (S.eq) \qquad \frac{\Delta \vdash t_1 = t_2 \colon A \quad \vdash_\omega \omega \colon A \leftrightarrow B}{\Delta \vdash \omega\ t_1 = \omega\ t_2 \colon B}\ (\omega.eq)$$

$$\frac{\Delta \vdash \Sigma_i(\alpha_i \cdot t_i) \colon A \quad \forall i, \Delta \vdash t_i = t_i' \colon A}{\Delta \vdash \Sigma_i(\alpha_i \cdot t_i) = \Sigma_i(\alpha_i \cdot t_i') \colon A}\ (\Sigma.eq)$$

Figure 3.7 – Congruence equational rules of simply-typed quantum control.

$$\frac{\vdash b' \colon A \quad \vdash_\omega \{\ |\ \ b_1 \leftrightarrow e_1 \mid \ldots \mid\ \ b_n \leftrightarrow e_n\ \} \colon A \leftrightarrow B \quad \mathrm{match}(\sigma, b_i, b')}{\vdash \{\ |\ \ b_1 \leftrightarrow e_1 \mid \ldots \mid\ \ b_n \leftrightarrow e_n\ \}\ b' = \sigma(e_i) \colon B}\ (\omega.\beta)$$

$$\frac{\vdash \omega\ b = v \colon B}{\vdash \omega^{-1}\ v = b \colon A}\ (\omega.inv) \qquad \frac{\vdash_\omega \omega_1 \colon A \leftrightarrow B \quad \vdash_\omega \omega_2 \colon B \leftrightarrow C \quad \Delta \vdash b \colon A}{\Delta \vdash (\omega_2 \circ \omega_1)\ b = \omega_2\ (\omega_1\ b) \colon C}\ (\omega.comp)$$

$$\frac{\vdash_\omega \omega_1 \colon A_1 \leftrightarrow B_1 \quad \vdash_\omega \omega_2 \colon A_2 \leftrightarrow B_2 \quad \vdash b_1 \colon A_1 \quad \vdash b_2 \colon A_2}{\vdash (\omega_1 \otimes \omega_2)\ (b_1 \otimes b_2) = (\omega_1\ b_1) \otimes (\omega_2\ b_2) \colon B_1 \otimes B_2}\ (\omega.\otimes)$$

$$\frac{\vdash_\omega \omega_1 \colon A_1 \leftrightarrow B_1 \quad \vdash_\omega \omega_2 \colon A_2 \leftrightarrow B_2 \quad \vdash b \colon A_1}{\vdash (\omega_1 \oplus \omega_2)\ (\mathtt{inj}_l\ b) = \mathtt{inj}_l\ (\omega_1\ b) \colon B_1 \oplus B_2}\ (\omega.\oplus_1)$$

$$\frac{\vdash_\omega \omega_1 \colon A_1 \leftrightarrow B_1 \quad \vdash_\omega \omega_2 \colon A_2 \leftrightarrow B_2 \quad \vdash b \colon A_2}{\vdash (\omega_1 \oplus \omega_2)\ (\mathtt{inj}_r\ b) = \mathtt{inj}_r\ (\omega_2\ b) \colon B_1 \oplus B_2}\ (\omega.\oplus_2)$$

$$\frac{\vdash_\omega \omega \colon A \leftrightarrow A \quad \vdash b \colon A}{\vdash (\mathtt{ctrl}\ \omega)\ ((\mathtt{inj}_l\ *) \otimes b) = (\mathtt{inj}_l\ *) \otimes b \colon (\mathtt{I} \oplus \mathtt{I}) \otimes A}\ (\omega.ctrl_1)$$

$$\frac{\vdash_\omega \omega \colon A \leftrightarrow A \quad \vdash b \colon A}{\vdash (\mathtt{ctrl}\ \omega)\ ((\mathtt{inj}_r\ *) \otimes b) = (\mathtt{inj}_r\ *) \otimes (\omega\ b) \colon (\mathtt{I} \oplus \mathtt{I}) \otimes A}\ (\omega.ctrl_2)$$

Figure 3.8 – Computational equational rules of simply-typed quantum control.

The proof of the next proposition heavily relies on the previous lemma. Indeed, to prove that a linear combination is well-typed, one needs to prove that all the terms involved in the linear combination are pairwise orthogonal.

Proposition 3.30. *If $\Delta \vdash t_1 = t_2 : A$ is well-formed, then $\Delta \vdash t_1 : A$ and $\Delta \vdash t_2 : A$ also are.*

Proof. By induction on the rules of the equational theory. □

3.3.2 Bases

As expected, and thanks to the equational theory, an orthogonal decomposition gives a finite representation of an orthonormal basis. This is proven in Lemma 3.33. We start by proving that any expression is equal to a combination of basis values.

Lemma 3.31. *Given a well-typed closed expression $\vdash e : A$, there exists a set of indices I, a family of basis values $(b_i)_{i\in I}$, $(\alpha_i)_{i\in I}$ a family of complex numbers, such that $\vdash e = \Sigma_i(\alpha_i \cdot b_i) : A$.*

Proof. The proof is done by induction on $\vdash e : A$.

- $\vdash * : \mathtt{I}$. Nothing to do.
- $\vdash e_1 \otimes e_2 : A \otimes B$. The induction hypothesis gives I, (b_i^1) and (α_i), and J, (b_j^2) and (β_j), such that $\vdash e_1 = \Sigma_i(\alpha_i \cdot b_i^1) : A$ and $\vdash e_2 = \Sigma_j(\beta_j \cdot b_j^2) : B$. Thus, we have that

$$
\begin{aligned}
&\vdash e_1 \otimes e_2 \\
&= \left(\Sigma_i(\alpha_i \cdot b_i^1)\right) \otimes \left(\Sigma_j(\beta_j \cdot b_j^2)\right) : A \otimes B && \text{(induction hypothesis)} \\
&= \Sigma_i \left(\alpha_i \cdot b_i^1 \otimes \left(\Sigma_j(\beta_j \cdot b_j^2)\right)\right) : A \otimes B && (\otimes.linear_2) \\
&= \Sigma_i \left(\alpha_i \cdot \Sigma_j(\beta_j \cdot b_i^1 \otimes b_j^2)\right) : A \otimes B && (\otimes.linear_1) \\
&= \Sigma_{ij}(\alpha_i\beta_j \cdot b_i^1 \otimes b_j^2) : A \otimes B && (double)
\end{aligned}
$$

- $\vdash \mathtt{inj}_l\ e : A \oplus B$. The induction hypothesis gives $\vdash e = \Sigma_i(\alpha_i \cdot b_i) : A$, and observe that $\vdash \mathtt{inj}_l\ \Sigma_i(\alpha_i \cdot b_i) = \Sigma_i(\alpha_i \cdot \mathtt{inj}_l\ b_i) : A \oplus B$.
- $\vdash \mathtt{inj}_r\ e : A \oplus B$ has the same conclusion.
- $\vdash \mathtt{S}\ e : \mathtt{Nat}$ is similar to the previous point.
- $\vdash \Sigma_i(\alpha_i \cdot e_i) : A$. The induction hypothesis gives (β_{ij}) and (b_j) (the b does not depend in i without loss of generality, because $0 \cdot b$ can be added to any sum term t, as long as b is orthogonal to t). Finally, we have $\vdash \Sigma_i(\alpha_i \cdot \Sigma_j(\beta_{ij} \cdot b_j)) = \Sigma_j((\sum_i \alpha_i\beta_{ij}) \cdot b_j) : A$.

□

Remark 3.32. The resulting term in the previous lemma, written $\Sigma_i(\alpha_i \cdot b_i)$ is a value if the basis values are correctly ordered and if all the scalars are non zero. Thanks to the $(perm)$ and $(0.scal)$ rules in Figure 3.6, we can assume so. Thus, the lemma above shows that expressions have a unique normal form.

Lemma 3.31 shows that any expressions can be decomposed as a linear combination of elements of the *canonical* orthogonal decomposition – namely, made of basis values only. We can generalise this lemma to any orthogonal decomposition, with the help of substitutions. We show that, given an orthogonal decomposition S, a closed expression e can be written as a normalised decomposition of elements of S, where variables are substituted. The elements of S can appear several times in the decomposition. For example, $\{x\}$ is an orthogonal decomposition of $\mathtt{I} \oplus \mathtt{I}$, and the term $\frac{1}{\sqrt{2}} \cdot (\mathtt{inj}_l\ *) + \frac{1}{\sqrt{2}} (\mathtt{inj}_r\ *)$ is of the form $\frac{1}{\sqrt{2}} \cdot \sigma_1(x) + \frac{1}{\sqrt{2}} \cdot \sigma_2(x)$, where σ_1 replaces x with $\mathtt{inj}_l\ *$ and σ_2 replaces x with $\mathtt{inj}_r\ *$.

Lemma 3.33. *Given $\mathrm{OD}_B^{ext} S$, where all elements of S are well-typed, and a well-typed closed expression $\vdash e : B$, there exists I a set of indices, $(s_i)_{i \in I}$ a family of elements of S, $(\alpha_i)_{i \in I}$ a family of complex numbers and $(\sigma_i)_{i \in I}$ a family of valuations such that $\vdash e = \Sigma_i(\alpha_i \cdot \sigma_i(s_i)) : B$.*

Proof. This is proven by induction on OD. The previous lemma gives a term equal to e in the equational theory written as a finite sum of basis values $\Sigma_i(\alpha_i \cdot b_i)$.

- $\mathrm{OD}_A^{ext}(\{x\})$. The substitution $\sigma = \{x \mapsto e\}$ is suitable with $e = \sigma(x)$.
- $\mathrm{OD}_{\mathtt{I}}^{ext}(\{*\})$. Nothing to do.
- $\mathrm{OD}_{A \oplus B}^{ext} S \boxplus T$. Each b_i is either $\mathtt{inj}_l\ b_i'$, with gives a suitable substitution σ_i and $s_i \in S$, thus $\mathtt{inj}_l\ s_i \in S \boxplus T$, or b_i is $\mathtt{inj}_r\ b_i'$, giving suitable substitution σ_i and $s_i \in T$, thus $\mathtt{inj}_r\ s_i \in S \boxplus T$; all this by induction hypothesis.
- $\mathrm{OD}_{A \otimes B}^{ext} S$. Each b_i is of the form $b_i' \otimes b_i''$, the induction hypothesis gives suitable σ_i', s_i', σ_i'', s_i'', that can be assembled into $\sigma_i = \sigma_i' \cup \sigma_i''$ and $s_i = s_i' \otimes s_i''$.
- $\mathrm{OD}_{\mathtt{Nat}}^{ext} S^{\oplus 0}$. Each b_i is either $\mathtt{zero}$, for which there is nothing to do, or $\mathtt{S}\ b_i'$, in which case the induction hypothesis concludes.
- $\mathrm{OD}_B^{ext} S^{\beta}$. First, we show that each $s_i \in S$ can be written as a linear combination of elements of S^{β}. Indeed, in the equational theory:

$$\begin{aligned} \Sigma_{s \in S}(\overline{\beta}_{s,s_i} \cdot \Sigma_{s' \in S}(\beta_{s,s'} \cdot s')) &= \Sigma_{s' \in S} \left(\sum_{s \in S} \overline{\beta}_{s,s_i} \beta_{s,s'} \right) \cdot s' \\ &= \Sigma_{s' \in S}(\delta_{s'=s_i} \cdot s') = s_i \end{aligned}$$

  and the conclusion is then direct.

□

3.3.3 Normal Forms

The following lemma is loosely equivalent to progress for an operational semantics, and involves both directions: the application of a unitary to a value reduces to a value, and given a unitary and a value, there exists a value that is the inverse image of the latter.

Lemma 3.34. *Given $\vdash_\omega \omega : A \leftrightarrow B$ (see Figure 3.3), we have the following:*

- *for all $\vdash e : A$, there exists a value judgement $\vdash v : B$ such that $\vdash \omega\ e = v : B$;*
- *for all $\vdash e : B$, there exists a value judgement $\vdash u : A$ such that $\vdash \omega\ u = e : B$.*

Proof. This is proven by induction on the judgement $\vdash_\omega \omega : A \leftrightarrow B$.

- Assume $\vdash_\omega \{ \mid b_1 \leftrightarrow e_1 \mid \ldots \mid b_n \leftrightarrow e_n \} : A \leftrightarrow B$, we have in particular that $\mathrm{OD}_A(\{b_i\}_{i\leq n})$. In the case $\vdash e\colon A$, Lemma 3.33 then gives a set J and a decomposition of e as follows: $\vdash e = \Sigma_j(\alpha_j \cdot \sigma_j(b_{i_j}))\colon A$. Therefore,

$$\begin{aligned}
&\vdash \{ \mid b_1 \leftrightarrow e_1 \mid \ldots \mid b_n \leftrightarrow e_n \}\ e \\
&\quad = \{ \mid b_1 \leftrightarrow e_1 \mid \ldots \mid b_n \leftrightarrow e_n \}\ \Sigma_j(\alpha_j \cdot \sigma_j(b_{i_j}))\colon B && (\omega.eq) \\
&\quad = \Sigma_j(\alpha_j \cdot \{ \mid b_1 \leftrightarrow e_1 \mid \ldots \mid b_n \leftrightarrow e_n \}\ \sigma_j(b_{i_j}))\colon B && (\omega.linear) \\
&\quad = \Sigma_j(\alpha_j \cdot \sigma_j(\{ \mid b_1 \leftrightarrow e_1 \mid \ldots \mid b_n \leftrightarrow e_n \}\ b_{i_j}))\colon B && \text{(definition)} \\
&\quad = \Sigma_j(\alpha_j \cdot \sigma_j(v_{i_j}))\colon B && (\omega.\beta)
\end{aligned}$$

  The latter term is a closed expression, thus Lemma 3.31 ensures that there exists a value $\vdash v\colon A$ such that $\vdash \{ \mid b_1 \leftrightarrow e_1 \mid \ldots \mid b_n \leftrightarrow e_n \}\ e = v\colon B$.
  On the other hand, we have $\mathrm{OD}^{ext}_B(\{v_i\}_{i\leq n})$. Assume $\vdash e\colon B$, then Lemma 3.33 provides a set K and a decomposition $\vdash e = \Sigma_k(\alpha_k \cdot \sigma_k(v_{i_k}))\colon B$. With the same computation as above, we have $\vdash \{ \mid b_1 \leftrightarrow e_1 \mid \ldots \mid b_n \leftrightarrow e_n \}\ u' = e\colon B$ with u' being the expression $\Sigma_k(\alpha_k \cdot \sigma_k(b_{i_k}))$. Since it is an expression, Lemma 3.31 ensures that there is value $\vdash u\colon A$ such that $\vdash u' = u\colon A$ and therefore $\vdash \{ \mid b_1 \leftrightarrow e_1 \mid \ldots \mid b_n \leftrightarrow e_n \}\ u = e\colon B$.
- Assume $\vdash_\omega \omega^{-1}\colon B \leftrightarrow A$. Given $\vdash e\colon B$, the induction hypothesis gives $\vdash u\colon A$ such that $\vdash \omega\ u = e\colon B$, and thus $\vdash \omega^{-1}\ e = u\colon A$. Moreover, given $\vdash e\colon A$, the induction hypothesis gives $\vdash v\colon B$ such that $\vdash \omega\ e = v\colon B$, and thus $\vdash \omega^{-1}\ v = e\colon A$.
- Assume $\vdash_\omega \omega_2 \circ \omega_1\colon A \leftrightarrow C$. The induction hypothesis gives us v_1 such that $\vdash \omega_1\ e = v_1\colon B$ and then v_2 such that $\vdash \omega_2\ v_1 = v_2\colon C$, which ensures the result. A related reasoning proves the second point.
- Assume $\vdash_\omega \omega_1 \oplus \omega_2\colon A_1 \oplus A_2 \leftrightarrow B_1 \oplus B_2$. Lemma 3.31 ensures that e is given as a combination of basis values $\vdash e = \Sigma_i(\alpha_i \cdot b_i)\colon A_1 \oplus A_2$. Moreover, we know that $\vdash (\omega_1 \oplus \omega_2)\ e = \Sigma_i(\alpha_i \cdot (\omega_1 \oplus \omega_2)\ b_i)\colon B_1 \oplus B_2$. Therefore, it is sufficient to consider the case of basis values. There are two similar cases, namely $\mathtt{inj}_l\ b$ and $\mathtt{inj}_r\ b$. In the first case, the induction hypothesis gives v_1 such that $\vdash \omega_1\ b = v_1\colon B_1$, thus $\vdash (\omega_1 \oplus \omega_2)\ (\mathtt{inj}_l\ b) = \mathtt{inj}_l\ v_1\colon B_1 \oplus B_2$. The other case is similar. A related reasoning proves the second point.
- Assume $\vdash_\omega \omega_1 \otimes \omega_2\colon A_1 \otimes A_2 \leftrightarrow B_1 \otimes B_2$. Like above, it is sufficient to prove the result for basis values. We write $b_1 \otimes b_2$ for b, and the induction hypothesis provides v_1 and v_2 such that $\vdash \omega_1\ b_1 = v_1\colon B_1$ and $\vdash \omega_2\ b_2 = v_2\colon B_2$, ensuring that $\vdash (\omega_1 \otimes \omega_2)\ (b_1 \otimes b_2) = v_1 \otimes v_2\colon B_1 \otimes B_2$. A related reasoning proves the second point.
- Assume $\vdash_\omega \mathtt{ctrl}\ \omega\colon (\mathtt{I} \oplus \mathtt{I}) \otimes A \leftrightarrow (\mathtt{I} \oplus \mathtt{I}) \otimes A$. Once again, it is sufficient to prove the result for basis values. In the case $(\mathtt{inj}_l\ *) \otimes b$, there is nothing to do. The other case is $(\mathtt{inj}_r\ *) \otimes b$, and the induction hypothesis gives v such that $\vdash \omega\ b = v\colon A$, and then $\vdash (\mathtt{ctrl}\ \omega)\ ((\mathtt{inj}_r\ *) \otimes b) = (\mathtt{inj}_r\ *) \otimes v\colon (\mathtt{I} \oplus \mathtt{I}) \otimes A$. A related reasoning proves the second point.

□

We have proven that unitary applications progress and reduce to values, if one wishes to have an operational point of view. This allows us to prove, with the same operational view, that the system admits unique normal forms; this means that any term t is equal to a single value.

Theorem 3.35. *Given* $\vdash t : A$*, there exists* $\vdash v : A$ *such that* $\vdash t = v : A$.

Proof. This is proven by induction on the typing rules of $\vdash t : A$.

- The cases $*$, x, $\mathtt{zero}$ and sum are straightforward.
- In the case $t_1 \otimes t_2$, the induction hypothesis gives corresponding v_1 and v_2, that ensure $\vdash t_1 \otimes t_2 = v_1 \otimes v_2 : A \otimes B$.
- In the case $\mathtt{inj}_i\ t$, the induction hypothesis provides v such that $\vdash t = v : A_i$, thus $\vdash \mathtt{inj}_i\ t = \mathtt{inj}_i\ v : A_1 \oplus A_2$.
- In the case $\mathtt{S}\ t$, the induction hypothesis provides v such that $\vdash t = v : \mathtt{Nat}$, thus $\vdash \mathtt{S}\ t = \mathtt{S}\ v : \mathtt{Nat}$.
- In the case $\omega\ t$, the induction hypothesis gives v such that $\vdash t = v : A$. The previous lemma, Lemma 3.34, provides v' such that $\vdash \omega\ v = v' : B$, thus $\vdash t = v' : B$.

□

3.3.4 Discussion: Operational Semantics

We briefly discuss the operational semantics for terms presented in [SVV18]. This section builds up the comparison with the λ-calculus by defining our version of β-reduction that suits the language. This reduction is given by the rule $(\omega.\beta)$ in Figure 3.8, when read left to right.

In [SVV18], values and terms are considered modulo associativity and commutativity of the addition, and modulo the equational theory of modules; and they consider the value and term constructs $- \otimes -$, $\mathtt{inj}_l\ (-)$, $\mathtt{inj}_r\ (-)$, $\mathtt{S}\ -$ and $\omega\ -$ distributive over sum and scalar multiplication, only in this subsection also.

Therefore, in that setting, an expression e is equal to some combination of basis values $\Sigma_i(\alpha_i \cdot b_i)$; and the application of a unitary ω to e is equal to $\Sigma_i(\alpha_i \cdot \omega\ b_i)$ thanks to linearity. Thus, it is sufficient to give a β-reduction rule for unitaries applied to basis values, as follows.

$$\frac{\mathrm{match}(\sigma, b_i, b')}{\{\ |\ b_1 \leftrightarrow e_1\ |\ \ldots\ |\ b_n \leftrightarrow e_n\ \}\ b' \to \sigma(v_i)}$$

This rule is the same as $(\omega.\beta)$, this time oriented left to right. Note that the reduction defined this way can only be applied with a closed b'. However, this formulation is not satisfying, because it requires working up to linear algebra equalities, which are then mixed with an operational semantics. Our solution in this chapter is to completely embrace the equational theory aspect and only work up to equalities, keeping in mind which rules bear a computational meaning such as the one above.

Another solution is to work only with rewriting rules, which has been the focus of several papers around algebraic λ-calculi [ADCV17, AD17, Vau09, SV09], where all the rules, whether they are computational or linear algebraic, have a direction.

3.4 Mathematical Development: Hilbert spaces for semantics

This section heavily relies on the notations and definitions in §1.4, where introductory notions on Hilbert spaces are outlined. The goal of this section is to provide the tools to define

the denotational semantics of the programming language given above. To do so, we work with contractions for convenience, since the main mathematical objects that we need – namely isometries and unitaries – are in particular contractions.

Sums. Given two maps $f, g \colon X \to Y$ in $\mathbf{Contr}$, their linear algebraic sum $f + g$ is not necessarily a contraction. We introduce the notion of *compatibility*. This notion is inherited from the one in restriction and inverse categories (see Definition 1.59). We use, in particular, an observation that links zero morphisms to compatibility in inverse categories (see Lemma 1.70); but this is adapted to Hilbert spaces in this chapter. In this context, the *join* is also different, as we use the algebraic sum inherited from the vector space structure.

Definition 3.36 (Compatibility)**.** Given $f, g \colon X \to Y$ two maps in $\mathbf{Contr}$, f and g are said to be compatible if $(\mathrm{Ker} f)^\perp \perp (\mathrm{Ker} g)^\perp$ and $\mathrm{Im} f \perp \mathrm{Im} g$.

This definition of compatibility ensures that there is no overlap between the inputs, and also between the outputs. The next lemma is then direct.

Lemma 3.37. *Given two compatible contractive maps $f, g \colon X \to Y$, $f + g$ is also contractive.*

Proof. Let $x \in X$. Given the assumptions, there exists $x' \in (\mathrm{Ker} f)^\perp$ and $x'' \in (\mathrm{Ker} g)^\perp$ such that $(f+g)(x) = f(x') + g(x'')$, $g(x') = 0$, $f(x'') = 0$ and $\langle f(x') \mid g(x'') \rangle$. Therefore,

$$\begin{aligned} \|(f+g)(x)\| &= \|f(x') + g(x'')\| = \langle f(x') + g(x'') \mid f(x') + g(x'') \rangle \\ &= \langle f(x') \mid f(x') \rangle + \langle g(x'') \mid g(x'') \rangle \leq \langle x' \mid x' \rangle + \langle x'' \mid x'' \rangle \\ &\leq \langle x \mid x \rangle = \|x\|. \end{aligned}$$

□

As mentioned above, the conditions in Def. 3.36 can be simplified in more algebraic expressions, in the spirit of Lemma 1.70. We prove a quick lemma first.

Lemma 3.38. *Given $f \colon X \to Y$ in $\mathbf{Contr}$, we have:*
- $\mathrm{Ker}(f^\dagger) = (\mathrm{Im} f)^\perp$;
- $\mathrm{Im}(f^\dagger)^\perp = \mathrm{Ker} f$.

Proof. Let us prove both points separately.
- We proceed by double inclusion.
  - Let $x \in \mathrm{Ker}(f^\dagger)$. Let $y \in X$. We have $\langle x \mid fy \rangle = \langle f^\dagger x \mid y \rangle = \langle 0 \mid y \rangle = 0$. Therefore, $\mathrm{Ker}(f^\dagger) \subseteq (\mathrm{Im} f)^\perp$.
  - Let $x \in (\mathrm{Im} f)^\perp$. Thus, for all $y \in X$, we have $\langle x \mid fy \rangle = 0$, which implies that $\langle f^\dagger x \mid y \rangle = 0$. Since it is true for all y, we have $f^\dagger x = 0$. Therefore, $(\mathrm{Im} f)^\perp \subseteq \mathrm{Ker}(f^\dagger)$.
- We proceed by double inclusion.
  - Let $x \in \mathrm{Im}(f^\dagger)^\perp$. Thus, for all $y \in Y$, $\langle x \mid f^\dagger y \rangle = 0$; then $\langle fx \mid y \rangle =$ for all y, thus $fx = 0$. Therefore, $\mathrm{Im}(f^\dagger)^\perp \subseteq \mathrm{Ker} f$.
  - Let $x \in \mathrm{Ker} f$. Let $y \in Y$. We have $\langle x \mid f^\dagger y \rangle = \langle fx \mid y \rangle = 0$. Therefore, $\mathrm{Ker} f \subseteq \mathrm{Im}(f^\dagger)^\perp$.

□

We can now express a sufficient condition for compatibility in algebraic terms.

Lemma 3.39. *Given two contractive maps $f, g\colon X \to Y$, $f^\dagger g = 0$ and $fg^\dagger = 0$ iff f and g are compatible.*

Proof. We prove this lemma by double implication.

- If f and g are compatible, then for all $x \in X$, gx is in $\mathrm{Im}g \subseteq \mathrm{Im}f^\perp = \mathrm{Ker}(f^\dagger)$, thus $f^\dagger g = 0$. Similarly, for all $y \in Y$, $g^\dagger y$ is in $\mathrm{Im}(g^\dagger) \subseteq (\mathrm{Im}(f^\dagger))^\perp = \mathrm{Ker}f$.
- If $f^\dagger g = 0$ and $fg^\dagger = 0$. The first equality implies that $\mathrm{Im}g \subseteq \mathrm{Ker}(f^\dagger) = (\mathrm{Im}f)^\perp$ and therefore $\mathrm{Im}g \perp \mathrm{Im}f$. The second equality similarly implies that $(\mathrm{Ker}f)^\perp \perp (\mathrm{Ker}g)^\perp$.

□

Remark 3.40. It might seem that the conditions introduced above are not symmetric on f and g. But one can observe that $0^\dagger = 0$ and $(f^\dagger g)^\dagger = g^\dagger f^{\dagger\dagger} = g^\dagger f$, thus $f^\dagger g = 0$ iff $g^\dagger f = 0$. Similarly, $fg^\dagger = 0$ iff $gf^\dagger = 0$.

Lemma 3.39 introduces a new point of view on compatibility, through an *orthogonality* between morphisms, as it was observed for inverse categories in Remark 1.77. This new point of view of orthogonality is a generalisation of the orthogonality in Hilbert spaces. Indeed, two vectors $|x\rangle$ and $|y\rangle$ in a Hilbert space H are orthogonal if $\langle x|y\rangle = 0$. In our generalised view, $|x\rangle$ and $|y\rangle$ are orthogonal if $|x\rangle^\dagger |y\rangle = 0$. Since $|x\rangle^\dagger = \langle x|$, our orthogonality between morphisms generalises the usual notion of orthogonality.

Example 3.41. The morphisms $|0\rangle\langle 0| : \mathbb{C}^2 \to \mathbb{C}^2$ and $|1\rangle\langle 1| : \mathbb{C}^2 \to \mathbb{C}^2$ are orthogonal in our generalised sense, because the vectors $|0\rangle$ and $|1\rangle$ are orthogonal in the linear algebraic sense. This justifies that their linear sum $|0\rangle\langle 0| + |1\rangle\langle 1|$ is a contraction (and, in this case, it is also a unitary).

Direct sum. Unsurprisingly, the unit type is to be represented by the one-dimensional Hilbert space $\mathbb{C}$, the line of complex numbers. In the syntax, orthogonality and thus pattern-matching, depend on direct sums. The latter are interpreted as direct sums of Hilbert spaces. We show that this interpretation gives rise to orthogonality in the sense of contractions.

Definition 3.42. We write $\iota_l^{X,Y} : X \to X \oplus Y$ for the isometry such that for all $x \in X$, $\iota_l^{X,Y} x = (x, 0)$. We call this the *left injection*. Similarly, the *right injection* is written $\iota_r^{X,Y} : Y \to X \oplus Y$.

Lemma 3.43 ([HV19])**.** *Given two Hilbert spaces X, Y, $(\iota_l^{X,Y})^\dagger \iota_r^{X,Y} = 0$ and $(\iota_r^{X,Y})^\dagger \iota_l^{X,Y} = 0$.*

Example 3.44. The previous lemma ensures that $\iota_l^{X,Y}(\iota_l^{X,Y})^\dagger$ and $\iota_r^{X,Y}(\iota_r^{X,Y})^\dagger$ are compatible. Note that $\iota_l^{X,Y}(\iota_l^{X,Y})^\dagger + \iota_r^{X,Y}(\iota_r^{X,Y})^\dagger = \mathrm{id}$.

Note that given a complex number α and a contraction $f\colon A \to B$, the outer product $\alpha \cdot f$ is written αf when it is not ambiguous. Given a set S, we write $(\alpha_i)_{i \in S}$ for a family of complex numbers indexed by S. Given two sets S and S', we write $(\alpha_{i,j})_{(i,j) \in S \times S'}$ for a matrix of complex numbers indexed by S and S'. The sets of indices can be omitted if there is no ambiguity, as in Lemma 3.45.

Lemma 3.45. *Given a family of pairwise output compatible isometries $f_i \colon A \to B$, and a family of complex numbers α_i such that $\sum_i |\alpha_i|^2 = 1$, $\sum_i \alpha_i f_i$ is an isometry.*

Proof.

$$\begin{aligned}
&(\sum_i \alpha_i f_i)^\dagger \circ (\sum_j \alpha_j f_j) \\
&= (\sum_i \overline{\alpha}_i f_i^\dagger) \circ (\sum_j \alpha_j f_j) && \text{(dagger and sum commute)} \\
&= \sum_{i,j} (\overline{\alpha}_i \alpha_j) f_i^\dagger \circ f_j && \text{(composition and sum commute)} \\
&= \sum_i (\overline{\alpha}_i \alpha_i) f_i^\dagger \circ f_i && \text{(pairwise compatibility)} \\
&= \left(\sum_i \overline{\alpha}_i \alpha_i \right) \mathrm{id} && \text{(isometry)} \\
&= \left(\sum_i |\alpha_i|^2 \right) \mathrm{id} = \mathrm{id}.
\end{aligned}$$

□

We recall that given two maps $f \colon A \to C$ and $g \colon B \to C$ in $\mathbf{Contr}$, if $f^\dagger g = 0_{C,A}$, we say that f and g are orthogonal. We show that this orthogonality is preserved by postcomposing with an isometry.

Lemma 3.46. *Given two orthogonal maps $f \colon A \to C$ and $g \colon B \to C$ in $\mathbf{Contr}$, and given an isometry $h \colon C \to D$, then $h \circ f \colon A \to D$ and $h \circ g \colon B \to D$ are also orthogonal.*

Proof.

$$\begin{aligned}
&(h \circ f)^\dagger \circ h \circ g \\
&= f^\dagger \circ h^\dagger \circ h \circ g && \text{(dagger is contravariant)} \\
&= f^\dagger \circ \mathrm{id}_C \circ g = f^\dagger \circ g && \text{(isometry)} \\
&= 0_{C,A}. && \text{(hypothesis)}
\end{aligned}$$

□

In a similar vein, the postcomposition of an isometry with an isometry is still an isometry. This was already observed when we mentionned that Hilbert spaces and isometries form a category.

The canonical countably-dimensional Hilbert space is $\ell^2(\mathbb{N})$, defined in §1.4. We recall that we write $|n\rangle$ for the elements of the canonical basis in $\ell^2(\mathbb{N})$. This is an abuse of notation, since the symbols $|0\rangle$ and $|1\rangle$ are already used for the canonical basis of $\mathbb{C}^2$. This is not an issue, since there is an isometric embedding $\mathbb{C}^2 \to \ell^2(\mathbb{N})$ which maps $|0\rangle$ to $|0\rangle$ and $|1\rangle$ to $|1\rangle$.

Definition 3.47. We write $\mathrm{succ} \colon \ell^2(\mathbb{N}) \to \ell^2(\mathbb{N})$ for the linear map $\ell^2(\mathbb{N}) \to \ell^2(\mathbb{N})$ which maps $|n\rangle$ to $|n+1\rangle$.

Remark 3.48. Note that succ can also be seen as the image of the successor function in the natural numbers by the functor ℓ^2. The linear map succ is an isometry.

Example 3.49.

$$\mathrm{succ}\,|7\rangle = |8\rangle \qquad \mathrm{succ}\left(\frac{\sqrt{3}}{2}\,|9\rangle + \frac{1}{2}\,|11\rangle\right) = \frac{\sqrt{3}}{2}\,|10\rangle + \frac{1}{2}\,|12\rangle$$

Unitaries. The denotational semantics of our programming language involves unitary maps to interpret the functions. Those maps live in the category $\mathbf{Uni}$, which is a rig category: it has bifunctors $\oplus$ and $\otimes$ inherited from $\mathbf{Hilb}$. Hence the next lemma.

Lemma 3.50. *Given two maps $f\colon A \to B$ and $g\colon C \to D$ in $\mathbf{Uni}$, $f \otimes g\colon A \otimes C \to B \otimes D$ is a map in $\mathbf{Uni}$ and $f \oplus g\colon A \oplus C \to B \oplus D$ is a map in $\mathbf{Uni}$.*

Proof. Direct since the functors $\oplus$ and $\otimes$ are †-functors. □

Finally, we present an operation that is common to quantum computing, and thus preserves the unitary structure.

Lemma 3.51 (Controlled unitary)**.** *Given a unitary map $f\colon A \to A$, there is a unitary map $\mathrm{ctrl}_A(f)\colon (\mathbb{C} \oplus \mathbb{C}) \otimes A \to (\mathbb{C} \oplus \mathbb{C}) \otimes A$ such that $\mathrm{ctrl}_A(f) = |0\rangle\langle 0| \otimes \mathrm{id} + |1\rangle\langle 1| \otimes f$.*

Proof. Direct. □

3.5 Denotational Semantics

As usual, we write $[\![-]\!]$ for the interpretation of types and term judgements. As mentioned in the previous section, the presentation makes extensive use of contractions for the denotational semantics. However, values and terms are directly announced to be isometries, for clarity. It will also help us highlight the fact that values and terms represent sound quantum states. In the same vein, the interpretation of unitaries is given as unitary maps between two Hilbert spaces; but the proof that the semantics of a unitary abstraction is unitary requires the mathematical development at the level of contractions.

3.5.1 Detailed presentation of the Semantics

Types. The interpretation of a type A is given by a countably-dimensional Hilbert space. It is given by induction on the grammar of the types. This interpretation is detailed in Figure 3.9.

Expressions. We start with *expressions*, whose typing rules are introduced in Figure 3.2. Expressions are terms without unitary application. The semantics of general terms in given below, once the semantics of unitaries is defined. Judgements for expressions are first interpreted as contractions between Hilbert spaces, and we then show that they are isometries. A judgement is of the form $\Delta \vdash e\colon A$, and its interpretation is written $[\![\Delta \vdash e\colon A]\!]$. Contexts $\Delta = x_1\colon A_1 \dots x_n\colon A_n$ are given a denotation $[\![\Delta]\!] = [\![A_1]\!] \otimes \cdots \otimes [\![A_n]\!]$. When it is not ambiguous, the interpretation of the judgement $\Delta \vdash e\colon A$ is written $[\![e]\!]$.

$$
\begin{aligned}
[\![A]\!] &: \mathbf{Hilb} \\
[\![\mathtt{I}]\!] &= \mathbb{C} \\
[\![A \otimes B]\!] &= [\![A]\!] \otimes [\![B]\!] \\
[\![A \oplus B]\!] &= [\![A]\!] \oplus [\![B]\!] \\
[\![\mathtt{Nat}]\!] &= \ell^2(\mathbb{N})
\end{aligned}
$$

Figure 3.9 – Interpretation of types.

$$
\begin{aligned}
[\![\Delta \vdash e \colon A]\!] &: \mathbf{Isom}([\![\Delta]\!], [\![A]\!]) \\
[\![\vdash * \colon \mathtt{I}]\!] &= \mathrm{id}_{[\![\mathtt{I}]\!]} \\
[\![x \colon A \vdash x \colon A]\!] &= \mathrm{id}_{[\![A]\!]} \\
[\![\Delta \vdash \mathtt{inj}_l\, e \colon A \oplus B]\!] &= \iota_l^{[\![A]\!],[\![B]\!]} \circ [\![\Delta \vdash e \colon A]\!] \\
[\![\Delta \vdash \mathtt{inj}_r\, e \colon A \oplus B]\!] &= \iota_r^{[\![A]\!],[\![B]\!]} \circ [\![\Delta \vdash e \colon A]\!] \\
[\![\Delta_1, \Delta_2 \vdash e \otimes e' \colon A \otimes B]\!] &= [\![\Delta_1 \vdash e \colon A]\!] \otimes [\![\Delta_2 \vdash e' \colon B]\!] \\
[\![\vdash \mathtt{zero} \colon \mathtt{Nat}]\!] &= |0\rangle \\
[\![\Delta \vdash \mathtt{S}\, e \colon \mathtt{Nat}]\!] &= \mathrm{succ} \circ [\![\Delta \vdash e \colon \mathtt{Nat}]\!] \\
[\![\Delta \vdash \Sigma_{i \leq k}(\alpha_i \cdot e_i) \colon A]\!] &= \sum_{i \leq k} \alpha_i \, [\![\Delta \vdash e_i \colon A]\!]
\end{aligned}
$$

Figure 3.10 – Interpretation of expression judgements as morphisms in $\mathbf{Isom}$.

Lemma 3.52 (Isometry). *If $\Delta \vdash e \colon A$ is a well-formed expression judgement, then $[\![\Delta \vdash e \colon A]\!]$ is an isometry.*

Proof. Given later with the semantics of terms in general, see Lemma 3.64. □

In quantum physics, the state of a particle is usually described as an isometry. Showing that our expressions are interpreted as isometries, we can justify that they are correct *quantum states*. The proof of the previous lemma is included in one of a larger result, showing that the denotation of all terms are isometries (see Lemma 3.64). Moreover, expressions are used to define the unitary abstractions as a collection of patterns: it is sensible to prove that these patterns are interpreted with compatible morphisms, in the sense of Definition 3.36.

Lemma 3.53. *Given two judgements $\Delta_1 \vdash e_1 \colon A$ and $\Delta_2 \vdash e_2 \colon A$, such that $e_1 \perp e_2$, we have $[\![e_1]\!]^\dagger \circ [\![e_2]\!] = 0$.*

Proof. The proof is done by induction on the derivation of $\perp$. It is a subproof of the one for Lemma 3.63. □

This result can also be stated for the predicate OD, with the help of Lemma 3.17, where it is shown that two values in an orthogonal decomposition are orthogonal.

Lemma 3.54. *Given two judgements $\Delta_1 \vdash e_1 \colon A$ and $\Delta_2 \vdash e_2 \colon A$, a set of S that contains e_1 and e_2 and such that $\mathrm{OD}_A^{ext} S$, we have $[\![e_1]\!]^\dagger \circ [\![e_2]\!] = 0$.*

An important property of an orthonormal basis in a Hilbert space is the *resolution of the identity*. We show that, given $\mathrm{OD}_A^{ext} S$, a similar property holds. This conforts us in calling S a *syntactic basis*.

Lemma 3.55. *Given $\mathrm{OD}_A^{ext}\{e_i\}_{i\leq n}$, and $\Delta_i \vdash e_i \colon A$ for all i, we have*

$$\sum_{i\leq n} [\![e_i]\!] \circ [\![e_i]\!]^\dagger = \mathrm{id}_{[\![A]\!]}.$$

Proof. The proof is done by induction on OD.

- $\mathrm{OD}_A^{ext}\{x\}$. $[\![x]\!] \circ [\![x]\!]^\dagger = \mathrm{id}_{[\![A]\!]} \circ \mathrm{id}_{[\![A]\!]} = \mathrm{id}_{[\![A]\!]}$.
- $\mathrm{OD}_{\mathtt{I}}^{ext}\{*\}$. $[\![*]\!] \circ [\![*]\!]^\dagger = \mathrm{id}_{[\![\mathtt{I}]\!]} \circ \mathrm{id}_{[\![\mathtt{I}]\!]} = \mathrm{id}_{[\![\mathtt{I}]\!]}$.
- $\mathrm{OD}_{A\oplus B}^{ext} S \boxplus T$.

$$\begin{aligned}
&\sum_{e\in S\boxplus T} [\![e]\!] \circ [\![e]\!]^\dagger \\
&= \sum_{s\in S} [\![\mathtt{inj}_l\ s]\!] \circ [\![\mathtt{inj}_l\ s]\!]^\dagger + \sum_{t\in T} [\![\mathtt{inj}_r\ t]\!] \circ [\![\mathtt{inj}_r\ t]\!]^\dagger && \text{(by definition)} \\
&= \iota_l \circ \left(\sum_{s\in S} [\![s]\!] \circ [\![s]\!]^\dagger\right) \circ \iota_l^\dagger + \iota_r \circ \left(\sum_{t\in T} [\![t]\!] \circ [\![t]\!]^\dagger\right) \circ \iota_r^\dagger && \text{(by linearity)} \\
&= \iota_l \circ \mathrm{id}_{[\![A]\!]} \circ \iota_l^\dagger + \iota_r \circ \mathrm{id}_{[\![B]\!]} \circ \iota_r^\dagger && \text{(by IH)} \\
&= \iota_l\iota_l^\dagger + \iota_r\iota_r^\dagger = \mathrm{id}_{[\![A\oplus B]\!]}. && \text{(Ex. 3.44)}
\end{aligned}$$

- $\mathrm{OD}^{ext}_{A\otimes B}S$. Suppose that $\mathrm{OD}^{ext}_{A}\pi_1(S)$ and $\mathrm{OD}^{ext}_{B}S^1_e$ for all $e \in \pi_1(S)$.

$$
\begin{aligned}
&\sum_{(e\otimes e')\in S} [\![e\otimes e']\!] \circ [\![e\otimes e']\!]^\dagger \\
&= \sum_{(e\otimes e')\in S} ([\![e]\!]\otimes[\![e']\!]) \circ ([\![e]\!]\otimes[\![e']\!]^\dagger) && \text{(by definition)} \\
&= \sum_{(e\otimes e')\in S} ([\![e]\!]\circ[\![e]\!]^\dagger)\otimes([\![e']\!]\circ[\![e']\!]^\dagger) && \text{(by monoidal } \dagger\text{-category)} \\
&= \sum_{e\in\pi_1(S)} ([\![e]\!]\circ[\![e]\!]^\dagger)\otimes\left(\sum_{e'\in S^1_b}[\![e']\!]\circ[\![e']\!]^\dagger\right) \\
&= \sum_{e\in\pi_1(S)} ([\![e]\!]\circ[\![e]\!]^\dagger)\otimes \mathrm{id}_{[\![B]\!]} && \text{(by IH)} \\
&= \left(\sum_{e\in\pi_1(S)} [\![e]\!]\circ[\![e]\!]^\dagger\right)\otimes \mathrm{id}_{[\![B]\!]} && \text{(by linearity)} \\
&= \mathrm{id}_{[\![A]\!]}\otimes\mathrm{id}_{[\![B]\!]} = \mathrm{id}_{[\![A\otimes B]\!]}. && \text{(by IH)}
\end{aligned}
$$

- $\mathrm{OD}^{ext}_{\mathtt{Nat}}S^{\oplus 0}$.

$$
\begin{aligned}
&\sum_{e\in S^{\oplus 0}} [\![e]\!]\circ[\![e]\!]^\dagger \\
&= [\![\mathtt{zero}]\!]\circ[\![\mathtt{zero}]\!] + \sum_{s\in S}[\![\mathtt{S}\ s]\!]\circ[\![\mathtt{S}\ s]\!]^\dagger && \text{(by definition)} \\
&= [\![\mathtt{zero}]\!]\circ[\![\mathtt{zero}]\!]^\dagger + \mathrm{succ}\circ\left(\sum_{s\in S}[\![s]\!]\circ[\![s]\!]^\dagger\right)\circ\mathrm{succ}^\dagger && \text{(by linearity)} \\
&= [\![\mathtt{zero}]\!]\circ[\![\mathtt{zero}]\!]^\dagger + \mathrm{succ}\circ\mathrm{id}_{[\![\mathtt{Nat}]\!]}\circ\mathrm{succ}^\dagger = \mathrm{id}_{[\![\mathtt{Nat}]\!]}. && \text{(by IH)}
\end{aligned}
$$

- $\mathrm{OD}^{ext}_{A}S^{\alpha}$.

$$
\begin{aligned}
&\sum_{e\in S^\alpha} [\![e]\!]\circ[\![e]\!]^\dagger \\
&= \sum_{s\in S} [\![\Sigma_{s'\in S}(\alpha_{s,s'}\cdot s')]\!]\circ[\![\Sigma_{s'\in S}(\alpha_{s,s'}\cdot s')]\!]^\dagger && \text{(by definition)} \\
&= \sum_{s\in S}\left(\sum_{s'\in S}\alpha_{s,s'}[\![s']\!]\right)\circ\left(\sum_{s''\in S}\alpha_{s,s''}[\![s'']\!]\right)^\dagger && \text{(by definition)} \\
&= \sum_{s\in S}\sum_{s',s''\in S}\alpha_{s,s'}\overline{\alpha_{s,s''}}[\![s']\!]\circ[\![s'']\!]^\dagger && \text{(by linearity)} \\
&= \sum_{s',s''\in S}\left(\sum_{s\in S}\alpha_{s,s'}\overline{\alpha_{s,s''}}\right)[\![s']\!]\circ[\![s'']\!]^\dagger \\
&= \sum_{s',s''\in S}\delta_{s'=s''}[\![s']\!]\circ[\![s'']\!]^\dagger = \sum_{s'\in S}[\![s']\!]\circ[\![s']\!]^\dagger && \text{(by unitarity)} \\
&= \mathrm{id}_{[\![A]\!]}. && \text{(by IH)}
\end{aligned}
$$

□

One final development on the interpretation of values is the link with substitutions, detailed in the next proposition.

Proposition 3.56. *Given a well-typed term $\Delta \vdash t : A$ and for all $(x_i : A_i) \in \Delta$, a well-typed expression $\vdash e_i : A_i$; if $\sigma = \{x_i \mapsto e_i\}_i$, then:*

$$[\![\vdash \sigma(t) : A]\!] = [\![\Delta \vdash t : A]\!] \circ \left(\bigotimes_i [\![\vdash e_i : A_i]\!] \right).$$

We define then $[\![\sigma]\!] = \bigotimes_i [\![\vdash e_i : A_i]\!]$.

Proof. The proof is straightforward by induction on the typing rules for t. □

Remark 3.57. The definition of the interpretation of a substitution above is somewhat informal. It would require a lot of care and unnecessary details to make the denotation of σ fit the denotation of a particular context Δ. Since we are working in symmetric monoidal categories, those details will be overlooked when working with substitutions. We assume that we work up to permutations, and that when an interpretation of a substitution is involved, it is with the right permutation.

Substitutions σ emerge from the matching of two basis values, thus we can prove that the interpretation of the matching gives the interpretation of the substitution, as stated in the next lemma.

Lemma 3.58. *Given two well-typed basis values $\Delta \vdash b : A$ and $\vdash b' : A$, and a substitution σ, if $\mathrm{match}(\sigma, b, b')$ then $[\![b]\!]^\dagger \circ [\![b']\!] = [\![\sigma]\!]$.*

Proof. The proof is straightforward by induction on $\mathrm{match}(\sigma, b, b')$ (see §3.2.3 for the definition). □

Unitaries. The type of unitaries are given as $A \leftrightarrow B$, and they are first interpreted as morphisms $[\![A]\!] \to [\![B]\!]$ in $\mathbf{Contr}$, before showing that their interpretation actually lies in $\mathbf{Uni}$. We also show that the OD conditions ensure that the denotation of (syntactic) untaries is not only a contractive map, but a unitary between Hilbert spaces. Working with contractions is necessary to use the notion of compatibility: given a unitary $\{ \mid b_1 \leftrightarrow e_1 \mid \ldots \mid b_n \leftrightarrow e_n \} : A \leftrightarrow B$, we provide an interpretation to each clause $b_i \leftrightarrow e_i$ as a contraction $[\![A]\!] \to [\![B]\!]$, and prove that all the contractions thus obtained are compatible, and can be summed. Unitary judgments are of the form $\vdash_\omega \omega : A \leftrightarrow B$, and their semantics is given by a morphism in $\mathbf{Uni}$:

$$[\![\vdash_\omega \omega : A \leftrightarrow B]\!] : \mathbf{Uni}([\![A]\!], [\![B]\!]).$$

Given $\vdash_\omega \{ \mid b_1 \leftrightarrow e_1 \mid \ldots \mid b_n \leftrightarrow e_n \} : A \leftrightarrow B$, the interpretation of a clause $b_i \leftrightarrow e_i$ is the following contraction: $[\![\Delta_i \vdash e_i : B]\!] \circ [\![\Delta_i \vdash b_i : A]\!]^\dagger$. It should be read as follows: if the

input of type A matches with b_i, it provides a substitution through Δ_i, that is applied to e_i. This is better understood through a diagram:

$$[\![A]\!] \xrightarrow{[\![\Delta_i \vdash b_i \colon A]\!]^\dagger} [\![\Delta_i]\!] \xrightarrow{[\![\Delta_i \vdash e_i \colon B]\!]} [\![B]\!]$$

The interpretation of a unitary abstraction is then:

$$[\![\vdash_\omega \{ \mid b_1 \leftrightarrow e_1 \mid \ldots \mid b_n \leftrightarrow e_n \} : A \leftrightarrow B]\!] = \sum_{i \leq n} [\![\Delta_i \vdash e_i \colon B]\!] \circ [\![\Delta_i \vdash b_i \colon A]\!]^\dagger .$$

It is left to prove that it is well-defined, and then that it is a proper unitary operation.

Corollary 3.59. *Given* $\vdash_\omega \{ \mid b_1 \leftrightarrow e_1 \mid \ldots \mid b_n \leftrightarrow e_n \} : A \leftrightarrow B$*, its interpretation* $[\![\vdash_\omega \{ \mid b_1 \leftrightarrow e_1 \mid \ldots \mid b_n \leftrightarrow e_n \} : A \leftrightarrow B]\!]$ *is a well-defined morphism in* $\mathbf{Contr}$.

Proof. Given $\vdash_\omega \{ \mid b_1 \leftrightarrow e_1 \mid \ldots \mid b_n \leftrightarrow e_n \} : A \leftrightarrow B$, we know that $\mathrm{OD}_B^{ext}(\{e_i\}_{i \leq n})$ and $\mathrm{OD}_A(\{b_i\}_{i \leq n})$ hold, and for all $i \leq n$, $\Delta_i \vdash b_i \colon A$ and $\Delta_i \vdash e_i \colon B$.

Since $\mathrm{OD}_A(\{b_i\}_{i \leq n})$ holds, Lemma 3.54 ensures that for all $i \neq j \leq k$, $[\![b_i]\!]^\dagger \circ [\![b_j]\!] = 0_{[\![\Delta_j]\!],[\![\Delta_i]\!]}$. The same lemma with $\mathrm{OD}_B^{ext}(\{e_i\}_{i \leq n})$ ensures that for all $i \neq j \leq n$, $[\![e_i]\!]^\dagger \circ [\![e_j]\!] = 0_{[\![\Delta_j]\!],[\![\Delta_i]\!]}$. This proves that, for all $i \neq j \leq n$, $([\![e_i]\!] \circ [\![b_i]\!]^\dagger)^\dagger \circ [\![e_j]\!] \circ [\![b_j]\!]^\dagger = 0_{[\![\Delta_j]\!],[\![\Delta_i]\!]}$ and $[\![e_i]\!] \circ [\![b_i]\!]^\dagger \circ ([\![e_j]\!] \circ [\![b_j]\!]^\dagger)^\dagger = 0_{[\![\Delta_j]\!],[\![\Delta_i]\!]}$. This proves that for all $i \neq j \leq n$, $[\![e_i]\!] \circ [\![b_i]\!]^\dagger$ and $[\![e_j]\!] \circ [\![b_j]\!]^\dagger$ are compatible, thanks to Lemma 3.39. Then, Lemma 3.37 ensures that $\sum_{i \leq n} [\![e_i]\!] \circ [\![b_i]\!]^\dagger$ is a contraction. □

Theorem 3.60. *Given* $\vdash_\omega \{ \mid b_1 \leftrightarrow e_1 \mid \ldots \mid b_n \leftrightarrow e_n \} : A \leftrightarrow B$*, its interpretation* $[\![\vdash_\omega \{ \mid b_1 \leftrightarrow e_1 \mid \ldots \mid b_n \leftrightarrow e_n \} : A \leftrightarrow B]\!]$ *is unitary.*

Proof. Given $\vdash_\omega \{ \mid b_1 \leftrightarrow e_1 \mid \ldots \mid b_n \leftrightarrow e_n \} : A \leftrightarrow B$, we know that $\mathrm{OD}_B^{ext}(\{e_i\}_{i \leq n})$ and $\mathrm{OD}_A(\{b_i\}_{i \leq n})$ hold, and for all $i \leq n$, $\Delta_i \vdash b_i \colon A$ and $\Delta_i \vdash e_i \colon B$.

First, we prove that $[\![\omega]\!]^\dagger \circ [\![\omega]\!] = \mathrm{id}_{[\![A]\!]}$, with $\omega = \{ \mid b_1 \leftrightarrow e_1 \mid \ldots \mid b_n \leftrightarrow e_n \}$.

$$\begin{aligned}
&[\![\omega]\!]^\dagger \circ [\![\omega]\!] \\
&= \left(\sum_{i \leq n} [\![e_i]\!] \circ [\![b_i]\!]^\dagger \right)^\dagger \circ \sum_{j \leq n} [\![e_j]\!] \circ [\![b_j]\!]^\dagger && \text{(by definition)} \\
&= \sum_{i \leq n} ([\![e_i]\!] \circ [\![b_i]\!]^\dagger)^\dagger \circ \sum_{j \leq n} [\![e_j]\!] \circ [\![b_j]\!]^\dagger && \text{(dagger distributes over sum)} \\
&= \sum_{i \leq n} [\![b_i]\!] \circ [\![e_i]\!]^\dagger \circ \sum_{j \leq n} [\![e_j]\!] \circ [\![b_j]\!]^\dagger && \text{(dagger is contravariant)} \\
&= \sum_{i,j \leq n} [\![b_i]\!] \circ [\![e_i]\!]^\dagger \circ [\![e_j]\!] \circ [\![b_j]\!]^\dagger && \text{(linearity)} \\
&= \sum_{i \leq n} [\![b_i]\!] \circ [\![e_i]\!]^\dagger \circ [\![e_i]\!] \circ [\![b_i]\!]^\dagger && \text{(Lemma 3.54)} \\
&= \sum_{i \leq n} [\![b_i]\!] \circ [\![b_i]\!]^\dagger && \text{(Lemma 3.52)} \\
&= \mathrm{id}_{[\![A]\!]} && \text{(Lemma 3.55)}
\end{aligned}$$

The other direction $[\![\omega]\!] \circ [\![\omega]\!]^\dagger = \mathrm{id}_{[\![B]\!]}$ is similar. □

Note that we have only proven so far that unitary abstractions have a sound denotational semantics in $\mathbf{Uni}$. The interpretation of operations on unitaries is given in Figure 3.11. It is explained in §3.4 why this interpretation is in $\mathbf{Uni}$, although this does not come as a surprise.

$$
\begin{aligned}
[\![\vdash_\omega \omega \colon A \leftrightarrow B]\!] &\colon \mathbf{Uni}([\![A]\!], [\![B]\!]) \\
[\![\{ \mid \ b_i \leftrightarrow e_i \ \}_{i \in I}]\!] &= \sum_{i \in I} [\![e_i]\!] \circ [\![b_i]\!]^\dagger \\
[\![\omega_2 \circ \omega_1]\!] &= [\![\omega_2]\!] \circ [\![\omega_1]\!] \\
[\![\omega_1 \otimes \omega_2]\!] &= [\![\omega_1]\!] \otimes [\![\omega_2]\!] \\
[\![\omega_1 \oplus \omega_2]\!] &= [\![\omega_1]\!] \oplus [\![\omega_2]\!] \\
[\![\omega^{-1}]\!] &= [\![\omega]\!]^\dagger \\
[\![\mathtt{ctrl}\ \omega]\!] &= \mathrm{ctrl}_{[\![A]\!]}([\![\omega]\!])
\end{aligned}
$$

Figure 3.11 – Interpretation of unitaries in $\mathbf{Uni}$.

Terms. One remaining term is the application of a unitary.

$$[\![\Delta \vdash \omega\ t \colon B]\!] = [\![\vdash_\omega \omega \colon A \leftrightarrow B]\!] \circ [\![\Delta \vdash t \colon A]\!] .$$

The interpretation of all term judgements is found in Figure 3.12.

$$
\begin{aligned}
[\![\Delta \vdash t \colon A]\!] &\colon \mathbf{Isom}([\![\Delta]\!], [\![A]\!]) \\
[\![\vdash * \colon \mathtt{I}]\!] &= \mathrm{id}_{[\![\mathtt{I}]\!]} \\
[\![x \colon A \vdash x \colon A]\!] &= \mathrm{id}_{[\![A]\!]} \\
[\![\Delta \vdash \mathtt{inj}_l\, t \colon A \oplus B]\!] &= \iota_l^{[\![A]\!],[\![B]\!]} \circ [\![\Delta \vdash t \colon A]\!] \\
[\![\Delta \vdash \mathtt{inj}_r\, t \colon A \oplus B]\!] &= \iota_r^{[\![A]\!],[\![B]\!]} \circ [\![\Delta \vdash t \colon A]\!] \\
[\![\Delta_1, \Delta_2 \vdash t \otimes t' \colon A \otimes B]\!] &= [\![\Delta_1 \vdash t \colon A]\!] \otimes [\![\Delta_2 \vdash t' \colon B]\!] \\
[\![\vdash \mathtt{zero} \colon \mathtt{Nat}]\!] &= |0\rangle \\
[\![\Delta \vdash \mathtt{S}\ t \colon \mathtt{Nat}]\!] &= \mathrm{succ} \circ [\![\Delta \vdash t \colon \mathtt{Nat}]\!] \\
[\![\Delta \vdash \Sigma_{i \leq k}(\alpha_i \cdot t_i) \colon A]\!] &= \sum_{i \leq k} \alpha_i\, [\![\Delta \vdash t_i \colon A]\!] \\
[\![\Delta \vdash \omega\ t \colon B]\!] &= [\![\vdash_\omega \omega \colon A \leftrightarrow B]\!] \circ [\![\Delta \vdash t \colon A]\!]
\end{aligned}
$$

Figure 3.12 – Interpretation of term judgements as morphisms in $\mathbf{Isom}$.

We can already show that this interpretation of terms is sound with the sketch of operational semantics given in the previous section.

Proposition 3.61 (Operational Soundness). *Given a well-typed unitary abstraction* $\vdash_\omega \{ \mid b_1 \leftrightarrow e_1 \mid \ldots \mid b_n \leftrightarrow e_n \} : A \leftrightarrow B$ *and a well-typed basis value* $\vdash b' \colon A$, *if* $\mathrm{match}(\sigma, b_i, b')$, *then*

$$[\![\vdash \{ \mid b_1 \leftrightarrow e_1 \mid \ldots \mid b_n \leftrightarrow e_n \} \ b' \colon B]\!] = [\![\vdash \sigma(e_i) \colon B]\!] .$$

Proof. First, we deduce from the assumption $\mathrm{match}(\sigma, b_i, b')$ that

- $[\![b_i]\!]^\dagger \circ [\![b']\!] = [\![\sigma]\!]$, thanks to Lemma 3.58.
- for all $j \neq i$, $b_j \perp b'$, and thus $[\![b_j]\!]^\dagger \circ [\![b']\!] = 0$, thanks to Lemma 3.53.

We can then compute the semantics, with $\omega \stackrel{\text{def}}{=} \{ \mid b_1 \leftrightarrow e_1 \mid \ldots \mid b_n \leftrightarrow e_n \}$:

$$\begin{aligned}
&[\![\omega \ b']\!] \\
&= [\![\omega]\!] \circ [\![b']\!] && \text{(by definition)} \\
&= \left(\sum_j [\![e_j]\!] \circ [\![b_j]\!]^\dagger \right) \circ [\![b']\!] && \text{(by definition)} \\
&= \sum_j [\![e_j]\!] \circ [\![b_j]\!]^\dagger \circ [\![b']\!] && \text{(linearity)} \\
&= [\![e_i]\!] \circ [\![b_i]\!]^\dagger \circ [\![b']\!] && \text{(Lemma 3.53)} \\
&= [\![e_i]\!] \circ [\![\sigma]\!] && \text{(Lemma 3.58)} \\
&= [\![\sigma(e_i)]\!] && \text{(Prop. 3.56)}
\end{aligned}$$

□

We prove that, like expressions, terms are indeed interpreted as isometries, reinforcing the link with quantum physics. This requires several lemmas. The first lemma shows that the denotational orthogonality is preserved by linear combinations.

Lemma 3.62. *Given* $\Delta_1 \vdash t \colon A$ *and* $\Delta_2 \vdash \Sigma_i(\alpha_i \cdot t_i) \colon A$ *such that for all* i, $[\![\Delta_1 \vdash t \colon A]\!]^\dagger \circ [\![\Delta_2 \vdash t_i \colon A]\!] = 0_{[\![\Delta_2]\!],[\![\Delta_1]\!]}$; *then* $[\![\Delta_1 \vdash t \colon A]\!]^\dagger \circ [\![\Delta_2 \vdash \Sigma_i(\alpha_i \cdot t_i) \colon A]\!] = 0_{[\![\Delta_2]\!],[\![\Delta_1]\!]}$.

Proof. The proof involves few steps, without surprises.

$$\begin{aligned}
&[\![\Delta_1 \vdash t \colon A]\!]^\dagger \circ [\![\Delta_2 \vdash \Sigma_i(\alpha_i \cdot t_i) \colon A]\!] \\
&= [\![\Delta_1 \vdash t \colon A]\!]^\dagger \circ \sum_i \alpha_i \, [\![\Delta_2 \vdash t_i \colon A]\!] && \text{(by definition)} \\
&= \sum_i \alpha_i \, [\![\Delta_1 \vdash t \colon A]\!]^\dagger \circ [\![\Delta_2 \vdash t_i \colon A]\!] && \text{(by linearity)} \\
&= \sum_i \alpha_i \, 0_{[\![\Delta_2]\!],[\![\Delta_1]\!]} && \text{(hypothesis)} \\
&= 0_{[\![\Delta_2]\!],[\![\Delta_1]\!]}.
\end{aligned}$$

□

The next lemma states that syntactic orthogonality (defined in Definition 3.3) implies denotational orthogonality.

Lemma 3.63. *Given two judgements $\Delta_1 \vdash t_1 \colon A$ and $\Delta_2 \vdash t_2 \colon A$, such that $t_1 \perp t_2$, we have $[\![t_1]\!]^\dagger \circ [\![t_2]\!] = 0_{[\![\Delta_2]\!],[\![\Delta_1]\!]}$.*

The proof of this lemma is interdependent with the proof of the next lemma, where it is proven that the interpretations of term judgements are normalised quantum states – namely, isometries.

Lemma 3.64 (Isometry)**.** *If $\Delta \vdash t \colon A$ is a well-formed judgement, then $[\![\Delta \vdash t \colon A]\!]$ is an isometry.*

Proof of Lemma 3.63 and Lemma 3.64. We prove both theorems together, because they are interdependent. We recall the whole statement: given a well-formed judgement $\Delta_1 \vdash t_1 \colon A$,

- $[\![\Delta_1 \vdash t_1 \colon A]\!]^\dagger \circ [\![\Delta_1 \vdash t_1 \colon A]\!] = \mathrm{id}_{[\![\Delta_1]\!]}$;
- for all $\Delta_2 \vdash t_2 \colon A$ such that $t_1 \perp t_2$, $[\![\Delta_1 \vdash t_1 \colon A]\!]^\dagger \circ [\![\Delta_2 \vdash t_2 \colon A]\!] = 0_{[\![\Delta_2]\!],[\![\Delta_1]\!]}$.

We prove by induction on the derivation of $\Delta_1 \vdash t_1 \colon A$.

- $\vdash * \colon \mathtt{I}$. $[\![*]\!]^\dagger \circ [\![*]\!] = \mathrm{id}_{[\![\mathtt{I}]\!]} \circ \mathrm{id}_{[\![\mathtt{I}]\!]} = \mathrm{id}_{[\![\mathtt{I}]\!]}$. This term is not orthogonal to any other well-typed term.
- $x \colon A \vdash x \colon A$. $[\![x]\!]^\dagger \circ [\![x]\!] = \mathrm{id}_{[\![A]\!]} \circ \mathrm{id}_{[\![A]\!]} = \mathrm{id}_{[\![A]\!]}$. This term is not orthogonal to any other.
- $\Delta_1, \Delta_2 \vdash t_1 \otimes t_2 \colon A \otimes B$. We first prove that its denotation is an isometry.

$$
\begin{aligned}
&[\![t_1 \otimes t_2]\!]^\dagger \circ [\![t_1 \otimes t_2]\!] \\
&= ([\![t_1]\!] \otimes [\![t_2]\!])^\dagger \circ ([\![t_1]\!] \otimes [\![t_2]\!]) && \text{(by definition)} \\
&= ([\![t_1]\!]^\dagger \otimes [\![t_2]\!]^\dagger) \circ ([\![t_1]\!] \otimes [\![t_2]\!]) && \text{(dagger is a monoidal functor)} \\
&= ([\![t_1]\!]^\dagger \circ [\![t_1]\!]) \otimes ([\![t_2]\!]^\dagger \circ [\![t_2]\!]) && (\otimes \text{ is monoidal}) \\
&= \mathrm{id}_{[\![\Delta_1]\!]} \otimes \mathrm{id}_{[\![\Delta_2]\!]} && \text{(IH)} \\
&= \mathrm{id}_{[\![\Delta_1]\!] \otimes [\![\Delta_2]\!]} = \mathrm{id}_{[\![\Delta_1, \Delta_2]\!]}
\end{aligned}
$$

  Then we show that, if it is orthogonal to any other well-typed term, say $\Delta_3 \vdash t_3 \colon C$, then their interpretations are also orthogonal. We reason on a case by case basis. The term t_3 can be of the form $t_3' \otimes t_3''$, in which case, either $t_1 \perp t_3'$ or $t_2 \perp t_3''$. In both cases, the result is direct, because the zero morphism tensored with any other morphism is still zero. The other cases is t_3 being a linear combination; this case is covered by Lemma 3.62.
- $\Delta \vdash \mathtt{inj}_i\, t \colon A_1 \oplus A_2$. The denotation of this term is an isometry because the injections ι also are and a composition of isometries keeps being an isometry. Now, given another term $\Delta_1 \vdash t_2 \colon B$ such that $\mathtt{inj}_i\, t \perp t_2$, we have several cases:
  - either t_2 is of the form $\mathtt{inj}_i\, t_2'$ with the same injection as $\mathtt{inj}_i\, t$, and the result is given by Lemma 3.46;
  - or $\mathtt{inj}_i\, t = \mathtt{inj}_l\, t$ and $t_2 = \mathtt{inj}_r\, t_2'$, in which case the conclusion is direct thanks to Lemma 3.43 and the induction hypothesis;
  - or t_2 is a linear combination, and Lemma 3.62 concludes.
- $\vdash \mathtt{zero} \colon \mathtt{Nat}$. $[\![\mathtt{zero}]\!]^\dagger \circ [\![\mathtt{zero}]\!] = \langle 0|0\rangle = 1 = \mathrm{id}_{[\![\mathtt{I}]\!]}$. Moreover, given any natural number n, $\langle 0|n+1\rangle = 0$. Lemma 3.62 concludes for linear combinations.

- $\Delta \vdash \mathtt{S}\, t \colon \mathtt{Nat}$. The interpretation is an isometry by composition of isometries. The orthogonality part is either ensured with the last point, or by Lemma 3.46 and the induction hypothesis.
- $\Delta \vdash \Sigma_i(\alpha_i \cdot t_i) \colon A$. The interpretation is an isometry by induction hypothesis and Lemma 3.45. Moreover, the only case of orthogonality not yet covered is $t_2 = \Sigma_k(\beta_k \cdot t_k)$ where for all $j \neq k$, $t_j \perp t_k$ and $\sum \alpha_j \overline{\beta}_k = 0$.
  given that for all $i \neq j \in I$, $t_i \perp t_j$, $J, K \subseteq I$, and $\sum_{i \in J \cap K} \overline{\alpha}_i \beta_i = 0$:

$$
\begin{aligned}
&[\![\Sigma_j(\alpha_j \cdot t_j)]\!]^\dagger \circ [\![\Sigma_k(\beta_k \cdot t_k)]\!] \\
&= \left(\sum_j \alpha_j \, [\![t_j]\!] \right)^\dagger \circ \sum_k \beta_k \, [\![t_k]\!] && \text{(by definition)} \\
&= \left(\sum_j \overline{\alpha}_j \, [\![t_j]\!]^\dagger \right) \circ \sum_k \beta_k \, [\![t_k]\!] && \text{(dagger distributes over the sum)} \\
&= \sum_{j,k} \overline{\alpha}_j \beta_k \, [\![t_j]\!]^\dagger \circ [\![t_k]\!] && \text{(by linearity)} \\
&= \sum_{i \in J \cap K} \overline{\alpha}_i \beta_i \, [\![t_i]\!]^\dagger \circ [\![t_i]\!] && \text{(by induction hypothesis)} \\
&= \sum_{i \in J \cap K} \overline{\alpha}_i \beta_i \, \mathrm{id}_{[\![\Delta]\!]} && \text{(by induction hypothesis)} \\
&= \left(\sum_{i \in J \cap K} \overline{\alpha}_i \beta_i \right) \mathrm{id}_{[\![\Delta]\!]} = 0_{[\![\Delta_2]\!], [\![\Delta_1]\!]}
\end{aligned}
$$

- $\Delta \vdash \omega \; t \colon B$. The interpretation is an isometry by composition of isometries. The orthogonality is covered by either Lemma 3.46, because a unitary is in particular an isometry, or Lemma 3.62, similarly to the previous points.

□

Providing an interpretation that fits the expectations from quantum physics is meaningful, but not completely satisfying. We prove a stronger link in the coming section between the syntax and the semantics through the equational theory, in the vein of the previous chapter.

3.5.2 Completeness

We prove a strong link between the denotational semantics and the equational theory, namely that an equality statement in one is also an equality statement in the other. We start by showing soundness, meaning that two terms equal in the equational theory have the same denotational interpretation.

Proposition 3.65. *Given* $\Delta \vdash t_1 = t_2 \colon A$, *then* $[\![\Delta \vdash t_1 \colon A]\!] = [\![\Delta \vdash t_2 \colon A]\!]$.

Proof. By induction on the rules of the equational theory. The only non-trivial case is done within Proposition 3.61. □

We prove then completeness, starting with terms without unitary functions.

Lemma 3.66 (Completeness of values). *Given* $\vdash v_1 : A$ *and* $\vdash v_2 : A$, *if* $[\![\vdash v_1 : A]\!] = [\![\vdash v_2 : A]\!]$, *then* $\vdash v_1 = v_2 : A$.

Proof. This is proven by induction on $\vdash v_1 : A$.

- The cases of $*$ and $\mathtt{zero}$ are straightforward.
- If $v_1 = b_1 \otimes b_1'$ with type $A \otimes B$, then also $v_2 = b_2 \otimes b_2'$, and $[\![b_1]\!] \otimes [\![b_1']\!] = [\![b_2]\!] \otimes [\![b_2']\!]$, thus $[\![b_1]\!] = [\![b_2]\!]$ and $[\![b_1']\!] = [\![b_2']\!]$, the induction hypothesis ensures that $\vdash b_1 = b_2 : A$ and $\vdash b_1' = b_2' : B$, and thus $\vdash b_1 \otimes b_1' = b_2 \otimes b_2' : A \otimes B$, since we are working with basis values.
- If $v_1 = \mathtt{inj}_i\, b_1$ of type $A_1 \oplus A_2$, with $[\![v_1]\!] = [\![v_2]\!]$, necessarily $v_2 = \mathtt{inj}_i\, b_2$, and $\iota_l^{A,B} \circ [\![b_1]\!] = \iota_l^{A,B} \circ [\![b_2]\!]$, thus $(\iota_l^{A,B})^\dagger \circ \iota_l^{A,B} \circ [\![b_1]\!] = (\iota_l^{A,B})^\dagger \circ \iota_l^{A,B} \circ [\![b_2]\!]$ which ends with $[\![b_1]\!] = [\![b_2]\!]$, and the induction hypothesis gives that $\vdash b_1 = b_2 : A_i$, and thus $\vdash \mathtt{inj}_i\, b_1 = \mathtt{inj}_i\, b_2 : A_i$.
- Else, with type A, $v_1 = \Sigma_i(\alpha_i \cdot b_i^1)$ with the (b_i^1) that are pairwise orthogonal, and $v_2 = \Sigma_j(\beta_j \cdot b_j^2)$ with the (b_j^2) that are also pairwise orthogonal. We know that

$$[\![v_1]\!] = \sum_i \alpha_i \, [\![b_i^1]\!] = \sum_j \beta_j \, [\![b_j^2]\!] = [\![v_2]\!] \,.$$

  Thus, for all $\vdash b : A$, $[\![b]\!]^\dagger \circ \sum_i \alpha_i \, [\![b_i^1]\!] = [\![b]\!]^\dagger \circ \sum_j \beta_j \left[\!\!\left[b_j^2\right]\!\!\right]$; by orthogonality, we have $[\![b]\!]^\dagger \circ \sum_i \alpha_i \, [\![b_i^1]\!] = \alpha_k \, [\![b_k^1]\!]$ for some k, and $[\![b]\!]^\dagger \circ \sum_j \beta_j \left[\!\!\left[b_j^2\right]\!\!\right] = \beta_{k'} \, [\![b_{k'}^2]\!]$ for some k'. Thus, $\alpha_k \, [\![b_k^1]\!] = \beta_{k'} \, [\![b_{k'}^2]\!]$, and because they are basis values, we have $\alpha_k = \beta_{k'}$ and $[\![b_k^1]\!] = [\![b_{k'}^2]\!]$, and the induction hypothesis ensures that $\vdash b_k^1 = b_{k'}^2 : A$. Note that this is done for all $\vdash b : A$, and thus $\vdash v_1 = v_2 : A$.

□

We then prove the final result of this section – namely, completeness on closed terms –, meaning that the equality statement of the equational theory and of the denotational semantics on closed terms are equivalent.

Theorem 3.67 (Completeness). $\vdash t_1 = t_2 : A$ *iff* $[\![\vdash t_1 : A]\!] = [\![\vdash t_2 : A]\!]$.

Proof. We prove both directions.

- For the implication $\vdash t_1 = t_2 : A$ to $[\![\vdash t_1 : A]\!] = [\![\vdash t_2 : A]\!]$, see Proposition 3.65.
- The other direction uses Theorem 3.35, the equivalent of strong normalisation, that gives v_1 and v_2 such that $\vdash t_1 = v_1 : A$ and $\vdash t_2 = v_2 : A$. Our hypothesis is that $[\![t_1]\!] = [\![t_2]\!]$, and thus $[\![v_1]\!] = [\![t_1]\!] = [\![t_2]\!] = [\![v_2]\!]$. Lemma 3.66 gives then that $\vdash v_1 = v_2 : A$ and transitivity ensures that $\vdash t_1 = t_2 : A$.

□

3.6 Discussion and conclusion

This section concludes with personal comments on the choices made. We discuss the parts of the language presented in [SVV18] that are *missing* in this chapter, and we give a short explanation on why they could or could not be included in the chapter.

3.6.1 Inductive types, Higher-order, Recursion

The points discussed in this section are concerned with features that we add to the language in the next chapter. In that chapter, we work with the reversible language based on symmetric pattern-matching, but without the quantum effect. This happens for two reasons. Firstly, it is new in the literature even in the classical case; and secondly, because the mathematical structure for *classical* reversibility is more suitable. We explain later in the thesis the limitations encountered in adding these features to quantum computing.

Inductive types. General inductive types can be added to the language without any issue. In [SVV18], the quantum language only works with lists, but all their results regarding the syntax hold even when generalising with a fixed point combinator for types, usually written $\mu X.A$ (see the explanations in §1.2.2). The denotational semantics of inductive types, in the context of countably-dimensional Hilbert spaces and isometries, is not hard to achieve, as observed by Michael Barr [Bar92, Theorem 3.2]. This means that, instead of working with a type `Nat`, the language presented in this chapter could work with a general fixed point combinator for types.

Higher-order. In this chapter, the functions of the language are called *unitaries* because of their interpretation as unitary maps between Hilbert spaces. They are the only operations allowed by quantum mechanics, alongside state preparation and measurement. Their treatment in the grammar and in the type system is separated from terms; even more, unitaries are not terms. This is different from the traditional λ-calculus. There are multiple reasons that justify this choice. We adopt a denotational point of view and comment on this choice through the mathematical model. Indeed, models of classical programming languages often involve a *closed* category – meaning that function types can be interpreted as objects of the category. However, the category of countably-dimensional Hilbert spaces and unitary maps is not closed. We will see in the next chapter how to go around this limitation with the enrichment of categories. Regarding the syntax, the language can be extended with *extended values* as they are called in the original paper; they are expressions that can contain not only variables at the ground level, but also function variables. These extended values can be used on the right-hand side of an abstraction, as in the following example:

$$\{|\ x \leftrightarrow \texttt{let}\, y = \phi\ x\ \texttt{in}\ y\}$$

where ϕ can be any well-typed function, including a function variable. Once we include those variables, we can introduce higher-order functions such as $\lambda\phi.\omega$, and application of those functions.

Recursion. Once function variables are introduced, one can add a fixed point combinator for recursion: for example, given a function ω, we can introduce a function $\texttt{fix}\ \phi.\omega$ that is the fixed point of ω with regard to the function variable ϕ. However, the functions $\texttt{fix}\ \phi.\phi$ and $\texttt{fix}\ \phi.\{|\ x \leftrightarrow \texttt{let}\, y = \phi\ x\ \texttt{in}\ y\}$ do not terminate, and thus cannot be interpreted as unitaries between Hilbert spaces. Moreover, we have yet to find a mathematical interpretation for such fixed points in Hilbert spaces; and Hilbert spaces might not be the solution. This is discussed in Chapter 5.

3.6.2 Conclusion

We have presented a programming language equipped with simple types aimed at quantum control through an algebraic effect. This is done through a syntax that allows for linear combinations of terms, and a type system which ensures that the latter are normalised. Then, we have formalised an equational theory, preserving the typing judgements and handling linear algebra as well as the computational aspect of the language. Finally, we have provided a denotational semantics, proven complete with regard to the equational theory.

Chapter 4
Reversibility and Fixed Points

> "Should mathematical semantics still be conceived as following in the track of pre-existing languages, trying to explain their novel features, and to provide firm foundations for them? Or should it be seen as operating in a more autonomous fashion, developing new semantic paradigms, which may then give rise to new languages?" — Samson Abramsky, in [Abr20].

Abstract

This chapter is concerned with the expressivity and denotational semantics of a functional higher-order reversible programming language based on Theseus. In this language, pattern-matching is used to ensure the reversibility of functions. We then build a sound and adequate categorical semantics based on join inverse categories, with additional structures to capture pattern-matching. We show how one can encode any Reversible Turing Machine in said language. Finally, we derive a full completeness result, stating that any computable, partial injective function is the image of a term in the language.

References. This work is based upon a paper, under submission, coauthored with Kostia Chardonnet and Benoît Valiron. The preprint is available at [CLV23].

4.1 Introduction

As said in the previous chapter, reversible computation has emerged as a energy-preserving model of computation in which no data is ever erased. This comes from Laundauer's principle which states that the erasure of information is linked to the dissipation of energy as heat [Lan61, BAP+12]. In reversible computation, given some process f, there always exists an inverse process f^{-1} such that their composition is equal to the identity: it is always possible to "*go back in time*" and recover the input of your computation. Although this can be seen as very restrictive, as shown for instance in [Ben73] non-reversible computation can be emulated in a reversible setting, by keeping track of intermediate results.

In order to avoid erasure of information, reversible computation often makes use of *garbage* or *auxiliary wires*: additional information kept in order to ensure both reversibility and the non-erasure of information. In programming languages, this is done by ensuring both *forward* and *backward* determinism. Forward determinism is almost always ensured in programming languages: it is about making sure that, given some state of your system, there is a unique next state that it can go to. Backward determinism on the other hand makes sure that for given a state, there is only one original state possible.

Reversible computation has since been shown to be a versatile model. In the realm of quantum computation, reversible computing is at the root of the construction of *oracles*, subroutines describing problem instances in quantum algorithms [NC02]. Most of the research in reversible circuit design can then been repurposed to design efficient quantum circuits. On the theoretical side, reversible computing serves as the main ingredient in several operational models of linear logics, whether through token-based Geometry of Interaction [Mac95] or through the Curry-Howard correspondence for μMALL [CSV23, Cha23].

Reversible programming has been approached in two different ways. The first one, based on Janus and later R-CORE and R-WHILE [Lut86, YG07, GKY19, YAG16], considers imperative and flow-chart languages. The other one follows a functional approach [YAG12, TA15, JS14, JKT18, SVV18, CSV23]: a function $A \to B$ in the language represents a function – a bijection – between values of type A and values of type B. In this approach, types are typically structured, and functional reversible languages usually feature pattern-matching to discriminate on values.

One of the issue reversible programming has to deal with is *non-termination*: in general, a reversible program computes a *partial injective map*. This intuition can be formalised with the concept of *inverse categories* [Kas79, CL02, CL03, CL07]: categories in which every morphism comes with a partial inverse, for which the category $\mathbf{PInj}$ of sets and partial injective maps is the emblematic concrete instance.

This categorical setting has been successfully used in the study of reversible programming semantics, whether based on flow-charts [GK18, Kaa19a], with recursion [AK16, KAG17, Kaa19b, KV19], with side-effects [HK15, HKK18b], *etc.*

Although much work has been dedicated to the categorical analysis of reversible computation, the *adequacy* of the developed categorical constructs with reversible functional programming languages has only recently been under scrutiny, either in *concrete* categories of partial isomorphisms [KV19, KR21], or for simple, *non Turing-complete* languages [CLV21]. A formal, categorical analysis of a Turing-complete, reversible language is still missing. Turing-completeness for a reversible programming language might come as a surprise; however, the literature is already filled with evidence that any irreversible computation can be simulated in a reversible setting [Ben73, Ben82, Abr05].

4.1.1 Related work

The work in [KAG17] is foundational in the development of the semantics of reversible programming languages. In that paper, the authors lay the foundations for the interpretation of what a programmer needs: data types and loops. However, no example of practical use is shown for this model. This is covered in another paper [KR21], where a denotational semantics of the reversible programming language `RFun` is provided. The latter is not typed and is conceptually

further from, say, a simply-typed λ-calculus. We argue that the language in [SVV18] is a more interesting case study.

The language we bring under scrutiny is the one introduced in [SVV18]. The main goal of that paper was to shed light on the possibility to program with quantum control at a higher level than the one of circuits. This was partially achieved: the authors have indeed brought forward a novel syntax that handles reversibility through pattern-matching, and where reversible quantum effects can be added. Nevertheless, the denotational semantics they provide is not satisfactory, as it is not compositional. Without compositionality, there is no certainty that, for instance, substitution preserves the interpretation (see the previous chapter for a compositional denotation semantics for the so-called language with quantum effects).

Our denotational semantics of the classical, reversible language involves categorical enrichment. However new in the development of reversible programming languages, this technique has been used in several other instances [Fio96, RS18, LMZ18, PPRZ20, LMZ21, HLMS23].

Other kinds of techniques can be used for the denotation of reversibility, as compact closed categories [CS21], or traced monoidal categories in the sense of [Kar19].

4.1.2 Contribution

In this chapter, we aim to close the gap: we propose a Turing-complete, reversible language, together with a categorical semantics. In particular, the contributions of this paper are as follows.

- A Turing-complete, higher-order reversible language with inductive types – this language is described as *classical*, as opposed to a *quantum* programming language. Building on the Theseus-based family of languages studied in [SVV18, CLV21, CSV23, Cha23], we consider an extension with *inductive types*, general *recursion* and *higher-order* functions. Note that the Turing-completeness has been the work of Kostia Chardonnet, and will not be presented in details here. See the paper [CLV23].
- A sound and adequate categorical semantics. We show how the language can be interpreted in join inverse rig categories. The result relies on the $\mathbf{DCPO}$-enrichments of join inverse rig categories. This part of the chapter is entirely the author's work.
- A full completeness result for *computable* functions. We finally discuss how the interpretation of the language in the category $\mathbf{PInj}$ is fully complete in the sense that any computable, partial injective set-function on the images of types is realisable within the language. This part was produced in collaboration with Kostia Chardonnet.

4.1.3 Work of the Author

The author of this thesis has contributed to the following points.

- A generalisation of the syntax presented in [SVV18] with a call-by-name λ-calculus on top of functions – called *isos* in this chapter. The operational semantics has been updated with regard to these changes. The author has also provided the corresponding proofs of the usual substitution lemma, subject reduction and progress for this system.
- A denotational semantics of the language in join inverse rig $\mathbf{DCPO}$-categories.

- A proof that the denotational semantics is adequate with regard to the operational semantics.
- A statement – akin to a completeness result – saying that any computable injection is captured by a function in the language.

4.2 The Language: Classical Symmetric Pattern-Matching

In this section, we present a reversible language, unifying and extending the Theseus-based variants presented in the literature [SVV18, CLV21, CSV23]. In particular, the language we propose features higher-order (unlike [CLV21]), pairing, injection, inductive types (unlike [SVV18]) and general recursion (unlike [CSV23]). Functions in the language are based on pattern-matching, following a strict syntactic discipline: term variables in patterns should be used linearly, and clauses should be non-overlapping on the left *and* on the right (therefore enforcing non-ambiguity and injectivity). In [SVV18, CLV21, CSV23] one also requires exhaustivity for totality. In this paper, we drop this condition in order to allow non-terminating behaviour.

$$
\begin{array}{lll}
\text{(Base types)} & A, B ::= \mathtt{I} \mid A \oplus B \mid A \otimes B \mid \mu X.A \mid X \\
\text{(Isos)} & T ::= A \leftrightarrow B \mid T_1 \to T_2 \\
\\
\text{(Values)} & v ::= * \mid x \mid \mathtt{inj}_l\, v \mid \mathtt{inj}_r\, v \mid v_1 \otimes v_2 \mid \mathtt{fold}\, v \\
\text{(Patterns)} & p ::= x \mid p_1 \otimes p_2 \\
\text{(Expressions)} & e ::= v \mid \mathtt{let}\, p_1 = \omega\ p_2\ \mathtt{in}\ e \\
\text{(Isos)} & \omega ::= \{ \mid v_1 \leftrightarrow e_1 \mid \ldots \mid v_n \leftrightarrow e_n \} \mid \mathtt{fix}\ \phi.\omega \\
 & \qquad \mid \lambda\psi.\omega \mid \phi \mid \omega_1\ \omega_2 \\
\text{(Terms)} & t ::= * \mid x \mid \mathtt{inj}_l\, t \mid \mathtt{inj}_r\, t \mid t_1 \otimes t_2 \\
 & \qquad \mid \mathtt{fold}\, t \mid \omega\ t \mid \mathtt{let}\, p = t_1\ \mathtt{in}\ t_2
\end{array}
$$

Figure 4.1 – Terms and types.

The language is presented in Figure 4.1. It consists of two layers.

- Base types: The base types consist of the unit type $\mathtt{I}$ along with its sole constructor $*$, coproduct $A \oplus B$ and tensor product $A \otimes B$ with their respective constructors, $\mathtt{inj}_l\ (t), \mathtt{inj}_r\ (t)$ and $t_1 \otimes t_2$. Finally, the language features inductive types of the form $\mu X.A$ where X is a type variable occurring in A and μ is its binder. Its associated constructor is $\mathtt{fold}\ (t)$. The inductive type $\mu X.A$ can then be unfolded into $A[\mu X.A/X]$, *i.e.*, substituting each occurrence of X by $\mu X.A$ in A. Typical examples of inductive types that can be encoded this way are the natural number, as $\mathtt{nat} = \mu X.(\mathtt{I} \oplus X)$ or the lists of types A, noted $[A] = \mu X.\mathtt{I} \oplus (A \otimes X)$. Note that we only work with closed types. We shall denote term-variables with x, y, z.

- Isos types: The language features isos, denoted ω, higher order reversible functions whose types T consist either of a pair of base type, noted $A \leftrightarrow B$ or function types between isos, $T_1 \to T_2$. A first-order iso of type $A \leftrightarrow B$ consists of a finite set of *clauses*, written $v \leftrightarrow e$ where v is a value of type A and e an expression of type B. An expression consists of a succession of applications of isos to some argument, described by `let` constructions: $\texttt{let}\ (x_1, \ldots, x_n) = \omega\ (y_1, \ldots, y_n)\ \texttt{in}\ e$. Isos can take other, first-order isos as arguments through the $\lambda\phi.\omega$ construction. Finally, isos can also represent *recursive computation* through the $\texttt{fix}\ \phi.\omega$ construction, where ϕ is an *iso-variable*.

Remark 4.1. What was called a unitary in the previous chapter is called an iso. Note that the term *iso* is the vocabulary in the original paper [SVV18]. The use of the word *unitary* before now was an emphasis on the quantum aspect of the language. The language described in this chapter being classical, we choose to use the orginal terminology.

Formation rules. While [CSV23], [SVV18] and the previous chapter require isos to be exhaustive (*i.e.* to cover all the possible values of their input types) and non-overlapping (*i.e.* two clauses cannot match a same value), we relax the exhaustivity requirement in this paper, in the spirit of what was done in [CLV21]. However, the syntax still depends on a form of orthogonality between values (resp. expressions) to define isos and to ensure pattern-matching.

Remark 4.2. The definition below mentions terms in the fashion of being as general as possible; but the notion of orthogonality is only used within values and expressions, because we do not have linear combinations of terms anymore.

Definition 4.3 (Orthogonality)**.** We introduce a binary relation $\perp$ on terms. Given two terms t_1, t_2, $t_1 \perp t_2$ holds if it can be derived inductively with the rules below; we say that t_1 and t_2 are orthogonal. The relation $\perp$ is defined as the smallest relation such that:

$$\frac{}{\texttt{inj}_l\ t_1 \perp \texttt{inj}_r\ t_2} \qquad \frac{}{\texttt{inj}_r\ t_1 \perp \texttt{inj}_l\ t_2} \qquad \frac{t_1 \perp t_2}{C_\perp[t_1] \perp C_\perp[t_2]}$$

with

$$C_\perp[-] ::= \quad - \mid \texttt{inj}_l\ C_\perp[-] \mid \texttt{inj}_r\ C_\perp[-] \mid C_\perp[-] \otimes t \mid t \otimes C_\perp[-] \mid \texttt{fold}\ C_\perp[-] \mid \texttt{let}\ p = t\ \texttt{in}\ C_\perp[-].$$

The typing rules are then given in Figure 4.2. While the rules to form terms do not come as a surprise, note the addition of a context Ψ which represents the iso-variables. The latter is non-linear, in the sense that given a well-typed term $\Psi; \Delta \vdash t : A$, a variable ϕ in Ψ can appear once, several times, or can also not appear in t. The context Δ is linear, in the same way as in the last chapter: a variable x in Δ is present exactly once in t. Note that the rule for applying an iso to a term $\omega\ t$ requires ω to have an iso ground type $A \leftrightarrow B$; this means in particular that the term $(\lambda\phi.\omega)\ t$ cannot be well-typed.

An iso abstraction $\{ \mid v_i \leftrightarrow e_i \}_{i \in I}$ is well-typed iff all the v_i and e_i are well-typed and the v_i (resp. the e_i) are pairwise orthogonal. This is necessary to ensure both forward and backward determinism. As mentioned above, there is no request for exhaustivity, and for example the *empty* iso $\{\cdot\}$ is well-typed at all ground iso types. At the same level of isos, we have a simply-typed λ-calculus as detailed in §1.1.1, this time without the product type.

$$\frac{}{\Psi;\emptyset \vdash *\colon \mathtt{I}} \qquad \frac{}{\Psi; x\colon A \vdash x\colon A}$$

$$\frac{\Psi;\Delta \vdash t\colon A}{\Psi;\Delta \vdash \mathtt{inj}_l\, t\colon A \oplus B} \qquad \frac{\Psi;\Delta \vdash t\colon B}{\Psi;\Delta \vdash \mathtt{inj}_r\, t\colon A \oplus B}$$

$$\frac{\Psi;\Delta_1 \vdash t_1\colon A \quad \Psi;\Delta_2 \vdash t_2\colon B}{\Psi;\Delta_1,\Delta_2;\vdash t_1 \otimes t_2\colon A \otimes B} \qquad \frac{\Psi;\Delta \vdash t\colon A[\mu X.A/X]}{\Psi;\Delta \vdash \mathtt{fold}\, t\colon \mu X.A}$$

$$\frac{\Psi \vdash_\omega \omega\colon A \leftrightarrow B \quad \Psi;\Delta \vdash t\colon A}{\Psi;\Delta \vdash \omega\ t\colon B}$$

$$\frac{\Psi;\Delta_1 \vdash t_1\colon A_1 \otimes \cdots \otimes A_n \qquad \Psi;\Delta_2, x_1\colon A_1, \ldots, x_n\colon A_n \vdash t_2\colon B}{\Psi;\Delta_1,\Delta_2 \vdash \mathtt{let}\, x_1 \otimes \cdots \otimes x_n = t_1\ \mathtt{in}\ t_2\colon B}$$

$$\frac{\begin{array}{cccc}\Psi;\Delta_1 \vdash v_1\colon A & \ldots & \Psi;\Delta_n \vdash v_n\colon A & \forall i \neq j, v_i \perp v_j \\ \Psi;\Delta_1 \vdash e_1\colon B & \ldots & \Psi;\Delta_n \vdash_e e_n\colon B & \forall i \neq j, e_i \perp e_j\end{array}}{\Psi \vdash_\omega \{\ |\ \ v_1 \leftrightarrow e_1 \mid \ldots \mid\ \ v_n \leftrightarrow e_n\ \}\colon A \leftrightarrow B.}$$

$$\frac{}{\Psi, \phi\colon T \vdash_\omega \phi\colon T} \qquad \frac{\Psi, \phi\colon T \vdash_\omega \omega\colon T}{\Psi \vdash_\omega \mathtt{fix}\ \phi.\omega\colon T} \qquad \frac{\Psi, \phi\colon T_1 \vdash_\omega \omega\colon T_2}{\Psi \vdash_\omega \lambda\phi.\omega\colon T_1 \to T_2}$$

$$\frac{\Psi \vdash_\omega \omega_1\colon T_1 \quad \Psi \vdash_\omega \omega_2\colon T_1 \to T_2}{\Psi \vdash_\omega \omega_2\ \omega_1\colon T_2}$$

Figure 4.2 – Typing rules of terms and isos.

Remark 4.4. As a further note, we will observe that our language has a sound and adequate denotational semantics in a $\mathbf{DCPO}$-enriched category; and the interpretation of the λ-calculus of isos happens at the dcpo level. This means in particular that any language, as long as it is adequately interpreted in the cartesian closed category $\mathbf{DCPO}$, can replace the simply-typed λ-calculus of isos. It also means that any usual structure on top of a λ-calculus can be added to the current language.

Iso abstractions handle the use of several variables, thus our version of β-reduction needs to behave accordingly. To do so, we introduce the notion of valuation, akin to the one in the previous chapter; and the application of a valuation to a term performs a substitution.

Substitutions. We recall the definitions of the last chapter, adapted to the language presented here. The difference is only embodied by inductive types formalism, where the `fold` constructor replaces contructors for natural numbers. Given two values v and v', we build the smallest valuation σ such that the patterns of v and v' match and that the application of the substitution to v, written $\sigma(v)$, is equal to v'. we denote the matching of a value v' against a pattern v and its associated valuation σ as $\mathrm{match}(\sigma, v, v')$. Thus, $\mathrm{match}(\sigma, v, v')$ means that v' matches with v and gives a smallest valuation σ, while $\sigma(v)$ is the substitution performed. The predicate

$\mathrm{match}(\sigma, v, v')$ it is defined as follows.

$$\frac{}{\mathrm{match}(\sigma, *, *)} \quad \frac{\sigma = \{x \mapsto v'\}}{\mathrm{match}(\sigma, x, v')} \quad \frac{\mathrm{match}(\sigma, v, v')}{\mathrm{match}(\sigma, \mathtt{inj}_i\ v, \mathtt{inj}_i\ v')}$$

$$\frac{\mathrm{match}(\sigma, v_1, v_1') \quad \mathrm{match}(\sigma, v_2, v_2') \quad \mathsf{supp}(\sigma_1) \cap \mathsf{supp}(\sigma_2) = \emptyset \quad \sigma = \sigma_1 \cup \sigma_2}{\mathrm{match}(\sigma, v_1 \otimes v_2, v_1' \otimes v_2')}$$

$$\frac{\mathrm{match}(\sigma, v, v')}{\mathrm{match}(\sigma, \mathtt{fold}\ v, \mathtt{fold}\ v')}$$

Whenever σ is a valuation whose support contains the variables of t, we write $\sigma(t)$ for the value where the variables of t have been replaced with the corresponding terms in σ, as follows:

- $\sigma(x) = t'$ if $\{x \mapsto t'\} \subseteq \sigma$,
- $\sigma(*) = *$,
- $\sigma(\mathtt{inj}_l\ t) = \mathtt{inj}_l\ \sigma(t)$,
- $\sigma(\mathtt{inj}_r\ t) = \mathtt{inj}_r\ \sigma(t)$,
- $\sigma(t_1 \otimes t_2) = \sigma(t_1) \otimes \sigma(t_2)$,
- $\sigma(\mathtt{fold}\ t) = \mathtt{fold}\ \sigma(t)$,
- $\sigma(\omega\ t) = \omega\ \sigma(t)$,
- $\sigma(\mathtt{let}\, p = t_1\ \mathtt{in}\ t_2) = \mathtt{let}\, p = \sigma(t_1)\ \mathtt{in}\ \sigma(t_2)$.

Remark 4.5. Definition 3.25 is reminded here. Even if it involves a slightly different syntax, it still holds. A valuation σ is said to be well-formed with regard to contexts Ψ and Δ if for all $(x_i \colon A_i) \in \Delta$, we have $\{x_i \mapsto t_i\} \subseteq \sigma$ and $\Psi; \emptyset \vdash t_i \colon A_i$. We write $\Psi; \Delta \Vdash \sigma$.

Lemma 4.6. *If $\Psi; \Delta \vdash t \colon A$ and $\Psi; \Delta \Vdash \sigma$ are well-formed, then $\Psi; \emptyset \vdash \sigma(t) \colon A$ is well-formed.*

Proof. The proof is done by induction on $\Psi; \Delta \vdash t \colon A$.

- $\Psi; \emptyset \vdash * \colon \mathtt{I}$. Nothing to do.
- $\Psi; x \colon A \vdash x \colon A$. Since $x \colon A \Vdash \sigma$ is well-formed, there is $\vdash t \colon A$ such that $\{x \mapsto t\} \subseteq \sigma$ and $\sigma(x) = t$.
- $\Psi; \Delta_1, \Delta_2 \vdash t_1 \otimes t_2 \colon A \otimes B$. The induction hypothesis ensures that $\sigma(t_1)$ and $\sigma(t_2)$ are well-typed, and thus $\sigma(t_1) \otimes \sigma(t_2) = \sigma(t_1 \otimes t_2)$ also is.
- $\Psi; \Delta \vdash \mathtt{inj}_i\ t \colon A_1 \oplus A_2$. The induction hypothesis ensures that $\sigma(t)$ is well-typed, and thus $\mathtt{inj}_i\ \sigma(t) = \sigma(\mathtt{inj}_i\ t)$ is.
- $\Psi; \Delta \vdash \mathtt{fold}\ t \colon \mu X.A$. The induction hypothesis ensures that $\sigma(t)$ is well-typed, and thus $\mathtt{fold}\ \sigma(t) = \sigma(\mathtt{fold}\ t)$ is.
- $\Psi; \Delta \vdash \omega\ t \colon B$. The induction hypothesis ensures that $\sigma(t)$ is well-typed, and thus $\omega\ \sigma(t) = \sigma(\omega\ t)$ is.
- $\Psi; \Delta_1, \Delta_2 \vdash \mathtt{let}\, p = t_1\ \mathtt{in}\ t_2 \colon B$. The induction hypothesis ensures that $\sigma(t_1)$ and $\sigma(t_2)$ are well-typed, and thus $\mathtt{let}\, p = \sigma(t_1)\ \mathtt{in}\ \sigma(t_2) = \sigma(\mathtt{let}\, p = t_1\ \mathtt{in}\ t_2)$ also is.

□

Once substitutions are defined, we can make our way through the operational semantics.

Operational semantics. The language is equipped with a small-step operational semantics, that revolves around a β-reduction at the level of isos and a reversible equivalent to β-reduction

at the level of terms. First, we introduce an operational semantics for isos, similar to a call-by-name reduction strategy in a λ-calculus. It is given in Figure 4.3. The β-reduction and the congruence rule are usually for a simply-typed λ-calculus. The `fix` operator is handled operationally *à la* PFC [Plo77].

$$\frac{}{\texttt{fix}\ \phi.\omega \to \omega[\texttt{fix}\ \phi.\omega/\phi]} \qquad \frac{}{(\lambda\phi.\omega_1)\omega_2 \to \omega_1[\omega_2/\phi]} \qquad \frac{\omega_1 \to \omega_1'}{\omega_1\omega_2 \to \omega_1'\omega_2}$$

Figure 4.3 – Isos operational semantics.

As in any formal programming language given an operational semantics, progress requires a notion of value, that is defined below. Note that the iso values are the ones for closed isos.

Definition 4.7 (Iso values)**.** We call *iso values* the following isos:

$$\omega ::= \{ \mid\ v_1 \leftrightarrow e_1 \mid \ldots \mid\ v_n \leftrightarrow e_n\ \} \mid\ \lambda\phi.\omega.$$

Lemma 4.8. *If* $\Psi, \psi\colon T_1 \vdash_\omega \omega_2\colon T_2$ *and* $\Psi \vdash_\omega \omega_1\colon T_1$ *are well-formed, then* $\Psi \vdash_\omega \omega_2[\omega_1/\psi]\colon T_2$.

Proof. Formally, the inductive definition of an iso judgement also depends on term judgements, in other words we also prove the following statement: If $\Psi, \psi\colon T_1; \Delta \vdash t\colon A$ and $\Psi \vdash_\omega \omega_1\colon T_1$ are well-formed, then $\Psi; \Delta \vdash t[\omega_1/\psi]\colon A$. The proof is done by mutual induction on the term and iso judgements.

- $\Psi, \psi\colon T_1; \emptyset \vdash *\colon \mathtt{I}$. Direct.
- $\Psi, \psi\colon T_1; x\colon A \vdash x\colon A$. Direct.
- $\Psi, \psi\colon T_1; \Delta_1, \Delta_2 \vdash t_1 \otimes t_2\colon A \otimes B$. We observe that $(t_1 \otimes t_2)[\omega_1/\psi] = t_1[\omega_1/\psi] \otimes t_2[\omega_1/\psi]$ and the induction hypothesis concludes.
- $\Psi, \psi\colon T_1; \Delta \vdash \mathtt{inj}_i\, t\colon A_1 \oplus A_2$. Similar to the previous point.
- $\Psi, \psi\colon T_1; \Delta \vdash \mathtt{fold}\, t\colon \mu X.A$. Similar to the previous point.
- $\Psi, \psi\colon T_1; \Delta \vdash \omega\ t\colon B$. We observe that $(\omega\ t)[\omega_1/\psi] = \omega[\omega_1/\psi]\ t[\omega_1/\psi]$ and the induction hypothesis concludes.
- $\Psi, \psi\colon T_1; \Delta_1, \Delta_2 \vdash \mathtt{let}\, p = t_1\ \mathtt{in}\ t_2\colon B$. With the induction hypothesis, similar to the previous point.
- $\Psi, \psi\colon T_1 \vdash_\omega \{ \mid\ v_i \leftrightarrow e_i\ \}_{i\in I} : A \leftrightarrow B$. By induction hypothesis, given any iso ω present in e_i, ω is well-typed and $\omega[\omega_1/\psi]$ is also.
- $\Psi, \psi\colon T_1, \phi\colon T \vdash_\omega \phi\colon T$. Direct.
- $\Psi, \psi\colon T_1 \vdash_\omega \mathtt{fix}\ \phi.\omega\colon T$. Note that $(\mathtt{fix}\ \phi.\omega)[\omega_1/\psi] = \mathtt{fix}\ \phi.(\omega[\omega_1/\psi])$, and by induction hypothesis $\omega[\omega_1/\psi]$ is well-typed.
- $\Psi, \psi\colon T_1 \vdash_\omega \lambda\phi.\omega\colon T_2 \to T_2'$. Similar to the previous point.
- $\Psi, \psi\colon T_1 \vdash_\omega \omega'\omega\colon T_2$. Note that $(\omega'\omega)[\omega_1/\psi] = \omega'[\omega_1/\psi]\omega[\omega_1/\psi]$, and by induction hypothesis, $\omega'[\omega_1/\psi]$ and $\omega[\omega_1/\psi]$ are well-typed.

□

Lemma 4.9 (Iso Subject Reduction). *If $\Psi \vdash_\omega \omega : T$ is well-formed and $\omega \to \omega'$, then $\Psi \vdash_\omega \omega' : T$.*

Proof. The proof is done by induction on $\to$.

- $\texttt{fix}\ \phi.\omega \to \omega[\texttt{fix}\ \phi.\omega/\phi]$. The iso $\texttt{fix}\ \phi.\omega$ is well-typed, thus ω is also, and the previous lemma concludes.
- $(\lambda\phi.\omega_1)\omega_2 \to \omega_1[\omega_2/\phi]$. For the application to be well-typed, we need both $\lambda\phi.\omega_1$ and ω_2 to be well-typed. The former ensures that ω_1 is well-typed, and the previous lemma concludes.
- $\omega_1\omega_2 \to \omega_1'\omega_2$. The induction hypothesis ensures that ω_1' is well-typed, and ω_2 is also because the application $\omega_1\omega_2$ is.

□

Lemma 4.10 (Iso Progress). *If $\vdash_\omega \omega : T$ is well-formed, ω is either an iso value or there exists ω' such that $\omega \to \omega'$.*

Proof. The proof is done by induction on $\vdash_\omega \omega : T$.

- $\vdash_\omega \{ \mid\ v_i \leftrightarrow e_i \}_{i \in I} : A \leftrightarrow B$ is an iso value.
- $\vdash_\omega \texttt{fix}\ \phi.\omega : T$ reduces.
- $\vdash_\omega \lambda\phi.\omega : T_1 \to T_2$ is an iso value.
- $\vdash_\omega \omega_2\omega_1 : T_2$. By induction hypothesis, either ω_2 is a value or it reduces. If it reduces, $\omega_2\omega_1$ reduces. If it is a value, it cannot be an iso abstraction: being applied to another iso, it must have a type $T_1 \to T_2$. Thus, it is of the form $\lambda\phi.\omega_2'$, and $(\lambda\phi.\omega_2')\omega_1$ reduces.

□

Corollary 4.11. *If $\vdash_\omega \omega : A \leftrightarrow B$ is well-formed, either there is some $\Delta_i \vdash v_i : A$ and $\Delta_i \vdash e_i : B$ such that $\omega = \{ \mid\ v_1 \leftrightarrow e_1 \mid \ldots \mid\ v_n \leftrightarrow e_n \}$, or there exists ω' such that $\omega \to \omega'$.*

We move to an operational semantics for terms of the language, which requires the introduction of congruence context. The operational semantics is detailed in Figure 4.4.

$$
\begin{aligned}
C_\to[-] ::= \quad & - \mid \texttt{inj}_l\, C_\to[-] \mid \texttt{inj}_r\, C_\to[-] \mid C_\to[-] \otimes t \mid v \otimes C_\to[-] \\
& \mid \{ \mid\ v_i \leftrightarrow e_i \}_{i \in I}\ C_\to[-] \mid \texttt{fold}\, C_\to[-] \\
& \mid \texttt{let}\, p = C_\to[-]\ \texttt{in}\ t.
\end{aligned}
$$

$$
\frac{\mathrm{match}(\sigma, v_i, v')}{\{ \mid\ v_1 \leftrightarrow e_1 \mid \ldots \mid\ v_n \leftrightarrow e_n \}\ v' \to \sigma(e_i)} \qquad \frac{\mathrm{match}(\sigma, p, v)}{\texttt{let}\, p = v\ \texttt{in}\ t \to \sigma(t)}
$$

$$
\frac{t_1 \to t_2}{C_\to[t_1] \to C_\to[t_2]} \qquad \frac{\omega \to \omega'}{\omega\ t \to \omega'\ t}
$$

Figure 4.4 – Term Operational Semantics

Lemma 4.12 (Subject Reduction). *If* $\Psi; \Delta \vdash t : A$ *is well-formed and* $t \to t'$*, then* $\Psi; \Delta \vdash t' : A$ *is also well-formed.*

Proof. The proof is done by induction on $\to$. It revolves around three quick observations: Lemma 4.9, Lemma 4.6, and that if t_2 and $C_{\to}[t_1]$ are well-typed, then $C_{\to}[t_2]$ also is. □

As usual we write $\to^*$ for the reflexive transitive closure of $\to$. In particular, the rewriting system follow a *call-by-value* strategy, requiring that the argument of an iso is fully evaluated to a value before firing the substitution. Note that unlike [CSV23, SVV18], we do not require any form of termination and isos are not required to be exhaustive: the rewriting system can diverge or be stuck. The evaluation of an iso applied to a value is dealt with by pattern-matching: the input value will try to match one of the value from the clauses and potentially create a substitution if the two values match, giving the corresponding expression as an output under that substitution.

Example 4.13. Observe that $\vdash_\omega \texttt{fix}\ \phi.\phi : A \leftrightarrow B$ is well-formed judgement. Given any closed term judgement $\vdash t : A$, the judgement $\vdash (\texttt{fix}\ \phi.\phi)\ t : B$ is also well-formed, and:

$$\begin{aligned}(\texttt{fix}\ \phi.\phi)\ t &\to (\phi[\texttt{fix}\ \phi.\phi/\phi])\ t = (\texttt{fix}\ \phi.\phi)\ t \\ &\to (\phi[\texttt{fix}\ \phi.\phi/\phi])\ t = (\texttt{fix}\ \phi.\phi)\ t \\ &\to (\phi[\texttt{fix}\ \phi.\phi/\phi])\ t = (\texttt{fix}\ \phi.\phi)\ t \\ &\to \cdots\end{aligned}$$

does not terminate. A slightly more subtle instance of non-termination is the term $\vdash (\texttt{fix}\ \phi.\{|\ x \leftrightarrow \texttt{let}\, y = \phi\ x\ \texttt{in}\ y\}\ v : B$, which reduces as follows:

$$\begin{aligned}&(\texttt{fix}\ \phi.\{|\ x \leftrightarrow \texttt{let}\, y = \phi\ x\ \texttt{in}\ y\})\ v \\ &\to (\{|\ x \leftrightarrow \texttt{let}\, y = \phi\ x\ \texttt{in}\ y\}[\texttt{fix}\ \phi.\{|\ x \leftrightarrow \texttt{let}\, y = \phi\ x\ \texttt{in}\ y\}/\phi])\ v \\ &= \{|\ x \leftrightarrow \texttt{let}\, y = (\texttt{fix}\ \phi.\{|\ x \leftrightarrow \texttt{let}\, y = \phi\ x\ \texttt{in}\ y\})\ x\ \texttt{in}\ y\}\ v \\ &\to \texttt{let}\, y = (\texttt{fix}\ \phi.\{|\ x \leftrightarrow \texttt{let}\, y = \phi\ x\ \texttt{in}\ y\})\ v\ \texttt{in}\ y \\ &\to \cdots\end{aligned}$$

and it does not terminate.

Example 4.14. Reductions can get stuck, because there is no pattern to match with. For example, $\vdash_\omega \{|\ \texttt{inj}_r * \leftrightarrow \texttt{inj}_l *\} : \mathtt{I} \oplus \mathtt{I} \leftrightarrow \mathtt{I} \oplus \mathtt{I}$ is a well-typed iso abstraction. The term $\vdash \{|\ \texttt{inj}_r * \leftrightarrow \texttt{inj}_l *\}\ (\texttt{inj}_l *) : \mathtt{I} \oplus \mathtt{I}$ does not reduce.

Example 4.15. Remember that $[A] = \mu X.\mathtt{I} \oplus (A \otimes X)$. One can define the *map* operator on list with an iso of type $(A \leftrightarrow B) \to [A] \leftrightarrow [B]$, defined as

$$\lambda\psi.\texttt{fix}\ \phi. \left\{ \begin{array}{ll} [\,] & \leftrightarrow [\,] \\ h :: t & \leftrightarrow \texttt{let}\, h' = \psi\ h\ \texttt{in let}\, t' = \phi\ t\ \texttt{in}\ h' :: t' \end{array} \right\},$$

with the terms $[\,] = \texttt{fold}\ (\texttt{inj}_l\ (*))$, representing the empty list, and $h :: t = \texttt{fold}\ (\texttt{inj}_r\ (h \otimes t))$, representing the head and tail of the list. Its inverse map^{-1} is

$$\lambda\psi.\texttt{fix}\ \phi. \left\{ \begin{array}{ll} [\,] & \leftrightarrow [\,] \\ h' :: t' & \leftrightarrow \texttt{let}\, t = \phi\ t'\ \texttt{in let}\, h = \psi h'\ \texttt{in}\ h :: t \end{array} \right\}.$$

Note that in the latter, the variable ψ has type $B \leftrightarrow A$. If we consider the inverse of the term $(\mathrm{map}\ \omega)$ we would obtain the term $(\mathrm{map}^{-1}\ \omega^{-1})$ where ω^{-1} would be of type $B \leftrightarrow A$.

Example 4.16 (Cantor Pairing). One can encode the Cantor Pairing between $\mathbb{N} \otimes \mathbb{N} \leftrightarrow \mathbb{N}$. First recall that the type of natural number $\mathtt{nat}$ is given by $\mu X.\mathtt{I} \oplus X$, then define $\overline{n}$ as the encoding of natural numbers into a closed value of type $\mathtt{nat}$ as $\overline{0} = \mathtt{fold}\ (\mathtt{inj}_l\ *)$ and given a variable x of type $\mathtt{nat}$, its successor is $\overline{S(x)} = \mathtt{fold}\ (\mathtt{inj}_r\ (x))$. Omitting the $\overline{\cdot}$ operator for readability, the pairing is then defined as:

$$\begin{aligned}
&\omega_1 : \mathtt{nat} \otimes \mathtt{nat} \leftrightarrow (\mathtt{nat} \otimes \mathtt{nat}) \oplus \mathtt{I} \\
&= \left\{ \begin{array}{lcl} S(i) \otimes j & \leftrightarrow & \mathtt{inj}_l\ (i \otimes S(j)) \\ 0 \otimes S(j) & \leftrightarrow & \mathtt{inj}_l\ (j \otimes 0) \\ 0 \otimes 0 & \leftrightarrow & \mathtt{inj}_r\ (*) \end{array} \right\},
\end{aligned}$$

$$\begin{aligned}
&\omega_2 : (\mathtt{nat} \otimes \mathtt{nat}) \oplus \mathtt{I} \leftrightarrow \mathtt{nat} \\
&= \left\{ \begin{array}{lcl} \mathtt{inj}_l\ (x) & \leftrightarrow & \mathtt{let}\, y = \phi\ x\ \mathtt{in}\ S(y) \\ \mathtt{inj}_r\ (*) & \leftrightarrow & 0 \end{array} \right\},
\end{aligned}$$

$$\begin{aligned}
&\mathrm{CantorPairing} : \mathtt{nat} \otimes \mathtt{nat} \leftrightarrow \mathtt{nat} \\
&= \mathtt{fix}\ \phi. \left\{ \begin{array}{lcl} x & \leftrightarrow & \mathtt{let}\, y = \omega_1\ x\ \mathtt{in} \\ & & \mathtt{let}\, z = \omega_2\ y\ \mathtt{in}\ z \end{array} \right\},
\end{aligned}$$

where the variable ϕ in ω_2 is the one being binded by the $\mathtt{fix}$ of the $\mathrm{CantorPairing}$ iso. Intuitively, ω_1 realise one step of the Cantor Pairing evaluation, while ω_2 check if we reached the end of the computation and either apply a recursive call, or stop.

For instance, $\mathrm{CantorPairing}\ \ 1 \otimes 1$ will match with the first clause of ω_1, evaluating into $\mathtt{inj}_l\, 0 \otimes 2$, and then, inside ω_2 the reduction $\mathrm{CantorPairing}\ \ 0 \otimes 2$ will be triggered through the recursive call, evaluating the second clause of ω_1, reducing to $\mathtt{inj}_l\, 1 \otimes 0$, etc.

4.3 Denotational semantics

We now show how to build a denotational semantics for the language we presented thus far. The semantics is akin to the one presented in [CLV21] but with extra structure to handle inductive types and recursive functions. The section is organised as follows.

- We first fix the interpretation of types. This requires us to make type judgements explicit in the formalisation of the syntax. The semantics of type judgement is then given, thanks to the work mentioned in §1.2.2. We then discuss the interpretation of closed types, which are the types actually involved in the syntax.
- We detail the interpretation of term judgements, that are given as Scott continuous maps between the interpretation of the linear context, and the dcpo of reversible programs at a certain type.
- An interpretation of iso judgements is given, also as Scott continuous maps. The development is similar to the denotational semantics of a simply-typed λ-calculus as given in §1.1.1, however our iso abstractions need more care.
- We finish with an interpretation of substitutions, allowing to fromulate a soundness and adequacy statement later on.

In the whole section, we consider $\mathbf{C}$ a join inverse rig category (see Definition 1.78), that is $\mathbf{DCPO}$-enriched (see Definition 1.32 and §1.2.1) and such that 0 and 1 are distinct objects.

4.3.1 Denotational Semantics of Types

Term types. As explained in the background section (see §1.3), we can assume without loss of generality that $\mathbf{C}$ satisfies the hypothesis of Theorem 1.50. In order to deal with open types, we make use of an auxiliary judgement for types, of the form $X_1, \dots, X_n \vDash A$, where $\{X_i\}_i$ is a subset of the free type variables, non necessarily appearing in A. We interpret this kind of judgement as a $\mathbf{DCPO}$-functor $\mathbf{C}^{|\Theta|} \to \mathbf{C}$ written $[\![\Theta \vDash A]\!]$. This is formally defined as an inductive relation, and the semantics is stated similarly to what is done in [Fio04, LMZ21, JLMZ21a].

$$\frac{}{\Theta, X \vDash X} \qquad \frac{}{\Theta \vDash \mathtt{I}} \qquad \frac{\Theta \vDash A \quad \Theta \vDash B}{\Theta \vDash A \star B}\ \star \in \{\oplus, \otimes\} \qquad \frac{\Theta, X \vDash A}{\Theta \vDash \mu X.A} \tag{4.1}$$

The type judgements are inductively defined as given above. The interpretation of types is detailed in Figure 4.5, where $\Pi \colon \mathbf{C}^{|\Theta|} \to \mathbf{C}$ is the projection functor on the last component, $K_1 \colon \mathbf{C}^{|\Theta|} \to \mathbf{C}$ is the constant functor that outputs 1 and the id, $\otimes$ and $\oplus$ are given by the rig structure, and $(-)^\dagger$ is part of the parameterised initial algebra (see Definition 1.45), that exists thanks to Theorem 1.50 (see details in §1.2.2).

$$\begin{aligned}
[\![\Theta \vDash A]\!] &\colon \mathbf{C}^{|\Theta|} \to \mathbf{C} \\
[\![\Theta, X \vDash X]\!] &= \Pi \\
[\![\Theta \vDash \mathtt{I}]\!] &= K_1 \\
[\![\Theta \vDash A \oplus B]\!] &= \oplus \circ \langle [\![\Theta \vDash A]\!], [\![\Theta \vDash B]\!] \rangle \\
[\![\Theta \vDash A \otimes B]\!] &= \otimes \circ \langle [\![\Theta \vDash A]\!], [\![\Theta \vDash B]\!] \rangle \\
[\![\Theta \vDash \mu X.A]\!] &= ([\![\Theta, X \vDash A]\!])^\dagger
\end{aligned}$$

Figure 4.5 – Interpretation of types.

We remind the typing rule of the term contructor `fold`.

$$\frac{\Psi; \Delta \vdash t \colon A[\mu X.A/X]}{\Psi; \Delta \vdash \mathtt{fold}\, t \colon \mu X.A}$$

This typing rule involves a type substitution. We show in the next lemma that type substitutions make sense, and we also provide a result about there interpretation.

Lemma 4.17 (Type Substitution). *Given well-formed type judgements $\Theta, X \vDash A$ and $\Theta \vDash B$, the judgement $\Theta \vDash A[B/X]$ is well-formed and*

$$[\![\Theta \vDash A[B/X]]\!] = [\![\Theta, X \vDash A]\!] \circ \langle \mathrm{id}, [\![\Theta \vDash B]\!] \rangle.$$

Proof. The proof that $\Theta \vDash A[B/X]$ is well-formed is direct by induction on the formation rules. The semantic equality is also proven by induction on the formation rules of $\Theta, X \vDash A$.

- $\Theta \vDash \mathtt{I}$. Nothing to do.
- $\Theta, X \vDash X$. Indeed, $[\![\Theta \vDash B]\!] = \Pi \circ \langle \mathrm{id}, [\![\Theta \vDash B]\!] \rangle$.
- $\Theta, X \vDash A_1 \star A_2$.

$$
\begin{aligned}
&[\![\Theta \vDash (A_1 \star A_2)[B/X]]\!] \\
&= [\![\Theta \vDash A_1[B/X] \star A_2[B/X]]\!] \\
&= \star \circ \langle [\![\Theta \vDash A_1[B/X]]\!], [\![\Theta \vDash A_2[B/X]]\!] \rangle && \text{(by definition)} \\
&= \star \circ \langle [\![\Theta, X \vDash A_1]\!] \circ \langle \mathrm{id}, [\![\Theta \vDash B]\!] \rangle, [\![\Theta, X \vDash A_2]\!] \circ \langle \mathrm{id}, [\![\Theta \vDash B]\!] \rangle \rangle && \text{(by IH)} \\
&= \star \circ \langle [\![\Theta, X \vDash A_1]\!], [\![\Theta, X \vDash A_1]\!] \rangle \circ \langle \mathrm{id}, [\![\Theta \vDash B]\!] \rangle && \text{(by unicity)} \\
&= [\![\Theta, X \vDash A_1 \star A_2]\!] \circ \langle \mathrm{id}, [\![\Theta \vDash B]\!] \rangle && \text{(by definition)}
\end{aligned}
$$

- $\Theta, X \vDash \mu Y.A$.

$$
\begin{aligned}
&[\![\Theta \vDash (\mu Y.A)[B/X]]\!] \\
&= [\![\Theta \vDash \mu Y.A[B/X]]\!] \\
&= ([\![\Theta, Y \vDash A[B/X]]\!])^\dagger && \text{(by definition)} \\
&= ([\![\Theta, Y, X \vDash A]\!] \circ \langle \mathrm{id}, [\![\Theta, Y \vDash B]\!] \rangle)^\dagger && \text{(by IH)} \\
&= ([\![\Theta, X, Y \vDash A]\!] \circ (\langle \mathrm{id}, [\![\Theta \vDash B]\!] \rangle \times \mathrm{id}))^\dagger && (Y \text{ is not in } B) \\
&= ([\![\Theta, X, Y \vDash A]\!])^\dagger \circ \langle \mathrm{id}, [\![\Theta \vDash B]\!] \rangle && \text{(see [LMZ21, Prop. 4.14])} \\
&= [\![\Theta, X \vDash \mu Y.A]\!] \circ \langle \mathrm{id}, [\![\Theta \vDash B]\!] \rangle && \text{(by definition)}
\end{aligned}
$$

□

The previous lemma embodies the link between the fixed point contructor μ and the parameterised initial algebra, with the following observation:

$$[\![\Theta \vDash A[\mu X.A/X]]\!] = [\![\Theta, X \vDash A]\!] \circ \langle \mathrm{id}, [\![\Theta \vDash B]\!] \rangle \hat{\Rightarrow} [\![\Theta \vDash \mu X.A]\!]$$

and thus there is a natural isomorphism:

$$\alpha^{[\![\Theta \vDash A]\!]} \colon [\![\Theta, X \vDash A[\mu X.A/X]]\!] \cong [\![\Theta \vDash \mu X.A]\!] .$$

In the syntax of this chapter, only closed types are involved in the typing rules of terms. We sum up the semantics of closed types below.

$$
\begin{gathered}
[\![\mathtt{I}]\!] = 1 \qquad [\![A \oplus B]\!] = [\![A]\!] \oplus [\![B]\!] \qquad [\![A \otimes B]\!] = [\![A]\!] \otimes [\![B]\!] \\
[\![\mu X.A]\!] \cong [\![A[\mu X.A/X]]\!]
\end{gathered}
\tag{4.2}
$$

Iso types. The basic types of isos are represented by pointed dcpos of morphisms in $\mathbf{C}$, written $[\![A \leftrightarrow B]\!] = \mathbf{C}([\![A]\!], [\![B]\!])$. The rest is given by the usual type interpretation of a simply-typed λ-calculus in the cartesian closed category $\mathbf{DCPO}$ (see §1.1.1 for the details).

The terms used to build isos are dependent in two contexts: variables in Δ and isos in Ψ. In general, if $\Delta = x_1 \colon A_1, \ldots, x_m \colon A_m$ and $\Psi = \phi_1 \colon B_1 \leftrightarrow C_1, \ldots, \phi_n \colon B_n \leftrightarrow C_n$, then we set $[\![\Delta]\!] = [\![A_1]\!] \otimes \cdots \otimes [\![A_m]\!]$ and $[\![\Psi]\!] = \mathbf{C}([\![B_1]\!], [\![C_1]\!]) \times \cdots \times \mathbf{C}([\![B_n]\!], [\![C_n]\!])$, with $\otimes$ being the monoidal product in $\mathbf{C}$ and $\times$ the cartesian product in $\mathbf{DCPO}$.

4.3.2 Denotational Semantics of Terms

A well-formed term judgement $\Psi; \Delta \vdash t \colon A$ is given a semantics

$$[\![\Psi; \Delta \vdash t \colon A]\!] : [\![\Psi]\!] \to \mathbf{C}([\![\Delta]\!], [\![A]\!])$$

as a Scott continuous map between two dcpos – in other words, as a morphism in $\mathbf{DCPO}$. Values do not contain iso variables, thus given a judgement $\Psi; \Delta \vdash v \colon A$ with v a value, $[\![\Psi; \Delta \vdash v \colon A]\!]$ is a constant function, whose output is a morphism $[\![\Delta]\!] \to [\![A]\!]$ in $\mathbf{C}$. The interpretation of values, and therefore of the corresponding terms, is as follows, for all $g \in [\![\Psi]\!]$:

$$\begin{aligned}
[\![\Psi; \Delta \vdash t \colon A]\!](g) &\in \mathbf{C}([\![\Delta]\!], [\![A]\!]) \\
[\![\Psi; \emptyset \vdash * \colon \mathtt{I}]\!](g) &= \mathrm{id}_{[\![\mathtt{I}]\!]} \\
[\![\Psi; x \colon A \vdash x \colon A]\!](g) &= \mathrm{id}_{[\![A]\!]} \\
[\![\Psi; \Delta \vdash \mathtt{inj}_l\, t \colon A \oplus B]\!](g) &= \iota_l \circ [\![\Psi; \Delta \vdash t \colon A]\!](g) \\
[\![\Psi; \Delta \vdash \mathtt{inj}_r\, t \colon A \oplus B]\!](g) &= \iota_r \circ [\![\Psi; \Delta \vdash t \colon B]\!](g) \\
[\![\Psi; \Delta_1, \Delta_2 \vdash t_1 \otimes t_2 \colon A \otimes B]\!](g) &= [\![\Psi; \Delta_1 \vdash t_1 \colon A]\!](g) \otimes [\![\Psi; \Delta_2 \vdash t_2 \colon B]\!](g) \\
[\![\Psi; \Delta \vdash \mathtt{fold}\, t \colon \mu X.A]\!](g) &= \alpha^{[\![X \vDash A]\!]} \circ [\![\Psi; \Delta \vdash t \colon A[\mu X.A/X]]\!](g)
\end{aligned}$$

Lemma 4.18. *Given two judgements $\Psi; \Delta_1 \vdash v_1 \colon A$ and $\Psi; \Delta_2 \vdash v_2 \colon A$, such that $v_1 \perp v_2$, we have that for all $g \in [\![\Psi]\!]$:*

$$[\![v_1]\!](g)^\circ \circ [\![v_2]\!](g) = 0_{[\![\Delta_2]\!], [\![\Delta_1]\!]}.$$

Proof. This is proven by induction on the definition of $\perp$. The cases $\mathtt{inj}_l\, v_1 \perp \mathtt{inj}_r\, v_2$ and $\mathtt{inj}_r\, v_1 \perp \mathtt{inj}_l\, v_2$ are covered by Lemma 1.76. the other cases involve precompositions and tensor products, the result is direct with the induction hypothesis. □

Once we have fixed the denotation of the easiest terms, we can cover the difficult part. Throughout the rest of the section, the semantics of a well-formed term judgement $[\![\Psi; \Delta \vdash t \colon A]\!]$ is obviously a map between sets, and the interesting part is proving that it is indeed a Scott continuous map between two dcpos. The interpretation of the remaining terms is given below.

$$\begin{aligned}
[\![\Psi; \Delta \vdash t \colon A]\!] &\in \mathbf{DCPO}([\![\Psi]\!], \mathbf{C}([\![\Delta]\!], [\![A]\!])) \\
[\![\Psi; \Delta_1, \Delta_2 \vdash \mathtt{let}\, p = t_1\ \mathtt{in}\ t_2 \colon B]\!] &= \mathrm{comp} \circ \\
&\qquad \langle [\![\Psi; \Delta_2, p \colon A \vdash t_2 \colon B]\!], (\otimes \circ \langle \mathrm{id}_{[\![\Delta_2]\!]}, [\![\Psi; \Delta_1 \vdash t_1 \colon A]\!] \rangle) \rangle \\
[\![\Psi; \Delta \vdash \omega\ t \colon B]\!] &= \mathrm{comp} \circ \langle [\![\Psi \vdash_\omega \omega \colon A \leftrightarrow B]\!], [\![\Psi; \Delta \vdash t \colon A]\!] \rangle
\end{aligned}$$

All this is well-defined in $\mathbf{DCPO}$ provided that $[\![\Psi \vdash_\omega \omega \colon A \leftrightarrow B]\!]$ is. This last point is the focus of the next subsection. Note that the interpretation on terms and iso is thus defined by mutual induction on the term and iso judgements. This does not cause any difficulty.

Before moving to the denotational semantics of isos, we prove a lemma of central importance, extending Lemma 4.18, and showing that the interpretations of two orthogonal expressions are also orthogonal, in the sense of Remark 1.77.

Lemma 4.19. *Given two judgements $\Psi;\Delta_1 \vdash e_1 \colon A$ and $\Psi;\Delta_2 \vdash e_2 \colon A$, such that $e_1 \perp e_2$, we have that for all $g \in [\![\Psi]\!]$:*

$$[\![e_1]\!]\,(g)^\circ \circ [\![e_2]\!]\,(g) = 0_{[\![\Delta_2]\!],[\![\Delta_1]\!]}.$$

Proof. A large part of this lemma is already proven in Lemma 4.18. It remains to observe that the interpretation of `let` involves a precomposition, thus the induction hypothesis on the definition of $\perp$ is enough to conclude. □

4.3.3 Denotational Semantics of Isos

Isos do only depend on function variables but they are innately morphisms, so their denotation will be similar to terms – a Scott continuous map:

$$[\![\Psi \vdash_\omega \omega \colon A \leftrightarrow B]\!] : [\![\Psi]\!] \to \mathbf{C}([\![A]\!], [\![B]\!]).$$

We define the denotation of an iso by induction on the typing derivation. The interpretation of an iso-variable is direct: it is the projection on the last component. The interpretations of evaluations and λ-abstractions are usual in a cartesian closed category, in our case, $\mathbf{DCPO}$ (see §1.1.1 and §1.2.1).

$$\begin{aligned}
[\![\Psi \vdash_\omega \omega \colon T]\!] &\in \mathbf{DCPO}([\![\Psi]\!], [\![T]\!]) \\
[\![\Psi, \phi \colon T \vdash_\omega \phi \colon T]\!] &= \pi_{[\![T]\!]} \\
[\![\Psi \vdash_\omega \omega_2\omega_1 \colon T_2]\!] &= \mathrm{eval} \circ \langle [\![\Psi \vdash_\omega \omega_2 \colon T_1 \to T_2]\!], [\![\Psi \vdash_\omega \omega_1 \colon T_1]\!] \rangle \\
[\![\Psi \vdash_\omega \lambda\phi.\omega \colon T_1 \to T_2]\!] &= \mathrm{curry}([\![\Psi, \phi \colon T_1 \vdash_\omega \omega \colon T_2]\!]) \\
[\![\Psi \vdash_\omega \texttt{fix}\ \phi.\omega \colon T]\!] &= \mathrm{fix}([\![\Psi, \phi \colon T \vdash_\omega \omega \colon T]\!])
\end{aligned}$$

The remaining rule, that builds an iso abstraction $\{ \mid v_i \leftrightarrow e_i \}_{i\in I}$, needs more details. The interpretation of an iso abstraction is close to the one in the previous chapter (see §3.5), in a related but different setting.

Lemma 4.20. *Given a well-typed iso abstraction $\Psi \vdash_\omega \{ \mid v_i \leftrightarrow e_i \}_{i\in I} : A \leftrightarrow B$, for all $g \in [\![\Psi]\!]$, the morphisms in $\mathbf{C}$ given by:*

$$[\![\Psi;\Delta_i \vdash e_i \colon B]\!]\,(g) \circ [\![\Psi;\Delta_i \vdash v_i \colon A]\!]\,(g)^\circ$$

with $i \in I$, are pairwise inverse compatible.

Proof. Lemma 4.19 gives us that for all $g \in [\![\Psi]\!]$ and $i \neq j \in I$:

$$\begin{aligned}
([\![e_i]\!]\,(g) \circ [\![v_i]\!]\,(g)^\circ)^\circ \circ ([\![e_j]\!]\,(g) \circ [\![v_j]\!]\,(g)^\circ) &= 0_{[\![A]\!],[\![A]\!]} \\
([\![e_i]\!]\,(g) \circ [\![v_i]\!]\,(g)^\circ) \circ ([\![e_j]\!]\,(g) \circ [\![v_j]\!]\,(g)^\circ)^\circ &= 0_{[\![B]\!],[\![B]\!]}
\end{aligned}$$

which are the hypotheses of Lemma 1.70. This is enough to ensure that for all $g \in [\![\Psi]\!]$ and $i \neq j \in I$,

$$[\![e_i]\!]\,(g) \circ [\![v_i]\!]\,(g)^\circ \asymp [\![e_j]\!]\,(g) \circ [\![v_j]\!]\,(g)^\circ$$

and this last point concludes. □

In a similar vein to the previous chapter, each clause $v_i \leftrightarrow e_i$, present in an iso abstraction, is given an interpretation $[\![e_i]\!] \circ [\![v_i]\!]^\circ$. The previous lemma shows that in the case of an iso abstraction, the interpretations of all clauses can be joined (in the sense of Definition 1.65), and thus the interpretation of the iso abstraction is the least upper bound of the interpretation of all the clauses. This least upper bound is also the one in $\mathbf{DCPO}$, as shown by the lemma below.

Lemma 4.21. *Given a dcpo Ξ, two objects X and Y of $\mathbf{C}$, a set of indices I and a family of Scott continuous maps $\xi_i \colon \Xi \to \mathbf{C}(X,Y)$ that are pairwise inverse compatible, the function given by:*

$$\begin{cases} \Xi & \to & \mathbf{C}(X,Y) \\ x & \mapsto & \bigvee_{i\in I} \xi_i(x) \end{cases}$$

is Scott continuous, and is written $\bigvee_{i\in I} \xi_i$.

Proof. The function can also be obtained as the join in the dcpo $[\Xi \to \mathbf{C}(X,Y)]$, it is therefore Scott continuous. □

The interpretation of an iso abstraction is then given by:

$$[\![\Psi \vdash_\omega \{ \mid \ v_i \leftrightarrow e_i \ \}_{i\in I} : A \leftrightarrow B]\!] = \bigvee_{i\in I} (\mathrm{comp} \circ \langle [\![\Psi; \Delta_i \vdash e_i \colon B]\!] , [\![\Psi; \Delta_i \vdash v_i \colon A]\!]^\circ \rangle)$$

Proposition 4.22. *Given a well-typed iso abstraction $\Psi \vdash_\omega \{ \mid \ v_i \leftrightarrow e_i \ \}_{i\in I} : A \leftrightarrow B$, its interpretation $[\![\Psi \vdash_\omega \{ \mid \ v_i \leftrightarrow e_i \ \}_{i\in I} : A \leftrightarrow B]\!]$ is well-defined as a Scott continuous map between the dcpos $[\![\Psi]\!]$ and $\mathbf{C}([\![A]\!], [\![B]\!])$.*

Proof. This is a conclusion of Lemmas 4.20 and 4.21. □

We complete the denotational semantics of isos with an interpretation of substitutions. It is not different to the one in a usual λ-calculus.

Lemma 4.23. *Given two well-typed isos $\Psi, \phi \colon T_2 \vdash_\omega \omega_1 \colon T_1$ and $\Psi \vdash_\omega \omega_2 \colon T_2$,*

$$[\![\Psi \vdash_\omega \omega_1[\omega_2/\phi] \colon T_1]\!] = [\![\Psi, \phi \colon T_2 \vdash_\omega \omega_1 \colon T_1]\!] \circ \langle \mathrm{id}, [\![\Psi \vdash_\omega \omega_2 \colon T_2]\!] \rangle.$$

Proof. The proof is done by induction on the typing derivation of ω_1. □

4.3.4 Denotational Semantics of Valuations and Substitution

We provide an interpretation to valuations and to their application to a term of the syntax. As expected, the result obtained is close to Proposition 3.56, that details the semantics of valuations in the last chapter.

Proposition 4.24 (Substitution lemma)**.** *Given a well-typed term $\Psi; \Delta \vdash t \colon A$ and for all $(x_i \colon A_i) \in \Delta$, a well-typed term $\Psi; \emptyset \vdash t_i \colon A_i$; if $\sigma = \{x_i \mapsto t_i\}_i$, then for all $g \in [\![\Psi]\!]$:*

$$[\![\Psi; \emptyset \vdash \sigma(t) \colon A]\!](g) = [\![\Psi; \Delta \vdash t \colon A]\!](g) \circ \left(\bigotimes_i [\![\Psi; \emptyset \vdash t_i \colon A_i]\!](g) \right).$$

We define then, for all $g \in [\![\Psi]\!]$: $[\![\sigma]\!](g) = \bigotimes_i [\![\vdash t_i \colon A_i]\!](g)$.

Proof. The proof is straightforward by induction on the typing derivation of t. □

Remark 4.25. We remind here that the interpretation of a valuation, given above, is not done in the most meticulous way. Indeed, there is no order in which the variables occur in a valuation, thus its categorical denotation is necessarily *up to permutation*, as the denotation of a context. Since $\mathbf{C}$ is a symmetric monoidal category, we argue that the results of this chapter are not impacted by this choice, and that providing an extra care to permutations would only introduce more notations and confusion.

Proposition 4.24 strengthens the observation that categorical composition is *exactly* substitution in the Curry-Howard-Lambek correspondence. In addition, in our context of category equipped with an *inverse* structure, similar to the dagger in the last chapter, we can show that the *inner product* (see Remark 1.77 to understand the intuition behind this notion) of two interpretations produces the interpretation of a valuation, provided that there is a match.

Lemma 4.26. *Given two well-typed values* $\Psi; \Delta \vdash v \colon A$ *and* $\Psi; \emptyset \vdash v' \colon A$*, and a substitution* σ*, if* $\mathrm{match}(\sigma, v, v')$ *then for all* $g \in [\![\Psi]\!]$*:*

$$[\![v]\!](g)^\circ \circ [\![v']\!](g) = [\![\sigma]\!](g).$$

Proof. The proof is straightforward by induction on $\mathrm{match}(\sigma, b, b')$. □

The lemmas above can be combined to assert a first step towards soundness: the interpretations of the left-hand side and right-hand side of the reversible β-reduction are equal.

Lemma 4.27. *Given a well-typed iso abstraction* $\vdash_\omega \{ \mid v_j \leftrightarrow e_j \}_{j \in J} : A \leftrightarrow B$ *and a well-typed value* $\vdash v' \colon A$*, if* $\mathrm{match}(\sigma, v_i, v')$*, then*

$$\left[\!\!\left[\vdash \{ \mid v_j \leftrightarrow e_j \}_{j \in J}\ v' \colon B\right]\!\!\right] = [\![\vdash \sigma(v_i) \colon B]\!].$$

Proof. First, we deduce from the assumption $\mathrm{match}(\sigma, v_i, v')$ that

- $\neg(v_i \perp v')$, and thus $[\![v_i]\!]^\circ \circ [\![v']\!] = [\![\sigma]\!]$, thanks to Lemma 4.26.
- for all $j \neq i$, $v_j \perp v'$, and thus $[\![v_j]\!]^\circ \circ [\![v']\!] = 0$, thanks to Lemma 4.19.

We can then compute the semantics, with $\omega \stackrel{\text{def}}{=} \{ \mid v_j \leftrightarrow e_j \}_{j \in J}$:

$$\begin{aligned}
&[\![\omega\ v']\!] \\
&= [\![\omega]\!] \circ [\![v']\!] && \text{(by definition)} \\
&= \left(\bigvee_j [\![v_j]\!] \circ [\![b_j]\!]^\circ \right) \circ [\![v']\!] && \text{(by definition)} \\
&= \bigvee_j [\![v_j]\!] \circ [\![b_j]\!]^\circ \circ [\![v']\!] && \text{(composition distributes over join)} \\
&= [\![v_i]\!] \circ [\![b_i]\!]^\circ \circ [\![v']\!] && \text{(Lemma 4.19)} \\
&= [\![v_i]\!] \circ [\![\sigma]\!] && \text{(Lemma 4.26)} \\
&= [\![\sigma(v_i)]\!] && \text{(Prop. 4.24)}
\end{aligned}$$

□

We have brought forward a categorical interpretation to the programming language introduced in §4.2. This interpretation makes use of the join inverse rig structure to define the iso abstraction and to perform a pattern-matching that ensures reversibility; an enrichment in $\mathbf{DCPO}$ allows to consider recursive isos and inductive data types are represented with the help or parameterised initial algebras.

Independently of how convincing this model is, it is good practice to prove it has a strong link with the operational semantics of the language. This link is called *adequacy*, and is the focus of the next section.

4.4 Adequacy

We show a strong relation between the operational semantics and the denotational semantics which were introduced in the previous sections. First, we fix a mathematical interpretation $[\![-]\!]$ in a join inverse rig category $\mathbf{C}$, that is $\mathbf{DCPO}$-enriched and whose objects 0 and 1 are distinct.

Since the language handles non-termination, our adequacy statement links the denotational semantics to the notion of termination in the operational semantics.

Definition 4.28 (Terminating)**.** Given $\vdash t : A$, t is said *terminating* if there exists a value v such that $t \to^* v$. We either write $t \downarrow$, or $t \downarrow v$.

Since the system is deterministic, if $t \downarrow v$, then v is unique.

The goal of this section is to prove the next theorem.

Theorem 4.29 (Adequacy)**.** *Given* $\vdash t : A$, $t \downarrow$ *iff* $[\![\vdash t : A]\!] \neq 0_{1,[\![A]\!]}$.

Interestingly enough, there are two ways for a term $\vdash t : A$ to have $0_{1,[\![A]\!]}$ as its interpretation: either it reduces over and over in an infinite loop (see Example 4.13), or it is stuck because of pattern-matching (see Example 4.14).

One strategy could be the use of formal approximation relations, introduced by Plotkin [Plo85]. However, because of our two seperate levels of abstraction (a fixed point calculus at the level of isos, and inductive types at the level of terms), the definition of the relations and proving that they exist would be long and of little interest to the reader. We rather choose a syntactic approach, inspired by the proof in [PSV14].

4.4.1 Soundness

We start by showing the simple implication in Theorem 4.29, that the denotational semantics is stable w.r.t. computation; in other words, applying a rule of the operational semantics does not change the mathematical interpretation. The other direction – which is made formal later – is called *adequacy*, and is usually harder to prove. See §4.4.2 for full details on the adequacy result.

Lemma 4.30. *Given a well-formed iso judgement* $\vdash_\omega \omega \colon T$*, if* $\omega \to \omega'$*, then*

$$[\![\vdash_\omega \omega \colon T]\!] = [\![\vdash_\omega \omega' \colon T]\!] .$$

Proof. The proof is done by induction on $\to$.

- $\texttt{fix}\ \phi.\omega \to \omega[\texttt{fix}\ \phi.\omega/\phi]$.

$$\begin{aligned}
&[\![\texttt{fix}\ \phi.\omega]\!] \\
&= \mathrm{fix}\,[\![\omega]\!] && \text{(by definition)} \\
&= [\![\omega]\!] \circ \mathrm{fix}([\![\omega]\!]) && \text{(fixed point)} \\
&= [\![\omega]\!] \circ [\![\texttt{fix}\ \phi.\omega]\!] && \text{(by definition)} \\
&= [\![\omega[\texttt{fix}\ \phi.\omega/\phi]]\!] && \text{(Lem. 4.23)}
\end{aligned}$$

- $(\lambda\phi.\omega_1)\omega_2 \to \omega_1[\omega_2/\phi]$.

$$\begin{aligned}
&[\![(\lambda\phi.\omega_1)\omega_2]\!] \\
&= \mathrm{eval} \circ \langle \mathrm{curry}([\![\omega_1]\!]), [\![\omega_2]\!] \rangle && \text{(by definition)} \\
&= [\![\omega_1]\!] \circ [\![\omega_2]\!] && \text{(§1.1.1)} \\
&= [\![\omega_1[\omega_2/\phi]]\!] && \text{(Lem. 4.23)}
\end{aligned}$$

- $\omega_1\omega_2 \to \omega_1'\omega_2$. Direct with the induction hypothesis.

□

Proposition 4.31 (Soundness). *Given a well-formed term judgement* $\vdash t \colon A$*, if* $t \to t'$*, then*

$$[\![\vdash t \colon A]\!] = [\![\vdash t' \colon A]\!] .$$

Proof. The proof is done by induction on $\to$.

- $\{\ |\ v_i \leftrightarrow e_i\ \}_{i\in I}\, v' \to \sigma(v_j)$. This is covered by Lemma 4.27.
- $\texttt{let}\, p = v\ \texttt{in}\ t \to \sigma(t)$. This is a conclusion of Prop. 4.24.
- $\omega\ t \to \omega'\ t$. We conclude with the previous lemma (Lemma 4.30).
- $\texttt{inj}_i\ t \to \texttt{inj}_i\ t'$ when $t \to t'$. The induction hypothesis gives $[\![t]\!] = [\![t']\!]$ and then

$$[\![\texttt{inj}_i\ t]\!] = \iota_i\,[\![t]\!] = \iota_i\,[\![t']\!] = [\![\texttt{inj}_i\ t']\!] .$$

- $t \otimes t_2 \to t' \otimes t_2$ when $t \to t'$.

$$[\![t \otimes t_2]\!] = [\![t]\!] \otimes [\![t_2]\!] \overset{\text{IH}}{=} [\![t']\!] \otimes [\![t_2]\!] = [\![t' \otimes t_2]\!] .$$

- $\{\ |\ v_i \leftrightarrow e_i\ \}_{i\in I}\, t \to \{\ |\ v_i \leftrightarrow e_i\ \}_{i\in I}\, t'$ when $t \to t'$. In general,

$$[\![\omega\ t]\!] = [\![\omega]\!]\,[\![t]\!] \overset{\text{IH}}{=} [\![\omega]\!]\,[\![t']\!] = [\![\omega\ t']\!] .$$

- $\texttt{fold}\ t \to \texttt{fold}\ t'$ when $t \to t'$.

$$[\![\texttt{fold}\ t]\!] = \alpha\,[\![t]\!] \overset{\text{IH}}{=} \alpha\,[\![t']\!] = [\![\texttt{inj}_i\ t']\!] .$$

- $\texttt{let}\, p = t\ \texttt{in}\ t_2 \to \texttt{let}\, p = t'\ \texttt{in}\ t_2$ when $t \to t'$.

$$[\![\texttt{let}\, p = t\ \texttt{in}\ t_2]\!] = [\![t_2]\!]\,(\mathrm{id} \otimes [\![t]\!]) \overset{\mathsf{IH}}{=} [\![t_2]\!]\,(\mathrm{id} \otimes [\![t']\!]) = [\![\texttt{let}\, p = t'\ \texttt{in}\ t_2]\!]\,.$$

□

Corollary 4.32. *Given a well-formed term judgement $\vdash t \colon A$ such that $t \downarrow$, then*

$$[\![\vdash t \colon A]\!] \neq 0_{[\![A]\!]}.$$

Proof. Knowing that there is a value v such that $t \to^* v$, Prop. 4.31 ensures that

$$[\![t]\!] = [\![v]\!] \neq 0.$$

□

However, this was only the simple direction of the main theorem. The proof of the other implication is the focus of the next section.

4.4.2 Proof of Adequacy

Our proof of adequacy involves a finitary sublanguage, where the number of recursive calls is controlled syntactically. We show the adequacy result for the finitary terms thanks to strong normalisation, and then show that it implies adequacy for the whole language; this is achieved by observing that a normalising finitary term is also normalising in its non-finitary form.

Finitary sublanguage. We introduce the syntax for finitary terms, where the number of possible reductions is limited by the syntax itself. The grammar of finitary isos is given by:

$$\omega \ ::= \ \{ \mid \ v_i \leftrightarrow e_i \ \}_{i \in I} \ \mid \ \lambda\phi.\omega \ \mid \ \phi \ \mid \ \omega\omega \ \mid \ \texttt{fix}^n\ \phi.\omega$$

where n is a natural number. The iso $\texttt{fix}^n\ \phi.\omega$ has the same typing rule as $\texttt{fix}\ \phi.\omega$ above. We introduce syntactic sugar, that denotes an expression that will never reduce, by induction on iso types:

$$\Omega_{A \leftrightarrow B} \overset{\text{def}}{=} \{\mid \cdot\} \qquad \Omega_{T_1 \to T_2} \overset{\text{def}}{=} \lambda\phi^{T_1}.\Omega_{T_2}$$

The syntax of finitary terms does not change compared to the language presented at the beginning of the chapter, with the addition of a term $\bot$, which indicates that there was no match when applying an iso abstraction to a value. The finitary operational semantics is then defined as:

$$\frac{}{\texttt{fix}^0\ \phi^T.\omega \underset{\text{fin}}{\to} \Omega_T} \qquad \frac{}{\texttt{fix}^{n+1}\ \phi.\omega \underset{\text{fin}}{\to} \omega[\texttt{fix}^n\ \phi.\omega/\phi]}$$

$$\frac{}{(\lambda\phi.\omega_1)\omega_2 \underset{\text{fin}}{\to} \omega_1[\omega_2/\phi]} \qquad \frac{\omega_1 \underset{\text{fin}}{\to} \omega_1'}{\omega_1\omega_2 \underset{\text{fin}}{\to} \omega_1'\omega_2}$$

$$\frac{\mathrm{match}(\sigma, v_i, v')}{\{ \mid v_i \leftrightarrow e_i \}_{i \in I} v' \underset{\text{fin}}{\to} \sigma(e_i)} \qquad \frac{\forall i, \neg(\mathrm{match}(\sigma, v_i, v'))}{\{ \mid v_i \leftrightarrow e_i \}_{i \in I} v' \underset{\text{fin}}{\to} \bot}$$

$$\frac{t_1 \underset{\text{fin}}{\to} t_2}{C_{\to}[t_1] \underset{\text{fin}}{\to} C_{\to}[t_2]} \qquad \frac{t \underset{\text{fin}}{\to} \bot}{C_{\to}[t] \underset{\text{fin}}{\to} \bot}$$

$$\frac{\mathrm{match}(\sigma, p, v)}{\mathtt{let}\, p = v\ \mathtt{in}\ t \underset{\text{fin}}{\to} \sigma(t)} \qquad \frac{\omega \underset{\text{fin}}{\to} \omega'}{\omega\ t \underset{\text{fin}}{\to} \omega'\ t}$$

Lemma 4.33 (Iso Subject Reduction). *If $\Psi \vdash_\omega \omega \colon T$ is well-formed, ω is finitary and $\omega \underset{\text{fin}}{\to} \omega'$, then $\Psi \vdash_\omega \omega' \colon T$.*

Proof. Strongly similar to Lemma 4.9. □

Lemma 4.34 (Iso Progress). *If $\Psi \vdash_\omega \omega \colon T$ is well-formed and ω is finitary, ω is either an iso value or there exists ω' such that $\omega \underset{\text{fin}}{\to} \omega'$.*

Proof. Strongly similar to Lemma 4.10. □

Lemma 4.35 (Subject Reduction). *If $\Psi; \Delta \vdash t \colon A$ is well-formed, t is finitary and $t \underset{\text{fin}}{\to} t'$, then $\Psi; \Delta \vdash t' \colon A$ is also well-formed.*

Proof. Strongly similar to Lemma 4.12. □

Lemma 4.36 (Progress). *If $\vdash t \colon A$ and t is finitary, then:*

- *either t is a value,*
- *or $t \to \bot$,*
- *or there exists t' such that $t \underset{\text{fin}}{\to} t'$.*

Proof. The proof is done by induction on $\vdash t \colon A$.

- $\vdash * \colon \mathtt{I}$ is a value.
- $\vdash t_1 \otimes t_2 \colon A \otimes B$. By induction hypothesis, either t_1 reduces, in which case $t_1 \otimes t_2$ reduces too, or t_1 is a value. If t_1 is a value, by induction hypothesis, either t_2 reduces, in which case $t_1 \otimes t_2$ reduces, or t_2 is a value, and thus $t_1 \otimes t_2$ is a value.
- $\Psi; \Delta \vdash \mathtt{inj}_i\, t \colon A_1 \oplus A_2$. By induction hypothesis, either t reduces, in which case $\mathtt{inj}_i\, t$ too, or t is a value, and thus $\mathtt{inj}_i\, t$ is a value.
- $\Psi; \Delta \vdash \mathtt{fold}\, t \colon \mu X.A$. By induction hypothesis, either t reduces, in which case $\mathtt{fold}\, t$ too, or t is a value, and thus $\mathtt{fold}\, t$ is a value.
- $\Psi; \Delta \vdash \omega\ t \colon B$. Thanks to Lemma 4.34, ω either reduces, in which case $\omega\ t$ also reduces, or it is an iso value $\{ \mid v_i \leftrightarrow e_i \}_{i \in I}$. In the last case, in the induction hypothesis gives that either t reduces, in which case $\omega\ t$ also reduces, or t is value. In that case, either t matches with one v_i, and $\omega\ t$ reduces, or it does not, and $\omega\ t \to \bot$.

- $\Psi; \Delta_1, \Delta_2 \vdash \texttt{let}\, p = t_1\ \texttt{in}\ t_2 \colon B$. In both cases of the induction hypothesis, $\texttt{let}\, p = t_1\ \texttt{in}\ t_2$ reduces.

□

Strong Normalisation. We prove that the reduction $\underset{\text{fin}}{\to}$ is strongly normalising, by observing that this system is separated in two very distinct systems: one that reduces the iso λ-terms, and another that performs the reversible computations. We show that both those systems can be extended to commute with each other, which ensures strong normalisation as long as both are strongly normalising. We start by introducing the system that performs the reductions on isos.

$$\frac{}{\texttt{fix}^0\ \phi^T.\omega \underset{\text{iso}}{\to} \Omega_T} \qquad \frac{}{\texttt{fix}^{n+1}\ \phi.\omega \underset{\text{iso}}{\to} \omega[\texttt{fix}^n\ \phi.\omega/\phi]}$$

$$\frac{}{(\lambda\phi.\omega_1)\omega_2 \underset{\text{iso}}{\to} \omega_1[\omega_2/\phi]} \qquad \frac{\omega_1 \underset{\text{iso}}{\to} \omega_1'}{\omega_1\omega_2 \underset{\text{iso}}{\to} \omega_1'\omega_2}$$

$$\frac{e_j \underset{\text{iso}}{\to} e_j'}{\{\mid v_i \leftrightarrow e_i \mid\}_{i\in I} \underset{\text{iso}}{\to} \left\{ \begin{array}{ll} \mid v_i \leftrightarrow e_i & \text{if } i \neq j \\ \mid v_j \leftrightarrow e_j' & \text{else} \end{array} \right\}}$$

$$\frac{t_1 \underset{\text{iso}}{\to} t_2}{C_\to[t_1] \underset{\text{iso}}{\to} C_\to[t_2]} \qquad \frac{t_1 \underset{\text{iso}}{\to} t_2}{\texttt{let}\, p = t'\ \texttt{in}\ t_1 \underset{\text{iso}}{\to} \texttt{let}\, p = t'\ \texttt{in}\ t_2} \qquad \frac{\omega \underset{\text{iso}}{\to} \omega'}{\omega\ t \underset{\text{iso}}{\to} \omega'\ t}$$

Lemma 4.37. *The reduction system $\underset{\text{iso}}{\to}$ is strongly normalising.*

Proof. We translate finitary isos and finitary terms into a simply-typed λ-calculus with pairs. We write $|t|$ the translation of t.

$$|*| \stackrel{\text{def}}{=} * \qquad |x| \stackrel{\text{def}}{=} * \qquad |\texttt{inj}_i\ t| \stackrel{\text{def}}{=} |t| \qquad |t \otimes t'| \stackrel{\text{def}}{=} \langle |t|, |t'| \rangle$$

$$|\texttt{fold}\ t| \stackrel{\text{def}}{=} |t| \qquad |\omega\ t| \stackrel{\text{def}}{=} \langle |\omega|, |t| \rangle \qquad |\texttt{let}\, p = t\ \texttt{in}\ t'| \stackrel{\text{def}}{=} \langle |t|, |t'| \rangle$$

$$\left|\{\mid v_i \leftrightarrow e_i \mid\}_{i\in I}\right| \stackrel{\text{def}}{=} \langle |e_i| \rangle_{i\in I} \qquad |\lambda\phi.\omega| \stackrel{\text{def}}{=} \lambda\phi.\,|\omega| \qquad |\phi| \stackrel{\text{def}}{=} \phi$$

$$|\omega_2\omega_1| \stackrel{\text{def}}{=} |\omega_2|\,|\omega_1| \qquad |\texttt{fix}^n\ \phi.\omega| \stackrel{\text{def}}{=} \texttt{fix}^n\ \phi.\,|\omega|$$

This λ-calculus is strong normalising; this can be proven with candidates of reducibility, *à la* System F [GTL89, Chapters 11 and 14]. □

We then introduce the reductions that perform the reversible computation, our equivalent of β-reduction but for our iso language.

$$\frac{\text{match}(\sigma, v_i, v')}{\{\mid v_i \leftrightarrow e_i \mid\}_{i\in I}\, v' \underset{\text{term}}{\to} \sigma(e_i)} \qquad \frac{\forall i, \neg(\text{match}(\sigma, v_i, v'))}{\{\mid v_i \leftrightarrow e_i \mid\}_{i\in I}\, v' \underset{\text{term}}{\to} \bot}$$

$$\frac{t_1 \underset{\text{term}}{\to} t_2}{C_\to[t_1] \underset{\text{term}}{\to} C_\to[t_2]} \qquad \frac{t \underset{\text{term}}{\to} \bot}{C_\to[t] \underset{\text{term}}{\to} \bot} \qquad \frac{\text{match}(\sigma, p, v)}{\texttt{let}\, p = v\ \texttt{in}\ t \underset{\text{term}}{\to} \sigma(t)}$$

This system is strongly normalising thanks to a decreasing argument: the number of isos and `let` contructors strictly decreases when applying the reduction $\underset{\text{term}}{\to}$.

Lemma 4.38. *The reduction system* $\underset{\text{term}}{\to}$ *is strongly normalising.*

Lemma 4.39. $\underset{\text{term}}{\to}\ \underset{\text{iso}}{\to} \subseteq \underset{\text{iso}}{\to}\ \underset{\text{term}}{\to}$.

We say that $\underset{\text{iso}}{\to}$ commutes [BD86] over $\underset{\text{term}}{\to}$. This and the strong normalisation of both systems $\underset{\text{iso}}{\to}$ and $\underset{\text{term}}{\to}$ ensures the strong normalisation of them combined $\underset{\text{iso}}{\to} \cup \underset{\text{term}}{\to}$ [BD86, Theorem 1].

Lemma 4.40. $\underset{\text{fin}}{\to} \subseteq \underset{\text{iso}}{\to} \cup \underset{\text{term}}{\to}$.

Proof. The proof is direct, by showing that any rule in $\underset{\text{fin}}{\to}$ is either in $\underset{\text{iso}}{\to}$ or $\underset{\text{term}}{\to}$. □

Theorem 4.41. *The reduction system* $\underset{\text{fin}}{\to}$ *is strongly normalising.*

Proof. With [BD86, Theorem 1] and Lemmas 4.37, 4.38, and 4.39, we can ensure that $\underset{\text{iso}}{\to} \cup \underset{\text{term}}{\to}$ is strongly normalising. We conclude then with Lemma 4.40, that shows that $\underset{\text{fin}}{\to}$ is a subsystem of a strongly normalising system. □

Finitary adequacy. We prove adequacy, but for finitary terms. To do so, we also need to introduce the denotational semantics of finitary isos. The interpratation of $\texttt{fix}^n$, instead of being Kleene's fixed point, is the morphism obtained by unfolding n times. The interpretation of Ω_T is the bottom element of $[\![T]\!]$.

Lemma 4.42. *Given a well-formed finitary iso judgement* $\vdash_\omega \omega \colon T$, *if* $\omega \underset{\text{fin}}{\to} \omega'$, *then*

$$[\![\vdash_\omega \omega \colon T]\!] = [\![\vdash_\omega \omega' \colon T]\!] .$$

Proof. Strongly similar to Lemma 4.30. □

Proposition 4.43 (Finitary Soundness). *Given a well-formed finitary term judgement* $\vdash t \colon A$, *if* $t \underset{\text{fin}}{\to} t'$, *then*

$$[\![\vdash t \colon A]\!] = [\![\vdash t' \colon A]\!] .$$

Proof. Strongly similar to Prop. 4.31. □

Theorem 4.44 (Finitary Adequacy). *Given a well-formed finitary term judgement* $\vdash t \colon A$, $t \downarrow$ *iff* $[\![\vdash t \colon A]\!] \neq 0_{[\![A]\!]}$.

Proof. We prove both directions of the double implication.

($\Rightarrow$) Knowing that $t \downarrow$, there exists a value v such that $t \underset{\text{fin}}{\to}^* v$, and Prop. 4.43 ensures that $[\![t]\!] = [\![v]\!] \neq 0$.

($\Leftarrow$) We know that $\underset{\text{fin}}{\to}$ is strongly normalising (see Th. 4.41), which means that the reduction from t terminates, and Lemma 4.36 ensures that it terminates either on a value v or on $\bot$. However, $[\![t]\!] \neq 0$, thus it cannot terminate on $\bot$ because of Prop. 4.43. We have then $t \underset{\text{fin}}{\to}^* v$, which concludes.

□

Finitary Subterms. We conclude in two steps. First we observe that the interpretation of a term is nothing more than the join of the interpretations of its finitary subterms, then we show that a reduction $\to^*$ can be linked to a finitary reduction $\underset{\text{fin}}{\to}^*$.

Definition 4.45 (Finitary Subiso). Let $\lhd$ be the smallest relation between (finitary or not) isos such that:

$$\frac{}{\texttt{fix}^n\ \phi.\omega \lhd \texttt{fix}\ \phi.\omega} \qquad \frac{\omega_1 \lhd \omega_2}{\omega[\omega_1/\phi] \lhd \omega[\omega_2/\phi]}$$

Lemma 4.46. *Given two well-formed (finitary or not) iso judgements* $\Psi \vdash_\omega \omega_1 \colon T$ *and* $\Psi \vdash_\omega \omega_2 \colon T$ *such that* $\omega_1 \lhd \omega_2$*, then*

$$[\![\Psi \vdash_\omega \omega_1 \colon T]\!] \leq [\![\Psi \vdash_\omega \omega_2 \colon T]\!] .$$

Proof. Direct. □

Lemma 4.47. *Given a well-formed iso judgement* $\Psi \vdash_\omega \omega \colon T$*, we have:*

$$[\![\Psi \vdash_\omega \omega \colon T]\!] = \bigvee_{\substack{\omega' \lhd\ \omega \\ \omega' \text{ finitary}}} [\![\Psi \vdash_\omega \omega' \colon T]\!] .$$

Proof. We observe that, by definition:

$$[\![\Psi \vdash_\omega \texttt{fix}\ \phi.\omega \colon T]\!] = \bigvee_{n \in \mathbb{N}} [\![\Psi \vdash_\omega \texttt{fix}^n\ \phi.\omega \colon T]\!]$$

which proves the desired result in the case of $\texttt{fix}\ \phi.\omega$. The general conclusion falls by induction. □

We generalise to terms the definition of subisos given above.

Definition 4.48 (Finitary Subterm). Let $\lhd$ be the smallest congruence relation between (finitary or not) terms such that:

$$\frac{\omega_1 \lhd\ \omega_2}{\omega_1\ t \lhd\ \omega_2\ t}$$

The following lemma follows from the previous definition and Lemma 4.47; this is because composition is distributive with joins (see Definition 1.65).

Lemma 4.49. *Given a well-formed term judgement $\Psi; \Delta \vdash t \colon A$, we have:*

$$[\![\Psi; \Delta \vdash t \colon A]\!] = \bigvee_{\substack{t' \triangleleft t \\ t' \text{ finitary}}} [\![\Psi; \Delta \vdash t' \colon A]\!] .$$

It is also the right time to observe that if a term has a finitary subterm that reduces to a value eventually, the former also normalises to the same value.

Lemma 4.50. *Given a well-formed closed term judgement $\vdash t \colon A$, if there exists a finitary subterm $t' \triangleleft t$ and a value such that $t' \underset{\text{fin}}{\to}{}^* v$, then $t \to^* v$.*

Proof. The finitary term t' has the same reduction steps as t up to a point. Lemma 4.36 ensures that this end point is either a value or $\bot$ in the finitary case. Thus if the reduction from t' gets to a value, the reduction from t must also finish on a value. Since the reduction steps were exactly the same, both reductions have the same normal form. □

Conclusion. We finally have all the tools to conclude with adequacy for closed terms of our original language.

Proof of Adequacy (Theorem 4.29). There are two implications to prove.

($\Rightarrow$) This first implication is proven as Corollary 4.32.

($\Leftarrow$) Suppose that $[\![t]\!] \neq 0$. Necessarily, thanks to Lemma 4.49, there exists a finitary term t' such that $t' \triangleleft t$ and $[\![t']\!] \neq 0$. In Theorem 4.44, we have proven adequacy for finitary terms, meaning that there exists a value v such that $t' \underset{\text{fin}}{\to}{}^* v$. Lemma 4.50 ensures then that $t \to^* v$, which concludes.

□

4.5 Expressivity

This section is devoted to assessing the expressivity of the language. To that end, we rely on Reversible Turing Machine (RTM) [AG11]. We describe how to encode an RTM as an iso, and prove that the iso realises the string semantics of the RTM.

Reference. The work in this section has been done by Kostia Chardonnet, firstly as a part of his thesis [Cha23], and in our paper [CLV23]. Is is presented here out of coherence with the next section.

4.5.1 Recovering duplication, erasure and manipulation of constants

Although the language is linear and reversible, since closed values are all finite, and one can build isos to encode notions of duplication, erasure, and constant manipulation thanks to partiality.

Definition 4.51 (Duplication [CLV23]). We define Dup_A^S the iso of type $A \leftrightarrow A \otimes A$ which can duplicate any closed value of type A by induction on A, where S is a set of pairs of a type-variable X and an iso-variable ϕ, such that for every free-variable $X \subseteq A$, there exists a unique pair $(X, \phi) \in S$ for some ϕ. The iso is defined by induction on A: $\mathrm{Dup}_{\mathbf{1}}^S = \{() \leftrightarrow \langle (), () \rangle\}$, and

- $\mathrm{Dup}_{A\otimes B}^S = \left\{ \begin{array}{rcl} \langle x, y \rangle & \leftrightarrow & \mathtt{let}\ \langle x_1, x_2 \rangle = \mathrm{Dup}_A^S\ x\ \mathtt{in}\ \mathtt{let}\ \langle y_1, y_2 \rangle = \mathrm{Dup}_B^S\ y\ \mathtt{in} \\ & & \langle \langle x_1, y_1 \rangle, \langle x_2, y_2 \rangle \rangle \end{array} \right\}$;
- $\mathrm{Dup}_{A\oplus B}^S = \left\{ \begin{array}{rcl} \mathtt{inj}_l\ (x) & \leftrightarrow & \mathtt{let}\ \langle x_1, x_2 \rangle = \mathrm{Dup}_A^S\ x\ \mathtt{in}\ \langle \mathtt{inj}_l\ (x_1), \mathtt{inj}_l\ (x_2) \rangle \\ \mathtt{inj}_r\ (y) & \leftrightarrow & \mathtt{let}\ \langle y_1, y_2 \rangle = \mathrm{Dup}_B^S\ y\ \mathtt{in}\ \langle \mathtt{inj}_r\ (y_1), \mathtt{inj}_r\ (y_2) \rangle \end{array} \right\}$;
- If $(X, \_) \notin S$: $\mathrm{Dup}_{\mu X.A}^S = \mathtt{fix}\ \phi. \left\{ \begin{array}{rcl} \mathtt{fold}\ (x) & \leftrightarrow & \mathtt{let}\ \langle x_1, x_2 \rangle = Dup_{A[\mu X.A/X]}^{S\cup\{(X,\phi)\}}\ x\ \mathtt{in} \\ & & \langle \mathtt{fold}\ (x_1), \mathtt{fold}\ (x_2) \rangle \end{array} \right\}$;
- If $(X, \phi) \in S$: $\mathrm{Dup}_{\mu X.A}^S = \{x \leftrightarrow \mathtt{let}\ \langle x_1, x_2 \rangle = \phi\ x\ \mathtt{in}\ \langle x_1, x_2 \rangle\}$.

Lemma 4.52 (Properties of Duplication [CLV23]). *Given a closed type A, then $\mathrm{Dup}_A^\emptyset$ is well-defined, and the iso $\mathrm{Dup}_A^\emptyset$ is well typed of type $A \leftrightarrow A \otimes A$.*

Lemma 4.53 (Semantics of Duplication [CLV23]). *Given a closed type A and a closed value v of type A, then $\mathrm{Dup}_A^\emptyset\ v \to^* \langle v_1, v_2 \rangle$ and $v = v_1 = v_2$.*

Definition 4.54 (Constant manipulation [CLV23]). We define $\mathrm{erase}_v \colon A \otimes \Sigma^T \leftrightarrow A$ which erase its second argument when its value is v as $\{\langle x, v \rangle \leftrightarrow x\}$. Reversed, it turns any x into $\langle x, v \rangle$.

4.5.2 Definition of Reversible Turing Machine

Definition 4.55 (Reversible Turing Machine [AG11]). Given a Turing Machine $M = (Q, \Sigma, \delta, b, q_s, q_f)$, where Q is a set of states, $\Sigma = \{b, a_1, \ldots, a_n\}$ is a finite set of tape symbols (in the following, a_i and b always refer to elements of Σ), $\delta \subseteq \Delta = (Q \times [(\Sigma \times \Sigma) \cup \{\leftarrow, \downarrow, \rightarrow\}] \times Q)$ is a partial relation defining the transition relation such that there must be no transitions leading out of q_f nor into q_s, b a blank symbol and q_s and q_f the initial and final states. We say that M is a *Reversible Turing Machine* (RTM) if it is:

- *forward* deterministic: for any two distinct pairs of triples (q_1, a_1, q_1') and (q_2, a_2, q_2') in δ, if $q_1 = q_2$ then $a_1 = (s_1, s_1')$ and $a_2 = (s_2, s_2')$ and $s_1 \neq s_2$.
- *Backward* deterministic: for any two distinct pairs of triples (q_1, a_1, q_1') and (q_2, a_2, q_2') in δ, if $q_1' = q_2'$ then $a_1 = (s_1, s_1')$ and $a_2 = (s_2, s_2')$ and $s_1' \neq s_2'$.

Definition 4.56 (Configurations [AG11]). A *configuration* of a RTM is a tuple $(q, (l, s, r)) \in \mathrm{Conf} = Q \times (\Sigma^* \times \Sigma \times \Sigma^*)$ where q is the internal state, l, r are the left and right parts of the tape (as string) and $s \in \Sigma$ is the current symbol being scanned. A configuration is *standard* when the cursor is on the immediate left of a finite, blank-free string $s \in (\Sigma \setminus \{b\})^*$ and the rest is blank, *i.e.* it is in configuration $(q, (\epsilon, b, s))$ for some q, where ϵ is the empty string, representing an infinite sequence of blank symbols b.

Definition 4.57 (RTM Transition [AG11]). An RTM M in configuration $C = (q, (l, s, r))$ goes to a configuration $C' = (q', (l', s', r'))$, written $T \vdash C \rightsquigarrow C'$ in a single step if there exists a transition $(q, a, q') \in \delta$ where a is either (s, s'), and then $l = l'$ and $r = r'$ or $a \in \{\leftarrow, \downarrow, \rightarrow\}$, and we have for the case $a = \leftarrow$: $l' = l \cdot s$ and for $r = x \cdot r_2$ we have $s' = x$ and $r' = r_2$, similarly for the case $a = \rightarrow$ and for the case $a = \downarrow$ we have $l' = l$ and $r' = r$ and $s = s'$.

The semantics of an RTM is given on *standard configurations* of the form $(q, (\epsilon, b, s))$ where q is a state, ϵ is the finite string standing for a blank-filled tape, and s is the blank-free, finite input of the RTM.

Definition 4.58 (String Semantics [AG11]). The semantics of a RTM M, written $\mathrm{Sem}(M)$ is defined on standards configurations and is given by the set $\mathrm{Sem}(M) = \{(s, s') \in ((\Sigma \backslash \{b\})^* \times (\Sigma \backslash \{b\})^*) \mid M \vdash (q_s, (\epsilon, b, s)) \rightsquigarrow^* (q_f, (\epsilon, b, s'))\}$.

Theorem 4.59 (Properties of RTM [AG11]). *For all RTM M, $\mathrm{Sem}(M)$ is the graph of an injective function. Conversely, all injective computable functions (on a tape) are realisable by a RTM. Finally, any Turing Machine can be simulated by a Reversible Turing Machine.*

4.5.3 Encoding RTMs as Isos

A RTM configuration is a set-based construction that we can model using the type constructors available in our language. Because the transition relation δ is backward and forward deterministic, it can be encoded as an iso. Several issues need to be dealt with; we discuss them in this section.

Encoding configurations. The set of states $Q = \{q_1, \ldots, q_n\}$ is modelled with the type $Q^T = I \oplus \cdots \oplus I$ (n times). The encoding of the state q_i is then a closed value q_i^T. They are pairwise orthogonal. The set Σ of tape symbols is represented similarly by $\Sigma^T = I \oplus \cdots \oplus I$, and the encoding of the tape symbol a is a^T. We then define the type of configurations in the obvious manner: a configuration $C = (q, (l, s, r))$ corresponds to a closed value $\mathrm{isos}(C)$ of type $Q^T \otimes ([\Sigma^T] \otimes \Sigma^T \otimes [\Sigma^T])$.

Encoding the transition relation δ. A limitation of our language is that every sub-computation has to be reversible and does not support infinite data structures such as streams. In the context of RTMs, the empty string ϵ is assimilated with an infinite string of blank symbols. If this can be formalised in set theory, in our limited model, we cannot emit blank symbols out of thin air without caution.

In order to simulate an infinite amount of blank symbols on both sides of the tape during the evaluation, we provide an iso that grows the size of the two tapes on both ends by blank symbols at each transition step. The iso growth is shown in Table 4.1. It is built using three auxiliary functions, written in a Haskell-like notation len sends a closed value $[v_1, \ldots, v_n]$ to $[v_1, \ldots, v_n] \otimes \overrightarrow{n}$. $\mathrm{snoc'}$ sends $[v_1, \ldots, v_n] \otimes v, \overrightarrow{n}$ to $[v_1, \ldots, v_n, v] \otimes v, \overrightarrow{n}$. snoc sends $\langle [v_1, \ldots, v_n], v \rangle$ to $\langle [v_1, \ldots, v_n, v], v \rangle$. Finally, growth sends $\langle [a_1^T, \ldots, a_n^T], [a_1'^T, \ldots, a_m'^T] \rangle$ to $\langle [a_1^T, \ldots, a_n^T, b^T], [a_1'^T, \ldots, a_m'^T, b^T] \rangle$.

Now, given a RTM $M = (Q, \Sigma, \delta, b, q_s, q_f)$, a relation $(q, r, q') \in \delta$ is encoded as a clause between values $\mathrm{iso}(q, r, q') = v_1 \leftrightarrow v_2$ of type $C^T \leftrightarrow C^T$. These clauses are defined by case analysis on r as follows. When x, x', z, y and y' are variables:

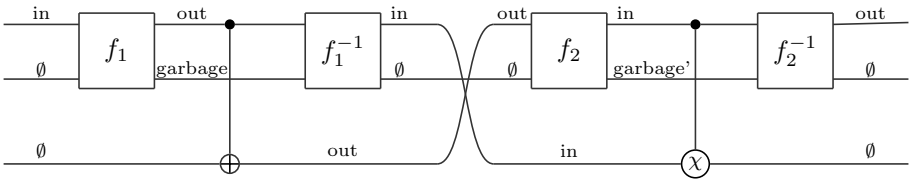


Figure 4.6 – Reversibly removing additional garbage from some process.

- $\mathrm{iso}(q, \rightarrow, q') = (q^T, (x', z, y :: y')) \leftrightarrow \texttt{let}\,(l, r) = \mathrm{growth}\ (x', y')\ \texttt{in}\ (q'^T, (z :: l, y, r))$,
- $\mathrm{iso}(q, \leftarrow, q') = (q^T, (x :: x', z, y')) \leftrightarrow \texttt{let}\,(l, r) = \mathrm{growth}\ (x', y')\ \texttt{in}\ (q'^T, (l, x, z :: r))$,
- $\mathrm{iso}(q, \downarrow, q') = (q^T, (x', z, x')) \leftrightarrow \texttt{let}\,(l, r) = \mathrm{growth}\ (x', y')\ \texttt{in}\ (q'^T, (l, z, r))$,
- $\mathrm{iso}(q, (s, s'), q') = (q^T, (x', s^T, y')) \leftrightarrow \texttt{let}\,(l, r) = \mathrm{growth}\ (x', y')\ \texttt{in}\ (q'^T, (l, s'^T, r))$.

The encoding of the RTM M is then the iso $\mathrm{isos}(M)$ whose clauses are the encoding of each rule of the transition relation δ, of type $\mathrm{Conf}^T \leftrightarrow \mathrm{Conf}^T$.

Encoding successive applications of δ. The transition δ needs to be iterated until the final state is reached. This behavior can be emulated in our language using the iso It, defined in Table 4.1. The iso $\mathrm{It}\ \omega$ is typed with $(A \leftrightarrow A \otimes \texttt{nat})$. Fed with a value of type A, it iterates ω until $\texttt{ff}$ is met. It then returns the result together with the number of iterations.

To iterate $\mathrm{iso}(M)$, we then only need to modify iso to return a boolean stating whether q_f was met. This can be done straightforwardly, yielding an iso $\mathrm{isos}_{\mathbb{B}}(M))$ of type $\mathrm{Conf}^T \leftrightarrow \mathrm{Conf}^T \otimes (I \oplus I)$. With such an iso, given M be a RTM such that $M \vdash (q_s, (\epsilon, b, s)) \rightsquigarrow^{n+1} (q_f, (\epsilon, b, (a_1, \ldots, a_n)))$, then $\mathrm{It}(\mathrm{isos}_{\mathbb{B}}(M))\ (q_s^T, ([b^T], b^T, s^T))$ reduces to the encoding term $((q_f^T, ([b^T, \ldots, b^T], b^T, [a_1^T, \ldots, a_n^T, b^T, \ldots, b^T])), \overline{n})$. If it were not for the additional blank tape elements, we would have the encoding of the final configuration.

Recovering a canonical presentation. Removing blank states at the *beginning* of a list is easy: it can for instance, be done with the iso $\mathrm{rmBlank}$, shown in Table 4.1. Cleaning up the tail of the list can then be done by reverting the list, using, e.g. rev in the same table. By abuse of notation, we use constants in some patterns: an exact representation would use Définition 4.54. Finally, we can define the operator $\mathrm{cleanUp}$, solving the issue raised in the previous paragraph. In particular, given a RTM M and an initial configuration C such that $M \vdash C \rightsquigarrow C' = (q, (\epsilon, b, (a_1, \ldots, a_n)))$. Then we have that $\mathrm{cleanUp}\,\mathrm{It}(\mathrm{isos}_{\mathbb{B}}(M))C^T \rightarrow^* ((q^T, ([], b^T, [a_1^T, \ldots, a_n^T])), v)$, where v is of type $\texttt{nat} \otimes \texttt{nat} \otimes \texttt{nat} \otimes [\Sigma^T]$. If we want to claim that we indeed capture the operational behaviour of RTMS, we need to get rid of this value v.

Getting rid of the garbage. To discard this value v, we rely on Bennett's trick [Ben73], shown in Figure 4.6. Given two Turing machines f_1 and f_2 and some input in such that if $f_1(\mathrm{in}) = \mathrm{out} \otimes \mathrm{garbage}$ and $f_2(\mathrm{out}) = \mathrm{in} \otimes \mathrm{garbage'}$, then the process consists of taking additional tapes in the Turing Machine in order to reversibly duplicate (represented by the $\oplus$) or reversibly erase some data (represented by the χ) in order to recover only the output of f_1, without any garbage.

Given an iso $\omega\colon A \leftrightarrow B \otimes C$ and $\omega'\colon B \leftrightarrow A \otimes C'$ where C, C' represent garbage, we can build an iso from $A \leftrightarrow B$ as follows, where the variables x, y, z (and their indices) respectively correspond to the first, second, and third wire of Figure 4.6. This operator makes use of the

$$
\begin{array}{l}
\mathrm{len} : [A] \leftrightarrow [A] \otimes \mathtt{nat} \\
\mathrm{len}\,[\,] \quad\;\; \leftrightarrow ([\,], 0) \\
\mathrm{len}\, h :: t \leftrightarrow \mathtt{let}\,(t', n) = \mathrm{len}\ t\ \mathtt{in} \\
\qquad\qquad\quad (h :: t', S(n))
\end{array}
$$

$$
\begin{array}{l}
\mathrm{snoc'} : [A] \otimes A \otimes \mathtt{nat} \leftrightarrow [A] \otimes A \otimes \mathtt{nat} \\
\mathrm{snoc'}\,([\,], x, 0) \qquad\quad \leftrightarrow \mathtt{let}\,(x_1, x_2) = \mathrm{Dup}_A^{\emptyset}\, x\ \mathtt{in} \\
\qquad\qquad\qquad\qquad\quad ([x_1], x_2, 0) \\
\mathrm{snoc'}\,(h :: t, x, S(n)) \leftrightarrow \mathtt{let}\,(t', x', n') = \mathrm{snoc'}(t, x, n)\ \mathtt{in} \\
\qquad\qquad\qquad\qquad\quad (h :: t', x', S(n'))
\end{array}
$$

$$
\begin{array}{l}
\mathrm{growth} : [\Sigma^T] \otimes [\Sigma^T] \leftrightarrow [\Sigma^T] \otimes [\Sigma^T] \\
\mathrm{growth}\,(l, r) \leftrightarrow \mathtt{let}\,\langle l', b_1\rangle = \mathrm{snoc}\langle l, b^T\rangle\ \mathtt{in} \\
\qquad\qquad\qquad\quad \mathtt{let}\,\langle r', b_2\rangle = \mathrm{snoc}\langle r, b^T\rangle\ \mathtt{in} \\
\qquad\qquad\qquad\quad \mathtt{let}\, l'' = \mathrm{erase}_b\langle l', b_1\rangle\ \mathtt{in} \\
\qquad\qquad\qquad\quad \mathtt{let}\, r'' = \mathrm{erase}_b\langle r', b_2\rangle\ \mathtt{in}\ (l'', r'')
\end{array}
$$

$$
\begin{array}{l}
\mathrm{It} : (A \leftrightarrow A \otimes (\mathtt{I} \oplus \mathtt{I})) \to (A \leftrightarrow A \otimes \mathtt{nat}) \\
\mathrm{It}\,\psi\, x \leftrightarrow \mathtt{let}\, y = \psi\ x\ \mathtt{in} \\
\qquad\qquad\quad \mathtt{let}\, z = \left\{ \begin{array}{l} (y, \mathtt{tt}) \leftrightarrow \mathtt{let}\,(z, n) = (\mathrm{It}\ \psi)\ y\ \mathtt{in}\ (z, S\ n) \\ (y, \mathtt{ff}) \leftrightarrow (y, 0) \end{array} \right\}\ y\ \mathtt{in}\ z
\end{array}
$$

$$
\begin{array}{l}
\mathrm{rmBlank} : [\Sigma] \leftrightarrow [\Sigma] \otimes \mathbb{N} \\
\mathrm{rmBlank}\,[] \qquad\;\; \leftrightarrow ([], 0) \\
\mathrm{rmBlank}\, b^T :: t \leftrightarrow \mathtt{let}\,(t', n) = \mathrm{rmBlank}\ t\ \mathtt{in}\ (t', S(n)) \\
\mathrm{rmBlank}\, a_1^T :: t \leftrightarrow ((a_1^T :: t), 0) \\
\vdots \qquad\quad \vdots \qquad\;\; \vdots\ \vdots \\
\mathrm{rmBlank}\, a_n^T :: t \leftrightarrow ((a_n^T :: t), 0)
\end{array}
$$

$$
\begin{array}{l}
\mathrm{rev}_{\mathsf{aux}} : [A] \otimes [A] \leftrightarrow [A] \otimes [A] \\
\mathrm{rev}_{\mathsf{aux}}\,([], y) \qquad\;\; \leftrightarrow ([], y) \\
\mathrm{rev}_{\mathsf{aux}}\,(h :: t, y) \leftrightarrow \mathtt{let}\,(h_1, h_2) = \mathrm{Dup}_A^{\emptyset}\ h\ \mathtt{in} \\
\qquad\qquad\qquad\qquad\quad \mathtt{let}\,(t_1, t_2) = \phi(t, h_2 :: y)\ \mathtt{in} \\
\qquad\qquad\qquad\qquad\quad (h_1 :: t_1, t_2)
\end{array}
$$

$$
\begin{array}{l}
\mathrm{rev} : [A] \leftrightarrow [A] \otimes [A] \\
\mathrm{rev} = \{x \leftrightarrow \mathtt{let}\,(t_1, t_2) = \mathrm{rev}_{\mathsf{aux}}\ (x, [])\ \mathtt{in}\ (t_1, t_2)\}
\end{array}
$$

$$
\begin{array}{l}
\mathrm{cleanUp} : C^T \otimes \mathtt{nat} \leftrightarrow C^T \otimes \mathtt{nat} \otimes \mathtt{nat} \otimes \mathtt{nat} \otimes [\Sigma^T] \\
\mathrm{cleanUp}\,((x, (l, y, r)), n) \leftrightarrow \mathtt{let}\,(l', n_1) = \mathrm{rmBlank}\ l\ \mathtt{in} \\
\qquad\qquad\qquad\qquad\qquad\quad \mathtt{let}\,(r_{\mathsf{ori}}, r_{\mathsf{rev}}) = \mathrm{rev}\ r\ \mathtt{in} \\
\qquad\qquad\qquad\qquad\qquad\quad \mathtt{let}\,(r', n_2) = \mathrm{rmBlank}\ r_{\mathsf{rev}}\ \mathtt{in} \\
\qquad\qquad\qquad\qquad\qquad\quad ((x, (l', y, r')), n, n_1, n_2, r_{\mathsf{ori}})
\end{array}
$$

Table 4.1 – Some useful isos for the encoding.

iso Dup discussed in Section 4.5.1.

$$\begin{array}{rcl}\mathrm{GarbRem}(\omega,\omega')\ x_1 & \leftrightarrow & \mathtt{let}\ \langle x_2,y\rangle = \omega\ x_1\ \mathtt{in}\ \mathtt{let}\ \langle x_3,z\rangle = \mathrm{Dup}_B^{\emptyset}\ x_2\ \mathtt{in}\\ & & \mathtt{let}\ x_4 = \omega^{-1}\ \langle x_3,y\rangle\ \mathtt{in}\ \mathtt{let}\ \langle z_2,y_2\rangle = \omega'\ z\ \mathtt{in}\\ & & \mathtt{let}\ z_3 = (\mathrm{Dup}_B^{\emptyset})^{-1}\ \langle z_2,x_4\rangle\ \mathtt{in}\ \mathtt{let}\ z_4 = \omega'^{-1}\ \langle z_3,y_2\rangle\ \mathtt{in}\ z_4.\end{array}$$

Theorem 4.60 (Capturing the exact semantics of a RTM [CLV23]). *For all RTM M with standard configurations $C = (q_s,(\epsilon,b,s))$ and $C' = (q_f,(\epsilon,b,s'))$ such that $M \vdash C \rightsquigarrow^* C'$, we have*

$$\mathrm{GarbRem}(\mathrm{cleanUp}(\mathrm{It}(\mathrm{isos}_{\mathbb{B}}(M))),\mathrm{cleanUp}(\mathrm{It}(\mathrm{isos}_{\mathbb{B}}(M^{-1}))))\ \ \mathrm{isos}(C) \to^* \mathrm{isos}(C')$$

The behavior of RTMs is thus captured by the language.

4.6 Semantics preservation

In this section, we fix the interpretation $[\![-]\!]$ of the language in $\mathbf{PInj}$, the category of sets and partial injections. This choice comes without any loss of generality, and allows us to consider *computable* set-functions. In this section, we show that given a computable, reversible set-function $f\colon [\![A]\!] \to [\![B]\!]$, there exists an iso $\omega\colon A \leftrightarrow B$ such that $[\![\omega]\!] = f$. In order to do that, we fix a canonical flat representation of our types.

4.6.1 A Canonical Representation

We define a canonical representation of closed values of some type A into a new type $\mathrm{Enc} = \mathbb{B} \oplus \mathtt{I} \oplus \mathtt{I} \oplus \mathtt{I} \oplus \mathtt{I} \oplus \mathtt{nat}$ (recall that $\mathbb{B} = \mathtt{I} \oplus \mathtt{I}$ and $\mathtt{nat} = \mu X.\mathtt{I} \oplus X$). For simplicity let us name each the following terms of type Enc : $\mathbf{tt} = \mathtt{inj}_l\ (\mathtt{inj}_l\ ())$, $\mathbf{ff} = \mathtt{inj}_l\ (\mathtt{inj}_r\ ())$, $S = \mathtt{inj}_r\ (\mathtt{inj}_l\ ())$, $D^{\oplus} = \mathtt{inj}_r\ (\mathtt{inj}_r\ (\mathtt{inj}_l\ ()))$, $D^{\otimes} = \mathtt{inj}_r\ (\mathtt{inj}_r\ (\mathtt{inj}_r\ (\mathtt{inj}_l\ ())))$, $D^{\mu} = \mathtt{inj}_r\ (\mathtt{inj}_r\ (\mathtt{inj}_r\ (\mathtt{inj}_r\ (\mathtt{inj}_l\ ()))))$, and for every natural number n, we write $\tilde{n}$ for the term $\mathtt{inj}_r\ (\mathtt{inj}_r\ (\mathtt{inj}_r\ (\mathtt{inj}_r\ (\mathtt{inj}_r\ (\mathtt{inj}_r\ (\overline{n}))))))$. Now, given some closed type A, we can define $\lfloor - \rfloor_A\colon A \leftrightarrow [\mathrm{Enc}]$ the iso that transform any closed value of type A into a list of Enc. The iso is defined inductively over A: $\lfloor - \rfloor_{\mathtt{I}} = \{() \leftrightarrow [S]\}$, and

$$\lfloor - \rfloor_{A\oplus B} = \left\{ \begin{array}{rcl} \mathtt{inj}_l\ (x) & \leftrightarrow & \mathtt{let}\ y = \lfloor x \rfloor_A\ \mathtt{in}\ D^{\oplus} :: \mathbf{ff} :: y \\ \mathtt{inj}_r\ (x) & \leftrightarrow & \mathtt{let}\ y = \lfloor x \rfloor_B\ \mathtt{in}\ D^{\oplus} :: \mathbf{tt} :: y \end{array} \right\},$$

$$\lfloor - \rfloor_{A\otimes B} = \left\{ \begin{array}{rcl} \langle x,y\rangle & \leftrightarrow & \mathtt{let}\ x' = \lfloor x \rfloor_A\ \mathtt{in}\ \mathtt{let}\ y' = \lfloor y \rfloor_B\ \mathtt{in} \\ & & \mathtt{let}\ \langle z,n\rangle = +\!+\ \langle x',y'\rangle\ \mathtt{in}\ D^{\otimes} :: \tilde{n} :: z \end{array} \right\},$$

$$\lfloor - \rfloor_{\mu X.A} = \left\{ \begin{array}{rcl} \mathtt{fold}\ x & \leftrightarrow & \mathtt{let}\ y = \lfloor x \rfloor_{A[\mu X.A/X]}\ \mathtt{in}\ D^{\mu} :: y \end{array} \right\},$$

where the iso $+\!+\colon [A] \otimes [A] \leftrightarrow [A] \otimes \mathtt{nat}$ is defined as:

$$\mathtt{fix}\ f. \left\{ \begin{array}{rcl} \langle [], x\rangle & \leftrightarrow & \langle x, 0\rangle \\ \langle h :: t, x\rangle & \leftrightarrow & \mathtt{let}\ \langle y,n\rangle = f\ \langle t,x\rangle\ \mathtt{in}\ \langle h :: y, S(n)\rangle \end{array} \right\}.$$

4.6.2 Capturing every computable injection

With this encoding, every iso $\omega : A \leftrightarrow B$ can be turned into another iso $\lfloor \omega \rfloor : [\mathrm{Enc}] \leftrightarrow [\mathrm{Enc}]$ by composing $\lfloor - \rfloor_A$, followed by ω, followed by $\lfloor - \rfloor_B^{-1}$. This is in particular the case for isos that are the images of a Turing Machine. We are now ready to see how every computable function f from $[\![A]\!] \to [\![B]\!]$ can be turned into an iso whose semantics is f. Given a computable function $f : [\![A]\!] \to [\![B]\!]$, call M_f the RTM computing f. Since f is in $\mathbf{PInj}$, its output uniquely determines its input. Following [Ben73], there exists another Turing Machine M'_f which, given the output of M_f recovers the initial input. In our encoding of a RTM, the iso will have another additional garbage which consist of a natural number, *i.e.* the number of steps of the RTM M_f. Using $\mathrm{GarbRem}(\mathrm{isos}(M_f), \mathrm{isos}(M'_f))$ we can obtain a single iso, from the encoding of A to the encoding of B, without any garbage left. This also ensures that $\left[\!\left[\mathrm{GarbRem}(\mathrm{isos}(M_f), \mathrm{isos}(M'_f))\right]\!\right](x) = (\left[\!\left[\mathrm{isos}(M_f)\right]\!\right](x))_1$, for any input x.

Theorem 4.61 (Computable function as Iso). *Given a computable function $f : [\![A]\!] \to [\![B]\!]$, let $g : [\![[\mathrm{Enc}] \otimes [\mathrm{Enc}]]\!] \to [\![[\mathrm{Enc}] \otimes [\mathrm{Enc}]]\!]$ be defined as $g = [\![\lfloor - \rfloor_B]\!] \circ f \circ [\![\lfloor - \rfloor_A^{-1}]\!]$, and let $\omega : A \leftrightarrow B$ be defined as*

$$\begin{aligned}\{x \ \leftrightarrow\ & \texttt{let}\ y = \lfloor x \rfloor_A\ \texttt{in} \\ & \texttt{let}\ y' = \mathrm{GarbRem}(\mathrm{isos}(M_g), \mathrm{isos}(M'_g))\ y\ \texttt{in} \\ & \texttt{let}\ z = \lfloor y' \rfloor_B^{-1}\ \texttt{in}\ z\}.\end{aligned}$$

Then $[\![\omega]\!] = f$.

Proof. In ω, call the right-hand-side e. Notice that

$$[\![e]\!] = [\![\lfloor - \rfloor_B^{-1}]\!] \circ [\![\mathrm{GarbRem}(\mathrm{isos}(M_g), \mathrm{isos}(M'_g))]\!] \circ [\![\lfloor - \rfloor_A]\!]\,.$$

By Prop. 4.31, we know that $[\![\mathrm{GarbRem}(\mathrm{isos}(M_g), \mathrm{isos}(M'_g))]\!] = g$. Therefore, since $g = [\![\lfloor - \rfloor_B]\!] \circ f \circ [\![\lfloor - \rfloor_A^{-1}]\!]$ by definition, we get

$$\begin{aligned}[\![e]\!] &= [\![\lfloor - \rfloor_B^{-1}]\!] \circ [\![\mathrm{GarbRem}(\mathrm{isos}(M_g), \mathrm{isos}(M'_g))]\!] \circ [\![\lfloor - \rfloor_A]\!] \\ &= [\![\lfloor - \rfloor_B^{-1}]\!] \circ g \circ [\![\lfloor - \rfloor_A]\!] \\ &= [\![\lfloor - \rfloor_B^{-1}]\!] \circ [\![\lfloor - \rfloor_B]\!] \circ f \circ [\![\lfloor - \rfloor_A^{-1}]\!] \circ [\![\lfloor - \rfloor_A]\!]\end{aligned}$$

By Prop. 4.31 we get that $[\![\lfloor - \rfloor_B^{-1}]\!] \circ [\![\lfloor - \rfloor_B]\!] = \mathrm{id}_B$ and $[\![\lfloor - \rfloor_A^{-1}]\!] \circ [\![\lfloor - \rfloor_A]\!] = \mathrm{id}_A$. Therefore $[\![e]\!] = f$. Since the left-hand-side of ω is just a variable we get $[\![\omega]\!] = [\![e]\!] \circ \mathrm{id}^{-1} = [\![e]\!] = f$. □

4.7 Further notes and conclusion

In this chapter, we have developed a functional, reversible programming language, based on [SVV18], with inductive types and general recursive calls. This language has been proven to have the same expressivity as a Turing Machine, meaning that any computable function can be performed. We have provided a mathematical interpretation of the language, based on the categorical presentation of inverse categories, equipped with:

- a join rig structure, to model pattern-matching and iso;
- an enrichment in $\mathbf{DCPO}$, to denote recursive calls through least fixed points;
- parameterised initial algebra, representing inductive data types;

for which the category of sets and partial injections, written $\mathbf{PInj}$, is a concrete model. This denotational semantics has been proven sound and adequacy with regard to the operational semantics of the language. The adequacy proof involves a *finitary* language, based on the original one, where the number of recursive calls is limited. This adequacy statement, together with Turing completeness, ensures that any computable partial injection between two types of the language, has a corresponding iso in the language.

The role of semantics. However abstract the development of a denotation semantics seems, it has a role to play in the formalisation of programming languages. In the original paper [SVV18], the language at the level of isos would only allow very specific λ-abstractions. This *ad hoc* presentation was a consequence of the authors having an operational understanding of the language only, without concerns for a denotation one. Once a denotational semantics was established by the author of this thesis, it was clear that any usual λ-calculus – and really any usual programming language – could be put on top of isos, for their semantics lives in the cartesian closed category $\mathbf{DCPO}$.

The aim of [SVV18] is to establish a high-order programming language that handles quantum control. While the ideas outlined in the paper are promising, the language in itself suffers from the same issue as observed above: the denotational understanding of the language is weak. This echoes to Abramsky's note in [Abr20], questioning whether denotational semantics should lead or follow. We do not argue in favour of one nor the other; however, we believe that in the presentation of a programming language, there should be both convincing operational and denotational arguments. This is what the author hopes he has done successfully all along the thesis.

Quantum control. While Chapter 3 presents an operational and a denotational account of simply-typed quantum control, a sound and adequate denotational semantics of the language in [SVV18] is yet to be found. With the development of the language in the current chapter, the desired result seems to be just around the corner. However, adding reversible quantum effects to the semantics presented here does not preserve fixed points (a more general intuition of this point is outlined in the next chapter), and thus an interpretation of recursive calls in that setting is yet to be found.

Chapter 5
Notes on Quantum Recursion

> "The category of Hilbert spaces is self-dual, has two monoidal structures, and its homsets are algebraic domains, but its enrichment and limit behaviour is wanting." — Chris Heunen, in [Heu13].

Abstract

We present some remarks and ideas on recursion in quantum control. First, we outline the limitations of Hilbert spaces as a denotational model of programming languages. Then, we present some results in the semantics of guarded recursion applied to quantum programming.

References. While the different contributions in this chapter are not yet enough to be published independently, they appear to the author as potential foundations for infinite-dimentional quantum-controlled programming. This work is the author's.

5.1 Introduction

In this chapter, we tackle the question of quantum recursion with quantum control – in other words, in the context of a quantum reversible effect. Let us recall that, by quantum control, we mean reversible quantum operations, which are usually unitaries between Hilbert spaces. It is yet unclear whether general recursion makes sense in that setting. For example, what is a non-terminating behaviour with unitaries?

While [SVV18] brings a syntactic approach to quantum control with infinite data types, we choose here a mathematical approach, through a (potential) denotational semantics. Our focus shall be on Hilbert spaces. They are the most natural candidates for a denotational semantics of quantum control, because their direct sum allows for quantum superposition.

We have seen in the previous chapters that unitaries can be written as a sum – or a *decomposition* – of contractions. Thus, contractions seem to be the right notion of *partial unitaries*. They can even be described as *subunitaries* thanks to [AMHK23, Proposition 14] which shows that a bounded linear map $f\colon H_1 \to H_2$ is contractive if and only if $f^\dagger f \sqsubseteq \mathrm{id}$.

5.2 Limitations

In this section, we present the *things that do not work* when working with Hilbert spaces and contractions to interpret quantum control. First, we recall the fact that the reversible quantum effect is not a monad, setting its study outside of the realm of Moggi. Then, we observe that the enrichment of Hilbert spaces is not sufficient to use them as a model such as the one in Chapter 4, where join inverse rig categories are $\mathbf{DCPO}$-enriched. Finally, we motivate the fact that the category of Hilbert spaces and contractions is probably not canonically traced, and therefore recursion could not studied with that angle.

5.2.1 Effects and the functor ℓ^2

As shown in [Heu13, Corollary 4.8], the functor ℓ^2 cannot induce a monad. Reversible quantum operations interpreted as maps between Hilbert spaces then cannot be seen as an effect in a traditional way [Mog89, Mog91], as laid out in §1.5.2 and studied in Chapter 2. We do not leave out the possibility of finding another category for which these operations form a monad, but the author thinks it is unlikely.

In any case, ℓ^2 is a functor from $\mathbf{PInj}$ to $\mathbf{Hilb}$, and one can show that a monad over $\mathbf{PInj}$ would not contain any interesting computational meaning. Given a monad $(\mathcal{M}, \eta, \mu)$ over $\mathbf{PInj}$, for all sets X, the multiplication μ_X is a monomorphism and an epimorphism, and therefore is an isomorphism between $\mathcal{M}^2X$ and $\mathcal{M}X$.

Instead, ℓ^2 can induce a *promonad* over $\mathbf{PInj}$, given by $\mathcal{P} \stackrel{\text{def}}{=} \mathbf{Hilb}(\ell^2(-), \ell^2(-)) \colon \mathbf{PInj}^{\text{op}} \times \mathbf{PInj} \to \mathbf{Set}$ (see [Hug00, JHH09, ASvW+05, Asa10] for a detail account on *arrows* – the programming language paradigm corresponding to promonads). Promonads in the context of reversible programming have been studied in details in [HKK18a]. The language in Chapter 3 can be seen as working directly with the promonad $\mathcal{P}$; however, it is unclear whether this point of view would improve or facilitate the presentation in that chapter.

5.2.2 Hilbert spaces are not properly enriched

As shown in [Heu13, Proposition 2.10], an order on bounded linear maps between Hilbert spaces can be established; however, this order is not preserved by composition, and thus does not form an enrichment. This is also true for the wide subcategories with isometries, or even contractions, as morphisms. This hints at the fact that fixed points in a reversible quantum setting cannot be studied in the same way as in Chapter 4. We are left to wonder how the *structurally recursive* fixed points introduced in [SVV18] can be interpreted in a proper, compositional semantics. While trying to answer this question, the author has come across the notion of *guarded recursion*, which led to the observations in §5.3.3.

5.2.3 Conjecture: infinite-dimensional Hilbert spaces are not canonically traced

It has been observed in [Bar14] that the computationally interesting tensor when working with isometries between Hilbert spaces is the direct sum $\oplus$, and not the tensor product $\otimes$. It

is shown in that paper that the category of *finite-dimensional* Hilbert spaces and isometries is traced over the direct sum $\oplus$. This would allow for a reversible quantum programming language managing finite data type to deal with finite loops, usual written `for` in programming languages. This trace is the canonical trace in the following sense: given an isometry $g\colon X \oplus U \to Y \oplus U$, which can be decomposed in blocks, giving

$$g = \begin{pmatrix} g_{X,Y} & g_{U,Y} \\ g_{X,U} & g_{U,U} \end{pmatrix}$$

the operator

$$\mathrm{Tr}_U^{X,Y}(g) = g_{X,Y} + \sum_{i=1}^{\infty} g_{U,Y} \circ g_{U,U}^{i} \circ g_{X,U} \tag{5.1}$$

is the trace given in [Bar14]. The existence of this trace depends on the Moore-Penrose generalised inverse [ABI03, SR20] of $\mathrm{id} - g_{U,U}$. What about non-finite dimensions? Pablo Andrés-Martínez [AM22, Section 3.3.4] raises this as an open question. We push the idea further by stating the following conjecture: the category of Hilbert spaces and contractions is not traced with the operator given in (5.1). The next lemma is a hint towards this conjecture.

Lemma 5.1. *Let f be the bounded linear map $\ell^2(\mathbb{N}) \to \ell^2(\mathbb{N})$ such that $f\,|n\rangle = \frac{1}{n+1}\,|n\rangle$. The map f is a contraction and does not admit a Moore-Penrose inverse.*

Proof. The Moore-Penrose inverse of f would be $g\colon \ell^2(\mathbb{N}) \to \ell^2(\mathbb{N})$ such that $g\,|n\rangle = (n+1)\,|n\rangle$, which is not bounded. □

We could then potentially provide a $g_{U,U}$ using the map given above, and obtain a suitable g, with the Szőkefalvi-Nagy ['sø:kɛfɒlvi 'nɒɟ] dilation theorem [SN54], which would not be traceable.

This raises more questions than it answers. One may wonder whether the category we are interested in in Chapter 3 – namely, countably-dimensional Hilbert spaces and isometries – is traced at all in a *computationally interesting* way. Making this statement mathematically precise is a challenge in itself; thus proving or disproving it might require some effort.

In his thesis, Pablo Andrés-Martínez [AM22] unveiled a category both traced with regard to its direct sum $\oplus$ and that could handle non-finite data types; however it has not been proven to be a model for a sufficiently expressive programming language. The work presented in the next section is similar in the idea, and has also not been shown to be a sound and adequate interpretation to a programming language yet. Nevertheless, it is based on a tried-and-texted paradigm for classical programming, called *guarded recursion*.

5.3 A Foundation for Guarded Quantum Recursion

Guarded recursion is a framework in which recursive calls are guarded by delay modalities. This framework is particularly useful to reason about streams in programming languages. A type system aimed for guarded recursion usually contains the *later* modality, given by the symbol $\blacktriangleright$. Its introduction rule is simple:

$$\frac{\Theta \vdash A}{\Theta \vdash \blacktriangleright A}$$

If we take the example of a guarded λ-calculus – such as the one introduced by Nakano [Nak00] – a contructor $\texttt{next}$ is added to the syntax, with the following rule:

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash \texttt{next}\ M : \blacktriangleright A}$$

A term under $\texttt{next}$ is *locked* and needs to wait for its time to be computed. The fixed point combinator introduced in the beginning of the thesis (see §1.2.1) becomes now $\texttt{fix} : (\blacktriangleright A \to A) \to A$, and given $\cdot \vdash M : \blacktriangleright A \to A$, we have the following operational rule:

$$\texttt{fix}\ M \to M(\texttt{next}\ \texttt{fix}\ M)$$

while does not allow for a infinite reduction in general, because the $\texttt{next}$ will control this behaviour.

This framework admits sound and adequate denotational semantics in the *topos of trees* $\mathbf{Set}^{\mathbb{N}^{op}}$, written $\mathbf{S}$ in the following, which is a cartesian closed category.

5.3.1 Work of the author

The author has contributed to the following points.

- The definition of categories ($\mathbf{N}$ and $\mathbf{Q}$, see below) to model guarded quantum recursion.
- A proof that those categories are $\mathbf{S}$-enriched (see Lemma 5.5), allowing for a similar semantic study as what is done in Chapter 4.
- A proof that inductive types can be interpreted in this model (see Theorem 5.32).
- Convincing arguments for this model to interpret a programming language with quantum guarded recursion.

5.3.2 Related work

The work in [BMSS12] can be described as a foundation for semantics of guarded recursion, and an introduction to what they call *synthetic guarded domain theory*. Their theory assumes a topos or a sheaves structure, which will not be the case in an attempt to interpret quantum control. However, their work is a great source of inspiration for our denotation of guarded inductive types. They prove that locally contractive functors have a unique fixed point, which shows that induction and coinduction have the same interpretation in their model; this has a practical consequence: the fixed point is obtained as a limit, but can still be manipulated as an initial algebra.

A first account of solution to solve recursive equations in (pre)sheaves is given in [DGM04], and the same authors also observed that working with contractive functors would unify induction and coinduction [DGM03]. Their theory is not suited to quantum for the same reason as above.

The work in [BSS10] details a denotational semantics to the guarded λ-calculus introduced by Nakano in [Nak00]. Their model is a category of ultrametric spaces, which is actually included in the theory of [BMSS12], as justified by the authors themselves.

Several papers have then used synthetic guarded domain theory to develop refined models of guarded types or guarded recursion [CBGB16, MMV20, BR23]. Their study, based on [BMSS12], cannot be used to model quantum computation, for the same reasons as above.

5.3.3 Contribution

This section aims at providing a categorical tool necessary to interpret guarded recursion with quantum control. Recursive domain equations are here tackled through locally contractive functors as defined in [BMSS12].

We will work with categories of the form $\mathbf{C}^{\mathbb{N}^{op}}$, where $\mathbb{N}$ is the category of natural numbers starting from 0, with morphisms defined by the classical order on natural numbers. Given an object X of $\mathbf{C}^{\mathbb{N}^{op}}$, meaning a functor $\mathbb{N}^{op} \to \mathbf{C}$, its image on the morphism $n \leq n+1$ is written $r_n^X : X(n+1) \to X(n)$, and is a morphism in $\mathbf{C}$. This object X of $\mathbf{C}^{\mathbb{N}^{op}}$ can be represented with a diagram, such as:

$$X(0) \xleftarrow{r_0^X} X(1) \xleftarrow{r_1^X} X(2) \xleftarrow{r_2^X} X(3) \longleftarrow \cdots \tag{5.2}$$

and a morphism $f : X \to Y$ can be pictured with the following diagram in $\mathbf{C}$:

$$\begin{array}{ccccccccc} X(0) & \xleftarrow{r_0^X} & X(1) & \xleftarrow{r_1^X} & X(2) & \xleftarrow{r_2^X} & X(3) & \longleftarrow & \cdots \\ \downarrow{\scriptstyle f_0} & & \downarrow{\scriptstyle f_1} & & \downarrow{\scriptstyle f_2} & & \downarrow{\scriptstyle f_3} & & \\ Y(0) & \xleftarrow[r_0^Y]{} & Y(1) & \xleftarrow[r_1^Y]{} & Y(2) & \xleftarrow[r_2^Y]{} & Y(3) & \longleftarrow & \cdots \end{array}$$

This way of picturing them will help the intuition throughout the section. The category $\mathbf{Set}^{\mathbb{N}^{op}}$ is called the *topos of trees*, and is a cartesian closed category. However, $\mathbf{C}^{\mathbb{N}^{op}}$ in general has a topos flavour without being cartesian closed. Some subcategories of $\mathbf{Coiso}^{\mathbb{N}^{op}}$ and $\mathbf{Contr}^{\mathbb{N}^{op}}$ will be used for our denotational model; a coisometry is in particular a contraction, thus $\mathbf{Coiso}^{\mathbb{N}^{op}}$ is a subcategory of $\mathbf{Contr}^{\mathbb{N}^{op}}$. This model was inspired by the work in [BMSS12] and [CBGB16]; however, the motivations, the point of view and the results differ. We use the following notations: $\mathbf{S} \stackrel{\text{def}}{=} \mathbf{Set}^{\mathbb{N}^{op}}$, $\mathbf{N} \stackrel{\text{def}}{=} \mathbf{Coiso}^{\mathbb{N}^{op}}$.

Lemma 5.2 ([MM12]). *The category $\mathbf{S}$ is a cartesian closed category with the following: given two objects X, Y in $\mathbf{S}$, their product $X \times Y$ is obtained pointwise, and $[X \to Y]$ is given by $\mathbf{S}(よ(-) \times X, Y)$ where $よ : \mathbb{N} \to \mathbf{S}$ is the Yoneda embedding ($よ$ is the Japanese hiragana "yo" , see Example 1.19).*

We write $\mathbf{Q}$, the full subcategory of $\mathbf{Contr}^{\mathbb{N}^{op}}$ whose objects are objects in $\mathbf{N}$. Note that $\mathbf{N}$ is embedded in $\mathbf{Q}$. The category $\mathbf{N}$ will be used to study the semantics of types, whereas $\mathbf{Q}$ is the category where the terms and the functions are interpreted.

Lemma 5.3. *The categories $\mathbf{Q}$ and $\mathbf{N}$ are symmetric monoidal, equipped with a pointwise monoidal product. More generally, if $\mathbf{C}$ is symmetric monoidal, so is $\mathbf{C}^{\mathbb{N}^{op}}$.*

Proof. Let $(\mathbf{C}, \otimes, I, \alpha, \lambda, \rho)$ be a symmetric monoidal category. We show that $\mathbf{C}^{\mathbb{N}^{op}}$ is also one. The tensor is obtained pointwise: given two objects X and Y in $\mathbf{C}^{\mathbb{N}^{op}}$, their tensor product is the following object:

$$X(0) \otimes Y(0) \xleftarrow{r_0^X \otimes r_0^Y} X(1) \otimes Y(1) \xleftarrow{r_1^X \otimes r_1^Y} X(2) \otimes Y(2) \xleftarrow{r_2^X \otimes r_2^Y} X(3) \otimes Y(3) \longleftarrow \cdots$$

We abuse notations and write $X \otimes Y$ for this tensor product. The tensor unit of this tensor is the object:

$$I \xleftarrow{\mathrm{id}_I} I \xleftarrow{\mathrm{id}_I} I \xleftarrow{\mathrm{id}_I} I \longleftarrow \cdots$$

We abuse notations by also writing I for this unit. The left unitor $I \otimes X \to X$ is given pointwise as well:

$$\begin{array}{ccccccccc}
I \otimes X(0) & \xleftarrow{\mathrm{id}_I \otimes r_0^X} & I \otimes X(1) & \xleftarrow{\mathrm{id}_I \otimes r_1^X} & I \otimes X(2) & \xleftarrow{\mathrm{id}_I \otimes r_2^X} & I \otimes X(3) & \longleftarrow & \cdots \\
\downarrow{\scriptstyle \lambda_{X(0)}} & & \downarrow{\scriptstyle \lambda_{X(1)}} & & \downarrow{\scriptstyle \lambda_{X(2)}} & & \downarrow{\scriptstyle \lambda_{X(3)}} & & \\
X(0) & \xleftarrow{r_0^X} & X(1) & \xleftarrow{r_1^X} & X(2) & \xleftarrow{r_2^X} & X(3) & \longleftarrow & \cdots
\end{array}$$

The right unitor and associator are defined pointwise in the same way. The proof of the coherence diagrams is direct. □

Similarly, given two objects X, Y of $\mathbf{Q}$ (resp. $\mathbf{N}$), their pointwise direct sum $X \oplus Y$ is an object of $\mathbf{Q}$ (resp. $\mathbf{N}$).

$$X(0) \oplus Y(0) \xleftarrow{r_0^X \oplus r_0^Y} X(1) \oplus Y(1) \xleftarrow{r_1^X \oplus r_1^Y} X(2) \oplus Y(2) \xleftarrow{r_2^X \oplus r_2^Y} X(3) \oplus Y(3) \longleftarrow \cdots$$

Moreover, in $\mathbf{Q}$, one can define injections $\iota_l^{X,Y} : X \to X \oplus Y$ as follows:

$$\begin{array}{ccccccccc}
X(0) & \xleftarrow{r_0^X} & X(1) & \xleftarrow{r_1^X} & X(2) & \xleftarrow{r_2^X} & X(3) & \longleftarrow & \cdots \\
\downarrow{\scriptstyle \iota_l^{X(0),Y(0)}} & & \downarrow{\scriptstyle \iota_l^{X(1),Y(1)}} & & \downarrow{\scriptstyle \iota_l^{X(2),Y(2)}} & & \downarrow{\scriptstyle \iota_l^{X(3),Y(3)}} & & \\
X(0) \oplus Y(0) & \xleftarrow[r_0^X \oplus r_0^Y]{} & X(1) \oplus Y(1) & \xleftarrow[r_1^X \oplus r_1^Y]{} & X(2) \oplus Y(2) & \xleftarrow[r_2^X \oplus r_2^Y]{} & X(3) \oplus Y(3) & \longleftarrow & \cdots
\end{array}$$

and $\iota_r^{X,Y} : Y \to X \oplus Y$ is defined similarly.

Remark 5.4 (Dagger). Morphisms in $\mathbf{Q}$ are natural transformations, whose components are morphisms in $\mathbf{Contr}$. In that regard, $\mathbf{Q}$ inherits some of the structure of $\mathbf{Contr}$, but not all. For example, $\mathbf{Q}$ is not a dagger category. However, given $\alpha \colon X \Rightarrow Y$ a morphism in $\mathbf{Q}$, we will write $\alpha^\dagger$ for the *componentwise* dagger of α, even if it might not be a morphism in $\mathbf{Q}$.

Lemma 5.5. *The categories* $\mathbf{Q}$ *and* $\mathbf{N}$ *are enriched in* $\mathbf{S}$.

Proof. The homsets $\mathbf{N}(X, Y)$ can be seen as objects of $\mathbf{S}$, with $\mathbf{N}(X, Y)(n)$ being the set of n-th component of natural transformations in $\mathbf{N}(X, Y)$, and the image of $n \leq n + 1$, written $r_n^{\mathbf{N}(X,Y)}$, is:

$$\mathbf{N}(X, Y)(n) \xleftarrow{r_n^Y \circ - \circ (r_n^X)^\dagger} \mathbf{N}(X, Y)(n + 1).$$

Note that elements of $\mathbf{N}(X, Y)(n)$ are in particular in $\mathbf{Coiso}(X(n), Y(n))$. This definition is sound because if α is a natural transformation, we have in particular that $\alpha_n \circ r_n^X = r_n^Y \circ \alpha_{n+1}$, and by precomposing by $(r_n^X)^\dagger$, we get

$$\begin{aligned}
r_n^Y \circ \alpha_{n+1} \circ (r_n^X)^\dagger &= \alpha_n \circ r_n^X \circ (r_n^X)^\dagger \\
&= \alpha_n.
\end{aligned}$$

We also have to prove that composition, defined pointwise by the composition in $\mathbf{Coiso}$, is a morphism in $\mathbf{S}$; formally that for every X, Y, Z objects of $\mathbf{N}$, $\mathrm{comp}_{X,Y,Z} \colon \mathbf{N}(X,Y) \times \mathbf{N}(Y,Z) \to \mathbf{N}(X,Z)$ is a morphism in $\mathbf{S}$. We need to prove that it is a natural transformation, in other words that for all n, the diagram:

$$\begin{array}{ccc} \mathbf{N}(X,Y)(n) \times \mathbf{N}(Y,Z)(n) & \xleftarrow{r_n^{\mathbf{N}(X,Y)} \times r_n^{\mathbf{N}(Y,Z)}} & \mathbf{N}(X,Y)(n+1) \times \mathbf{N}(Y,Z)(n+1) \\ \downarrow{\scriptstyle \mathrm{comp}_n} & & \downarrow{\scriptstyle \mathrm{comp}_{n+1}} \\ \mathbf{N}(X,Z)(n) & \xleftarrow[r_n^{\mathbf{N}(X,Z)}]{} & \mathbf{N}(X,Z)(n+1) \end{array}$$

commutes. Indeed:

$$\begin{aligned} (\mathrm{comp}_n \circ (r_n^{\mathbf{N}(X,Y)} \times r_n^{\mathbf{N}(Y,Z)}))(f_{n+1}, g_{n+1}) &= \mathrm{comp}_n(r_n^Y \circ f_{n+1} \circ (r_n^X)^\dagger, r_n^Z \circ g_{n+1} \circ (r_n^Y)^\dagger) \\ &= \mathrm{comp}_n(f_n, g_n) = g_n \circ f_n, \end{aligned}$$

and

$$\begin{aligned} (r_n^{\mathbf{N}(X,Z)} \circ \mathrm{comp}_{n+1})(f_{n+1}, g_{n+1}) &= r_n^{\mathbf{N}(X,Z)}(g_{n+1} \circ f_{n+1}) \\ &= r_n^Z \circ g_{n+1} \circ f_{n+1} \circ (r_n^X)^\dagger \\ &= g_n \circ r_n^Y \circ f_{n+1} \circ (r_n^X)^\dagger \\ &= g_n \circ f_n \circ r_n^X \circ (r_n^X)^\dagger \\ &= g_n \circ f_n, \end{aligned}$$

which ensures the commutativity of the diagram above.

The same observations apply to $\mathbf{Q}$. □

Remark 5.6. The embedding $E_{\mathbf{N}}^{\mathbf{Q}} \colon \mathbf{N} \hookrightarrow \mathbf{Q}$ is $\mathbf{S}$-enriched.

A feature of categories of the form $\mathbf{C}^{\mathbb{N}^{op}}$ is the later functor. This functor works as some sort of delay operation. It can be used to keep track of the depth of a term and the number of recursive calls. The way it works is fairly simple: it shifts the diagram in (5.2) one step to the right, and adds a terminal object on the left.

Definition 5.7 (Later functor)**.** Given any category $\mathbf{C}$ with terminal object 1, The later functor is a functor $L \colon \mathbf{C}^{\mathbb{N}^{op}} \to \mathbf{C}^{\mathbb{N}^{op}}$, such that given a functor $X \colon \mathbb{N}^{op} \to \mathbf{C}$, $LX(0) = 1$ and $LX(n+1) = X(n)$. Given $\alpha \colon X \Rightarrow Y$, $L\alpha_0 = !$ (the terminal map), and $(L\alpha)_{n+1} = \alpha_n$.

Remark 5.8. Note that this endofunctor can be defined on $\mathbf{S}$, $\mathbf{Q}$ and $\mathbf{N}$; where the terminal object in the latter categories is the zero-dimentional Hilbert space $\{0\}$. We will use the same letter L when it is not ambiguous. If any ambiguity arises, the notations $L^{\mathbf{S}}$, $L^{\mathbf{Q}}$ and $L^{\mathbf{N}}$ will be used.

Lemma 5.9 ([BMSS12])**.** *The functor $L^{\mathbf{S}} \colon \mathbf{S} \to \mathbf{S}$ is a strict monoidal functor.*

Remark 5.10. The functors $L^{\mathbf{N}}$ and $L^{\mathbf{Q}}$ preserve the monoidal structure in the sense that $L(X \otimes Y) = LX \otimes LY$, however L does not map the tensor unit to the tensor unit in those categories.

Lemma 5.11. *Given X, Y objects in $\mathbf{N}$, we have $L^{\mathbf{S}}\mathbf{N}(X,Y) \cong \mathbf{N}(L^{\mathbf{N}}X, L^{\mathbf{N}}Y)$.*

Proof. Let X and Y be two objects in $\mathbf{N}$. Remember that the homset $\mathbf{N}(X,Y)$ can be seen as an object of $\mathbf{S}$, with $\mathbf{N}(X,Y)(n)$ being the set of n-th component of natural transformations in $\mathbf{N}(X,Y)$, and the image of $n \leq n+1$, written $r_n^{\mathbf{N}(X,Y)}$, is:

$$\mathbf{N}(X,Y)(n) \xleftarrow{r_n^Y \circ - \circ (r_n^X)^\dagger} \mathbf{N}(X,Y)(n+1).$$

We have $L^{\mathbf{S}}\mathbf{N}(X,Y) = \{*\}$ and $\mathbf{N}(L^{\mathbf{N}}X, L^{\mathbf{N}}Y) = \{0_{\{0\}\to\{0\}}\}$. They are both singletons. Moreover, $L^{\mathbf{S}}\mathbf{N}(X,Y)_{n+1} = \mathbf{N}(X,Y)_n$ is the set of n-th components of natural transformations from X to Y, which is exactly like $\mathbf{N}(L^{\mathbf{N}}X, L^{\mathbf{N}}Y)_{n+1}$. Note also that the functor L does not change the morphisms image of $n \leq n+1$, except shifting them. Therefore, $L^{\mathbf{S}}\mathbf{N}(X,Y) \cong \mathbf{N}(L^{\mathbf{N}}X, L^{\mathbf{N}}Y)$. □

The delay embodied by the functor L can be introduced by a natural transformation, called *next*. This natural transformation helps us introduce the delay in a programming language, as the denotational semantics of a delayed program.

Definition 5.12 (Next)**.** Given any category $\mathbf{C}$ with terminal object 1, and $L\colon \mathbf{C}^{\mathbb{N}^{op}} \to \mathbf{C}^{\mathbb{N}^{op}}$ the later functor, there is a natural transformation $\nu\colon id \Rightarrow L$, defined as $\nu_{X,0} = !$ and $\nu_{X,n+1} = r_n^X$. Or as a diagram, it maps the functor X to the functor LX as follows:

$$\begin{array}{ccccccccc}
X(0) & \xleftarrow{r_0^X} & X(1) & \xleftarrow{r_1^X} & X(2) & \xleftarrow{r_2^X} & X(3) & \longleftarrow & \cdots \\
{\scriptstyle !}\downarrow & & \downarrow{\scriptstyle r_0^X} & & \downarrow{\scriptstyle r_1^X} & & \downarrow{\scriptstyle r_2^X} & & \\
1 & \xleftarrow[!]{} & X(0) & \xleftarrow[r_0^X]{} & X(1) & \xleftarrow[r_1^X]{} & X(2) & \longleftarrow & \cdots
\end{array}$$

Which gives, in the categories $\mathbf{Q}$ and $\mathbf{N}$:

$$\begin{array}{ccccccccc}
X(0) & \xleftarrow{r_0^X} & X(1) & \xleftarrow{r_1^X} & X(2) & \xleftarrow{r_2^X} & X(3) & \longleftarrow & \cdots \\
{\scriptstyle 0}\downarrow & & \downarrow{\scriptstyle r_0^X} & & \downarrow{\scriptstyle r_1^X} & & \downarrow{\scriptstyle r_2^X} & & \\
\{0\} & \xleftarrow[0]{} & X(0) & \xleftarrow[r_0^X]{} & X(1) & \xleftarrow[r_1^X]{} & X(2) & \longleftarrow & \cdots
\end{array}$$

Remark 5.13. Note that ν is a *high-level* natural transformation. It is an informal way to say that it is a natural transformation at the level of $\mathbf{C}^{\mathbb{N}^{op}}$. For all functors $X\colon \mathbb{N}^{op} \to \mathbf{C}$, ν_X is a morphism in $\mathbf{C}^{\mathbb{N}^{op}}$. A morphism in $\mathbf{C}^{\mathbb{N}^{op}}$ is a natural transformation – this time, *low-level* – at the level of $\mathbf{C}$. Thus, if n is a natural number, then $\nu_{X,n}$ is a morphism in $\mathbf{C}$.

Once again, and similarly to the functor L, the natural transformation ν can be defined in $\mathbf{S}$ as well as in $\mathbf{N}$ and $\mathbf{Q}$. If any ambiguity arises, the notations $\nu^{\mathbf{S}}$, $\nu^{\mathbf{Q}}$ and $\nu^{\mathbf{N}}$ will be used. Observe that the components of $\nu^{\mathbf{Q}}$ and $\nu^{\mathbf{N}}$ are the same.

A direct consequence of Lemma 5.5 is the possibility to move back and forth from $\mathbf{N}$ to its enrichment $\mathbf{S}$ with the natural transformation ν.

Lemma 5.14. *Given X, Y two objects of $\mathbf{N}$, we have $\nu^{\mathbf{S}}_{\mathbf{N}(X,Y)} = \nu^{\mathbf{N}}_Y \circ - \circ \left(\nu^{\mathbf{N}}_X\right)^\dagger$.*

Proof. This follows from Lemma 5.5, Lemma 5.11 and Definition 5.12. □

Lemmas 5.11 and 5.14 also allow for the following observation.

Corollary 5.15. *The functor $L^{\mathbf{N}}$ is $\mathbf{S}$-enriched.*

Note that, as explained in Remark 5.4, $\left(\nu^{\mathbf{N}}_X\right)^\dagger$ is not a natural transformation, and thus the notation above is loose. However, it can be used in some cases, as shown by the next lemma.

Lemma 5.16. *Given a morphism $f\colon X \to Y$ in $\mathbf{Q}$, we have that $\nu^{\mathbf{Q}}_Y \circ f \circ \left(\nu^{\mathbf{Q}}_X\right)^\dagger : LX \to LY$ is a morphism in $\mathbf{Q}$.*

Proof. Remember that ν is defined as $\nu_{X,n} = r^X_{n-1}$. Let us proceed:

$$\begin{aligned} r^Y_n \circ f_{n+1} \circ (r^X_n)^\dagger \circ r^X_n &= f_n \circ r^X_n \circ (r^X_n)^\dagger \circ r^X_n \\ &= f_n \circ r^X_n \\ &= r^Y_n \circ f_{n+1} \\ &= r^Y_n \circ f_{n+1} \circ r^X_{n+1} \circ (r^X_{n+1})^\dagger \\ &= r^Y_n \circ r^Y_{n+1} \circ f_{n+2} \circ (r^X_{n+1})^\dagger. \end{aligned}$$

□

To interpret inductive types, we need the help of fixed points to solve recursive domain equations. First, we recall some notions on contractive morphisms in $\mathbf{S}$, introduced in [BMSS12].

Definition 5.17 ([BMSS12])**.** A morphism $f\colon X \to Y$ in $\mathbf{S}$ is contractive if there exists a morphism $g\colon LX \to Y$ such that $f = g \circ \nu_X$. A morphism $f\colon X \times Y \to Z$ is contractive in the first variable if there exists $g\colon LX \times Y \to Z$ such that $f = g \circ (\nu_X \times \mathrm{id}_Y)$.

Lemma 5.18 ([BMSS12])**.** *The following assertions hold.*

- *Given $f\colon X \to Y$, $g\colon Y \to Z$, if either f or g is contractive, then gf is contractive.*
- *Given $f\colon X \to Y$ and $g\colon X' \to Y'$ contractive, so is $f \times g\colon X \times X' \to Y \to Y'$.*
- *A morphism $h\colon X \times Y \to Z$ is contractive in the first variable iff $\mathrm{curry}(h)\colon X \to Z^Y$ is contractive.*

Theorem 5.19 ([BMSS12])**.** *There exists a natural family of morphisms $\mathrm{fix}_X\colon (LX \to X) \to X$ which computes unique fixed points: given $f\colon X \times Y \to X$ contractive in the first variable and $g\colon LX \times Y \to X$ the resulting morphism, then $\mathrm{fix}_X \circ \mathrm{curry}(g)$ is the unique $h\colon Y \to X$ such that $f \circ \langle h, \mathrm{id}_Y \rangle = h$.*

Remark 5.20. Unsurprisingly, the previous theorem can be used for the interpretation of a fixed point combinator.

The definition of a contractive morphism in $\mathbf{S}$ help define functors that have a fixed point in $\mathbf{N}$. We precise what we mean by fixed point of a functor.

Definition 5.21 (Fixed Point)**.** A fixed point of an endofunctor $T\colon \mathbf{N} \to \mathbf{N}$ is a pair $(X, \alpha\colon TX \to X)$ such that α is an isomorphism.

We define locally contractive functors, which are functor that admit a fixed points. We will see that, as its name suggests, a locally contractive functor has a unique fixed point, up to isomorphism.

Definition 5.22 (Locally Contractive functor)**.** An $\mathbf{S}$-functor $F\colon \mathbf{N} \to \mathbf{N}$ is said to be *locally contractive* if its morphism mapping $F_{X,Y}\colon \mathbf{N}(X,Y) \to \mathbf{N}(FX,FY)$ is contractive; meaning it factorises through ν: for all X, Y there is a morphism mapping $G_{X,Y}\colon L(\mathbf{N}(X,Y)) \to \mathbf{N}(FX,FY)$ such that:

$$\begin{array}{ccc} \mathbf{N}(X,Y) & \xrightarrow{\quad F_{X,Y} \quad} & \mathbf{N}(FX,FY) \\ & {\scriptstyle \nu_{\mathbf{N}(X,Y)}} \searrow \quad \nearrow {\scriptstyle G_{X,Y}} & \\ & L(\mathbf{N}(X,Y)) & \end{array}$$

and such that G behaves *like a functor*; formally, such that the diagrams:

$$\begin{array}{ccccc} L\mathbf{N}(Y,Z) \times L\mathbf{N}(X,Y) & \xrightarrow{\cong} & L(\mathbf{N}(Y,Z) \times \mathbf{N}(X,Y)) & \xrightarrow{L(\mathrm{comp})} & L\mathbf{N}(X,Z) \\ \downarrow{\scriptstyle G_{X,Y} \times G_{Y,Z}} & & & & \downarrow{\scriptstyle G_{X,Z}} \\ \mathbf{N}(FY,FZ) \times \mathbf{N}(FX,FY) & & \xrightarrow{\qquad\qquad \mathrm{comp} \qquad\qquad} & & \mathbf{N}(FX,FZ) \end{array}$$

$$\begin{array}{ccc} \{\star\} & \xrightarrow{L(\mathrm{id})} & L\mathbf{N}(X,X) \\ & {\scriptstyle \mathrm{id}} \searrow & \downarrow{\scriptstyle G_{X,X}} \\ & & \mathbf{N}(FX,FX) \end{array}$$

commute in $\mathbf{S}$, for all objects X, Y, Z.

Remark 5.23. The definition above was inspired by a similar one in [BMSS12], where they use the fact that their category is closed to draw the diagrams with objects and morphisms of the category. We manage to do the same with the enrichment of $\mathbf{N}$ in $\mathbf{S}$.

There are some direct examples of locally contractive functors.

Lemma 5.24. *The functor $L^{\mathbf{N}}$ is locally contractive.*

Proof. This is a direct conclusion of Lemma 5.11 and Definition 5.22. □

Lemma 5.25 ([BMSS12]). *Given $F, G\colon \mathbf{N} \to \mathbf{N}$ two $\mathbf{S}$-enriched functors and such that either F or G is locally contractive, then FG is locally contractive.*

Remark 5.26 (Initial Algebra and Final Coalgebra). Given a locally contractive endofunctor $F\colon \mathbf{N} \to \mathbf{N}$, a fixed point of F is an initial algebra and a final coalgebra. Indeed, given a fixed point $\alpha\colon FX \cong X$ and an algebra $\beta\colon FY \Rightarrow Y$, an algebra morphism $\gamma\colon X \Rightarrow Y$ is given by a fixed point of $H\colon \gamma \mapsto \beta \circ F_{X,Y}(\gamma) \circ \alpha^{-1}$. Note that H is then a contractive morphism in $\mathbf{S}$, because F is locally contractive. It is proven in [BMSS12] that such a morphism has a unique fixed point. Thus the algebra morphism γ is unique; and this makes α an initial algebra. The proof of the coalgebra part is similar.

Remark 5.27. This basically means that induction and coinduction in our system are the same. This observation was made long before us (see related work §5.3.2). In this setting, it is a matter a choice whether the syntax should use μ or ν as a notation for fixed points. Our focus is on inductive data types, we then use μ.

Given a morphism $\alpha\colon X \to Y$ in $\mathbf{N}$, one says that α is an n-isomorphism if the first n components of α (that is to say, $\alpha_0, \ldots, \alpha_{n-1}$) are isomorphisms.

Lemma 5.28. *A locally contractive functor $F\colon \mathbf{N} \to \mathbf{N}$ maps an n-isomorphism to an $n+1$-isomorphism.*

Proof. This is the purpose of Definition 5.22. The proof is direct by observing that L shifts all components one step to the right. □

This observation allows to prove the next theorem, with the same proof strategy as in [BMSS12].

Theorem 5.29. *Any locally contractive endofunctor $T\colon \mathbf{N} \to \mathbf{N}$ has a fixed point.*

Proof. Note that the category $\mathbf{N}$ has a terminal object, written Z:

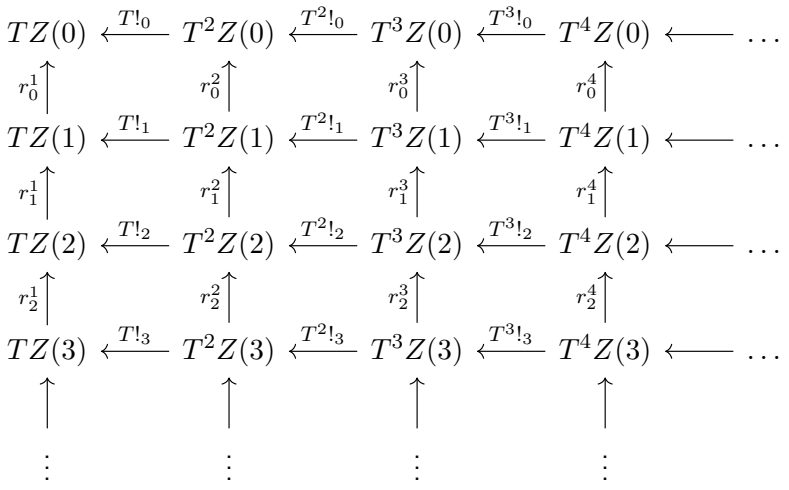
$$\{0\} \xleftarrow{0} \{0\} \xleftarrow{0} \{0\} \longleftarrow \ldots$$

and we will write ! the unique map to Z. Given a functor $T\colon \mathbf{N} \to \mathbf{N}$, let us have a look at the sequence:

$$TZ \xleftarrow{T!} T^2Z \xleftarrow{T^2!} T^3Z \xleftarrow{T^3!} T^4Z \longleftarrow \ldots \tag{5.3}$$

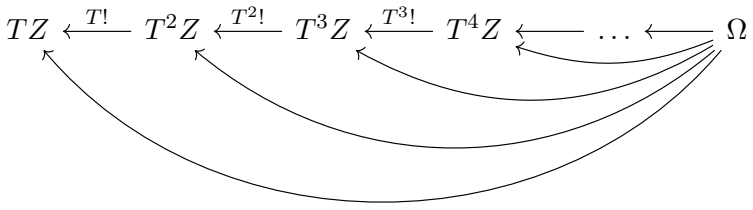
and we obtain the fixed point of T through the limit of the above diagram. The limit can be

built by observing the following diagram:



which is simply Diagram (5.3) expended with the diagram view of objects of $\mathbf{N}$, see Diagram (5.2). Let us consider the object of $\mathbf{N}$ made of the diagonal elements of the last diagram above, and we call this new object Ω. Its image on objects is $\Omega(n) = T^{n+1}Z(n)$ and its image on morphisms can be read on the diagram (it does not matter which one is chosen because the diagram commutes).

Moreover, Lemma 5.28 ensures that all $T^n!_k$, where $k < n$, is an isomorphism; so they can be soundly reversed, (in other words, having those arrows in the other direction does not break the commutativity of the diagram) which gives that:



commutes. This new object Ω is a limit of Diagram (5.3): it is made of elements of the diagram, thus any object that is mapped to the diagram has a single way to be mapped to Ω. For the same reason, $T\Omega$ is a limit of the diagram as well, thus $\Omega \cong T\Omega$. Note that this can all be viewed as a consequence of Lemma 5.28. □

Remark 5.30. The category $\mathbf{N}$ is *contractively complete* in the sense of [BMSS12].

We generalise the notion of locally contractive endofunctors to functors $\mathbf{N}^k \to \mathbf{N}$, to facilitate the discussion to come.

Definition 5.31. An $\mathbf{S}$-functor $T\colon \mathbf{N}^k \to \mathbf{N}$ is *locally contractive* if it is serarately locally contractive in each variable; formally, given any vector $\overrightarrow{F}(-)$ of objects of $\mathbf{N}$ with one hole (*e.g.*, $\overrightarrow{F}(-) = F_1, \ldots, F_{j-1}, -, F_{j+1}, \ldots, F_k$), the functor $T(\overrightarrow{F}(-))\colon \mathbf{N} \to \mathbf{N}$ is locally contractive.

Theorem 5.32 (Parameterised Fixed Point). *A locally contractive functor $T\colon \mathbf{N}^{k+1} \to \mathbf{N}$ admits a parameterised fixed point; in details, a pair $(T^{\dagger}, \phi^T)$ such that:*

- *$T^{\dagger}\colon \mathbf{N}^k \to \mathbf{N}$ is a locally contractive functor,*
- *$\phi^T : T \circ \langle \mathrm{id}, T^{\dagger} \rangle \Rightarrow T^{\dagger}$ is a natural isomorphism,*
- *for every object $\overrightarrow{F}$ in $\mathbf{N}^k$, $(T^{\dagger}\overrightarrow{F}, \phi^T)$ is the fixed point of $T(\overrightarrow{F}, -)$.*

Proof. The proof is the same as the one of [BMSS12, Theorem 7.5]. Given $\overrightarrow{F}$ an object in $\mathbf{N}^k$, $T(\overrightarrow{F}, -)\colon \mathbf{N} \to \mathbf{N}$ is a contractive functor, and thus has a fixed point $(\Omega(\overrightarrow{F}), \alpha^{\overrightarrow{F}})$. The next step is to prove that the statement $\Omega(-)$ induces a functor $\mathbf{N}^k \to \mathbf{N}$. Remember that $\Omega(\overrightarrow{F})(n) = T(\overrightarrow{F}, -)^{n+1}Z(n)$, which has a functor flavour; given $\beta\colon \overrightarrow{F} \Rightarrow \overrightarrow{G}$, $\Omega(\beta)_n = (T(\beta, -)^{n+1}Z)_n$ makes $\Omega(-)$ a functor (it preserves the identity and composition because T does). Also, in the formula, T is applied at least once and is locally contractive, thus Ω is locally contractive. The natural transformation $T \circ \langle \mathrm{id}, \Omega \rangle \Rightarrow \Omega$ is obtained by looking at the square diagram in the first part of the proof; its isomorphic nature is inherited from Lemma 5.28. □

Note that a parameterised fixed point provides a natural isomorphism, whose components are also isomorphisms. These components are coisometries, and an isomorphic coisometry is a unitary.

As a denotational semantics. Contractive functors and contractive morphisms would respectively be used to interpret inductive data types and guarded recursion functions in a programming language. We can imagine a language akin to the one in Chapter 4 with quantum superpositions, with the introduction of the modality ▶ and the combinator `next`. The interpretation of a type judgement $\Theta \vdash A$ is given by a functor $\mathbf{N}^{|\Theta|} \to \mathbf{N}$, and the interpretation of $\Theta \vdash \mu X.A$ is given by $[\![\Theta, X \vdash A]\!]^{\dagger}$ thanks to Theorem 5.32. The fixed point combinator would have the following typing rule:

$$\frac{\Psi, \phi\colon \blacktriangleright T \vdash_{\omega} \omega\colon T}{\Psi \vdash_{\omega} \mathtt{fix}\ \phi.\omega\colon T}$$

which ensures that the interpretation $[\![\Psi, \phi\colon \blacktriangleright T \vdash_{\omega} \omega\colon T]\!]$ is contractive (see Definition 5.17) and therefore admits a unique fixed point (see Theorem 5.19).

5.3.4 Conclusion on Guarded Quantum Recursion

We have laid out convincing grounds for guarded quantum recursion based on its denotational semantics in $\mathbf{N}$ and $\mathbf{Q}$, two categories enriched in the topos of trees $\mathbf{S}$, therefore allowing us to extract properties of the latter for the interpretation of the functions in the language.

The details on the corresponding programming language capturing both quantum control as an algebraic effect and guarded recursion is left as future work.

Conclusion

"What you say does not matter, it is what people remember that matters." — Benoît Valiron.

In this thesis, we are concerned with the question of the semantics of effects in programming languages. This study is conducted through the prism of formal programming languages, which are directly a λ-calculus or inspired by it. We have put forward new perspectives on various effects and their commutativity via the question of centrality, and we have thouroughly studied a reversible algebraic effect aimed at quantum computing, allowing for a quantum control of the program flow. We have also questioned the semantics of inductive types and recursion in the context of reversibility, to potentially adapt it to the quantum case.

In Chapter 2, we have studied the question of centrality for *monads*. In particular, we have provided three equivalent conditions for a monad to be centralisable. Monads have been shown to be the right structure to model effects in category theory. However, some effects cannot be captured as a monad, such as the reversible effect we focus on in Chapter 3. The question of centrality of effects should then be generalised to *promonads*, which are to monads what relations are to functions. Moreover, commutativity can be studied more broadly than with centres, and a whole theory of *centralisers* might be developed.

In Chapter 3, we have laid foundations for the semantics of quantum computing seen as a reversible effect. This point of view allows us to work with a *quantum control flow*. This is especially meaningful since some quantum-controlled operations, such as the quantum switch, cannot be performed with classical control. Once this simply-typed quantum control and its semantics are properly presented, we have wondered about possible additions to the language, such as infinite data types and recursion.

To do so, we have first studied the question of infinite data types and recursion in reversible computation, in Chapter 4. In particular, we have provided a sound and adequate denotational semantics of the same language, without the quantum effect, and with inductive types and recursion. This is done hoping that it is possible to generalise to the language incorporating the quantum effect.

However, we show in Chapter 5 that this generalisation is not that simple. On a more

positive note, we outline a potential categorical model for quantum control on infinite data types and recursion, with the help of *guarded recursion*. This model strongly echoes with the category $\mathbf{LSI}_{\leq}$ introduce by Pablo Andrés-Martínez in his thesis [AM22]. In the conclusion of his thesis, he wrote:

> "Quantum computer scientists tend to dismiss unbounded iteration in quantum algorithms as an uninteresting field of work: in the case of classical control flow due to the assumption that testing a termination condition on every iteration would destroy any achievable quantum speed-up and, in the case of quantum control flow, due to the technical obstacles that interference and the possibility of infinitely many execution paths would entail [...]"

Even if these concerns were verified, these comments apply to classical computing also, since the memory of our computer is finite. However, it did not prevent computer scientists from studying infinite data types, infinite loops and their semantics. Modern programming languages contain types such as natural numbers and lists, which are by essence infinite, even if their representation in the architecture is necessarily finite.

Pablo Andrés-Martínez finishes his conclusion with the following sentence.

> "It is my hope that further study on this field will yield quantum programming languages supporting quantum control flow and new algorithms that make use of unbounded iteration."

I share his hope.

Bibliography

[AB23] Steve Awodey and Andrej Bauer. Introduction to categorical logic, 2023.

[ABI03] Thomas N. E. Greville Adi Ben-Israel. *Generalized Inverses: Theory and Applications*. Springer New York, NY, 2003.

[Abr05] Samson Abramsky. A structural approach to reversible computation. *Theoretical Computer Science*, 347(3):441–464, 2005.

[Abr20] Samson Abramsky. Whither semantics? *Theoretical Computer Science*, 807:3–14, 2020. In memory of Maurice Nivat, a founding father of Theoretical Computer Science - Part II.

[ACG+20] Bogdan Aman, Gabriel Ciobanu, Robert Glück, Robin Kaarsgaard, Jarkko Kari, Martin Kutrib, Ivan Lanese, Claudio Antares Mezzina, Lukasz Mikulski, Rajagopal Nagarajan, Iain C. C. Phillips, G. Michele Pinna, Luca Prigioniero, Irek Ulidowski, and Germán Vidal. Foundations of reversible computation. In Irek Ulidowski, Ivan Lanese, Ulrik Pagh Schultz, and Carla Ferreira, editors, *Reversible Computation: Extending Horizons of Computing - Selected Results of the COST Action IC1405*, volume 12070 of *Lecture Notes in Computer Science*, pages 1–40. Springer, 2020.

[AD17] Pablo Arrighi and Gilles Dowek. Lineal: A linear-algebraic Lambda-calculus. *Logical Methods in Computer Science*, Volume 13, Issue 1, March 2017.

[ADCV17] Pablo Arrighi, Alejandro Díaz-Caro, and Benoît Valiron. The vectorial λ-calculus. *Information and Computation*, 254:105–139, 2017.

[AG05] T. Altenkirch and J. Grattage. A functional quantum programming language. In *20th Annual IEEE Symposium on Logic in Computer Science (LICS' 05)*. IEEE, 2005.

[AG11] Holger Bock Axelsen and Robert Glück. A simple and efficient universal reversible turing machine. In Adrian-Horia Dediu, Shunsuke Inenaga, and Carlos Martín-Vide, editors, *Language and Automata Theory and Applications*, pages 117–128, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[AJ95] Samson Abramsky and Achim Jung. *Domain Theory*, page 1–168. Oxford University Press, Inc., USA, 1995.

[AK16] Holger Bock Axelsen and Robin Kaarsgaard. Join inverse categories as models of reversible recursion. In Bart Jacobs and Christof Löding, editors, *Proceedings of*

the 19th International Conference on Foundations of Software Science and Computation Structures (FOSSACS'16), volume 9634 of *Lecture Notes in Computer Science*, pages 73–90, Eindhoven, The Netherlands, 2016. Springer.

[AM22] Pablo Andrés-Martínez. *Unbounded loops in quantum programs: categories and weak-while loops*. PhD thesis, Laboratory for Foundations of Computer Science, School of Informatics, University of Edinburgh, 2022.

[AMHK23] Pablo Andrés-Martínez, Chris Heunen, and Robin Kaarsgaard. Universal properties of partial quantum maps. *Electronic Proceedings in Theoretical Computer Science*, 394:192–207, November 2023.

[AMM18] Jiří Adámek, Stefan Milius, and Lawrence S. Moss. Fixed points of functors. *Journal of Logical and Algebraic Methods in Programming*, 95:41–81, 2018.

[Asa10] Kazuyuki Asada. Arrows are strong monads. In *Proceedings of the Third ACM SIGPLAN Workshop on Mathematically Structured Functional Programming*, MSFP '10, page 33–42, New York, NY, USA, 2010. Association for Computing Machinery.

[ASvW+05] Artem Alimarine, Sjaak Smetsers, Arjen van Weelden, Marko van Eekelen, and Rinus Plasmeijer. There and back again: arrows for invertible programming. In *Proceedings of the 2005 ACM SIGPLAN Workshop on Haskell*, Haskell '05, page 86–97, New York, NY, USA, 2005. Association for Computing Machinery.

[BAP+12] Antoine Bérut, Artak Arakelyan, Artyom Petrosyan, Sergio Ciliberto, Raoul Dillenschneider, and Eric Lutz. Experimental verification of landauer's principle linking information and thermodynamics. *Nature*, 483(7388):187–189, 2012.

[Bar84] Henk Barendregt. 'the lambda calculus: its syntax and semantics'. *Studies in logic and the foundations of Mathematics*, 1984.

[Bar92] Michael Barr. Algebraically compact functors. *Journal of Pure and Applied Algebra*, 82(3):211–231, 1992.

[Bar14] Miklós Bartha. Quantum turing automata. *Electronic Proceedings in Theoretical Computer Science*, 143:17–31, mar 2014.

[BD86] Leo Bachmair and Nachum Dershowitz. Commutation, transformation, and termination. In Jörg H. Siekmann, editor, *8th International Conference on Automated Deduction*, pages 5–20, Berlin, Heidelberg, 1986. Springer Berlin Heidelberg.

[Ben73] Charles H Bennett. Logical reversibility of computation. *IBM Journal of Research and Development*, 17(6):525–532, 1973.

[Ben82] Charles H. Bennett. The thermodynamics of computation–a review. *International Journal of Theoretical Physics*, 21(12):905–940, 1982.

[Ben00] Charles H. Bennett. Notes on the history of reversible computation. *IBM Journal of Research and Development*, 44(1):270–278, 2000.

[Ben15] Nick Benton. Categorical monads and computer programming. *LMS Impact150 Stories*, 1:9–13, 2015.

[BMSS12] L. Birkedal, R. Møgelberg, J. Schwinghammer, and K. Støvring. First steps in synthetic guarded domain theory: step-indexing in the topos of trees. *Logical Methods in Computer Science*, 8(4), October 2012.

[BR23] Henning Basold and Tanjona Ralaivaosaona. Composition and Recursion for Causal Structures. In Paolo Baldan and Valeria de Paiva, editors, *10th Conference on Algebra and Coalgebra in Computer Science (CALCO 2023)*, volume 270 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 18:1–18:17, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[BSS10] Lars Birkedal, Jan Swinghammer, and Kristian Støvring. A metric model of lambda calculus with guarded recursion. In Luigi Santocanale, editor, *Fixed Points in Computer Science 2010*, 2010. FICS 2010, the 7th Workshop on Fixed Points in Computer Science, was held in Brno, Czech Republic, on August 21-22 2010, as a satellite workshop to the conferences Mathematical Foundations of Computer Science and Computer Science Logic, 2010.

[Car12] Michael Kirkedal Carøe. *Design of Reversible Computing Systems*. PhD thesis, University of Copenhagen, Denmark, 2012.

[CBGB16] R. Clouston, A. Bizjak, H. Bugge Grathwohl, and L. Birkedal. The guarded lambda calculus: Programming and reasoning with guarded recursion for coinductive types. *Logical Methods in Computer Science*, 2016. Accepted for publication (journal version of FOSSACS 2015 paper).

[CDPV13] Giulio Chiribella, Giacomo Mauro D'Ariano, Paolo Perinotti, and Benoit Valiron. Quantum computations without definite causal structure. *Physical Review A*, 88(2):022318, 2013.

[CdVVV22] Kostia Chardonnet, Marc de Visme, Benoît Valiron, and Renaud Vilmart. The many-worlds calculus, 2022.

[Cha23] Kostia Chardonnet. *Towards a Curry-Howard Correspondence for Quantum Computation*. Theses, Université Paris-Saclay, January 2023.

[CHKS23] Jacques Carette, Chris Heunen, Robin Kaarsgaard, and Amr Sabry. The quantum effect: A recipe for quantumpi, 2023.

[Chu32] Alonzo Church. A set of postulates for the foundation of logic. *Annals of Mathematics*, 33(2):346–366, 1932.

[CL02] J. Robin B. Cockett and Stephen Lack. Restriction categories I: Categories of partial maps. *Theoretical Computer Science*, 270(1):223–259, 2002.

[CL03] J. Robin B. Cockett and Stephen Lack. Restriction categories ii: partial map classification. *Theoretical Computer Science*, 294(1):61–102, 2003.

[CL07] Robin Cockett and Stephen Lack. Restriction categories III: Colimits, partial limits and extensivity. *Mathematical Structures in Computer Science*, 17(4):775–817, 2007.

[CLV21] Kostia Chardonnet, Louis Lemonnier, and Benoît Valiron. Categorical semantics of reversible pattern-matching. *Electronic Proceedings in Theoretical Computer Science*, 351:18–33, Dec 2021.

[CLV23] Kostia Chardonnet, Louis Lemonnier, and Benoît Valiron. Semantics for a turing-complete reversible programming language with inductive types, 2023.

[CLZ23] Titouan Carette, Louis Lemonnier, and Vladimir Zamdzhiev. Central submonads and notions of computation: Soundness, completeness and internal languages. In *2023 38th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–13, 2023.

[Cro94] Roy L. Crole. *Categories for Types*. Cambridge University Press, 1994.

[Cro12] O.A. Cross. *Topics in Quantum Computing*. CreateSpace Independent Publishing Platform, 2012.

[CS21] Chao-Hong Chen and Amr Sabry. A computational interpretation of compact closed categories: reversible programming with negative and fractional types. *Proc. ACM Program. Lang.*, 5(POPL), jan 2021.

[CSV23] Kostia Chardonnet, Alexis Saurin, and Benoît Valiron. A Curry-Howard Correspondence for Linear, Reversible Computation. In Bartek Klin and Elaine Pimentel, editors, *31st EACSL Annual Conference on Computer Science Logic (CSL 2023)*, volume 252 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 13:1–13:18, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[Cur34] H. B. Curry. Functionality in combinatory logic. *Proceedings of the National Academy of Sciences*, 20(11):584–590, 1934.

[CW16] Kenta Cho and Abraham Westerbaan. Von neumann algebras form a model for the quantum lambda calculus, 2016.

[Day70] Brian John Day. *Construction of biclosed categories*. PhD thesis, University of New South Wales PhD thesis, 1970.

[DC22] Alejandro Díaz-Caro. A quick overview on the quantum control approach to the lambda calculus. *arXiv preprint arXiv:2204.03885*, 2022.

[DCGMV19] Alejandro Diaz-Caro, Mauricio Guillermo, Alexandre Miquel, and Benoit Valiron. Realizability in the unitary sphere. In *2019 34th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. IEEE, jun 2019.

[DCHPS23] Alejandro Díaz-Caro, Emmanuel Hainry, Romain Péchoux, and Mário Silva. A feasible and unitary programming language with quantum control, 2023.

[DCM22a] Alejandro Dí az-Caro and Octavio Malherbe. Quantum control in the unitary sphere: Lambda-s1 and its categorical model. *Logical Methods in Computer Science*, Volume 18, Issue 3, sep 2022.

[DCM22b] Alejandro Díaz-Caro and Octavio Malherbe. Semimodules and the (syntactically-)linear lambda calculus, 2022.

[DGM03] Pietro Di Gianantonio and Marino Miculan. A unifying approach to recursive and co-recursive definitions. In Herman Geuvers and Freek Wiedijk, editors, *Types for Proofs and Programs*, pages 148–161, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[DGM04] Pietro Di Gianantonio and Marino Miculan. Unifying recursive and co-recursive definitions in sheaf categories. In Igor Walukiewicz, editor, *Foundations of Software Science and Computation Structures*, pages 136–150, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[Fio96] Marcelo P. Fiore. *Axiomatic Domain Theory in Categories of Partial Maps*. Distinguished Dissertations in Computer Science. Cambridge University Press, 1996.

[Fio04] M.P. Fiore. *Axiomatic Domain Theory in Categories of Partial Maps*. Distinguished Dissertations in Computer Science. Cambridge University Press, 2004.

[Flo67] Robert W. Floyd. Assigning meanings to programs. In *J.T. Schwartz: Mathematical Aspects of Computer Science. Proceedings of Symposium on Applied Mathematics*, volume 19, pages 91–32. American Mathematical Society, 1967.

[Fü99] Carsten Führmann. Direct models of the computational lambda-calculus. *Electronic Notes in Theoretical Computer Science*, 20:245–292, 1999. MFPS XV, Mathematical Foundations of Progamming Semantics, Fifteenth Conference.

[Gar14] Richard Garner. Lawvere theories, finitary monads and cauchy-completion. *Journal of Pure and Applied Algebra*, 218(11):1973–1988, 2014.

[GF16] Richard Garner and Ignacio López Franco. Commutativity. *Journal of Pure and Applied Algebra*, 220(5):1707–1751, may 2016.

[GHK+12] Gerhard Gierz, Karl Heinrich Hofmann, Klaus Keimel, Jimmie D Lawson, Michael Mislove, and Dana S Scott. *A compendium of continuous lattices*. Springer Science & Business Media, 2012.

[Gil14] Brett Gordon Giles. *An Investigation of Some Theoretical Aspects of Reversible Computing*. PhD thesis, University of Calgary, 2014.

[Gir82] Michèle Giry. A categorical approach to probability theory. In B. Banaschewski, editor, *Categorical Aspects of Topology and Analysis*, pages 68–85, Berlin, Heidelberg, 1982. Springer Berlin Heidelberg.

[GJT21] Jean Goubault-Larrecq, Xiaodong Jia, and Clément Théron. A domain-theoretic approach to statistical programming languages. *CoRR*, abs/2106.16190, 2021.

[GK18] Robert Glück and Robin Kaarsgaard. A categorical foundation for structured reversible flowchart languages: Soundness and adequacy. *Log. Methods Comput. Sci.*, 14(3), 2018.

[GKY19] Robert Glück, Robin Kaarsgaard, and Tetsuo Yokoyama. Reversible programs have reversible semantics. In Emil Sekerinski, Nelma Moreira, José N. Oliveira, Daniel Ratiu, Riccardo Guidotti, Marie Farrell, Matt Luckcuck, Diego Marmsoler, José Campos, Troy Astarte, Laure Gonnord, Antonio Cerone, Luis Couto, Brijesh Dongol, Martin Kutrib, Pedro Monteiro, and David Delmas, editors, *Formal Methods. FM 2019 International Workshops - Porto, Portugal, October 7-11, 2019, Revised Selected Papers, Part II*, volume 12233 of *Lecture Notes in Computer Science*, pages 413–427. Springer, 2019.

[GP18] Richard Garner and John Power. An enriched view on the extended finitary monad–Lawvere theory correspondence. *Logical Methods in Computer Science*, Volume 14, Issue 1, February 2018.

[GTL89] Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and types*. Cambridge University Press, USA, 1989.

[Gun92] C.A. Gunter. *Semantics of Programming Languages: Structures and Techniques*. The MIT Press, 1992.

[Guo12] Xiuzhan Guo. *Products, Joins, Meets, and Ranges in Restriction Categories*. PhD thesis, University of Calgary, 2012.

[Heu13] Chris Heunen. *On the Functor ℓ^2*, pages 107–121. Springer Berlin Heidelberg, 2013.

[HK15] Chris Heunen and Martti Karvonen. Reversible monadic computing. In Dan Ghica, editor, *Proceedings of the 31st Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXI)*, volume 319 of *Electronic Notes in Theoretical Computer Science*, pages 217–237, Nijmegen, The Netherlands, 2015.

[HK22a] Chris Heunen and Robin Kaarsgaard. Quantum information effects. *Proc. ACM Program. Lang.*, 6(POPL), jan 2022.

[HK22b] Chris Heunen and Andre Kornell. Axioms for the category of hilbert spaces. *Proceedings of the National Academy of Sciences*, 119(9), 2022.

[HKK18a] Chris Heunen, Robin Kaarsgaard, and Martti Karvonen. Reversible effects as inverse arrows. *Electronic Notes in Theoretical Computer Science*, 341:179–199, 2018. Proceedings of the Thirty-Fourth Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXIV).

[HKK18b] Chris Heunen, Robin Kaarsgaard, and Martti Karvonen. Reversible effects as inverse arrows. In Sam Staton, editor, *Proceedings of the 34th Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXIV)*, volume 341 of *Electronic Notes in Theoretical Computer Science*, pages 179–199, Dalhousie University, Halifax, Canada, 2018. Elsevier.

[HKvdS22] Chris Heunen, Andre Kornell, and Nesta van der Schaff. Axioms for the category of hilbert spaces and linear contractions, 2022.

[HLMS23] Mathieu Huot, Alexander K. Lew, Vikash K. Mansinghka, and Sam Staton. ωpap spaces: Reasoning denotationally about higher-order, recursive probabilistic and differentiable programs. In *2023 38th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–14, 2023.

[How80] William Alvin Howard. The formulae-as-types notion of construction. In Haskell Curry, Hindley B., Seldin J. Roger, and P. Jonathan, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*. Academic Press, 1980.

[HP07] Martin Hyland and John Power. The category theoretic understanding of universal algebra: Lawvere theories and monads. *Electr. Notes Theor. Comput. Sci.*, 172:437–458, 04 2007.

[Hug00] John Hughes. Generalising monads to arrows. *Sci. Comput. Program.*, 37(1–3):67–111, may 2000.

[HV19] Chris Heunen and Jamie Vicary. *Categories for Quantum Theory: An Introduction*. Oxford University Press, 11 2019.

[Jac16] Bart Jacobs. *Introduction to Coalgebra: Towards Mathematics of States and Observation*, volume 59 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2016.

[JHH09] Bart Jacobs, Chris Heunen, and Ichiro Hasuo. Categorical semantics for arrows. *Journal of Functional Programming*, 19(3–4):403–438, 2009.

[JKL+22] Xiaodong Jia, Andre Kornell, Bert Lindenhovius, Michael Mislove, and Vladimir Zamdzhiev. Semantics for variational quantum programming. *Proc. ACM Program. Lang.*, 6(POPL), jan 2022.

[JKT18] Petur Andrias Højgaard Jacobsen, Robin Kaarsgaard, and Michael Kirkedal Thomsen. CoreFun : A typed functional reversible core language. In Jarkko Kari and Irek Ulidowski, editors, *Reversible Computation - 10th International Conference, RC 2018, Leicester, UK, September 12-14, 2018, Proceedings*, volume 11106 of *Lecture Notes in Computer Science*, pages 304–321. Springer, 2018.

[JLMZ21a] Xiaodong Jia, Bert Lindenhovius, Michael Mislove, and Vladimir Zamdzhiev. Commutative monads for probabilistic programming languages. In *2021 36th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. IEEE, jun 2021.

[JLMZ21b] Xiaodong Jia, Bert Lindenhovius, Michael W. Mislove, and Vladimir Zamdzhiev. Commutative monads for probabilistic programming languages. In *36th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2021, Rome, Italy, June 29 - July 2, 2021*, pages 1–14. IEEE, 2021.

[JMS22] Tomáš Jakl, Dan Marsden, and Nihil Shah. Generalizations of bilinear maps – technical report, 2022.

[JMZ21] Xiaodong Jia, Michael W. Mislove, and Vladimir Zamdzhiev. The central valuations monad (early ideas). In Fabio Gadducci and Alexandra Silva, editors, *9th Conference on Algebra and Coalgebra in Computer Science, CALCO 2021, August 31 to September 3, 2021, Salzburg, Austria*, volume 211 of *LIPIcs*, pages 18:1–18:5. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

[Jon90] Claire Jones. *Probabilistic Non-determinism*. PhD thesis, University of Edinburgh, UK, 1990.

[JP89] C. Jones and Gordon D. Plotkin. A probabilistic powerdomain of evaluations. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science (LICS '89), Pacific Grove, California, USA, June 5-8, 1989*, pages 186–195. IEEE Computer Society, 1989.

[JS12] Roshan P. James and Amr Sabry. Information effects. In John Field and Michael Hicks, editors, *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'12)*, pages 73–84, Philadelphia, Pennsylvania, USA, 2012. ACM.

[JS14] Rosham P. James and Amr Sabry. Theseus: A high-level language for reversible computing. Draft, available on Citeseerx, 2014.

[Kaa17] Robin Kaarsgaard. *The Logic of Reversible Computing*. PhD thesis, University of Copenhagen, 2017.

[Kaa19a] Robin Kaarsgaard. Condition/decision duality and the internal logic of extensive restriction categories. In Barbara König, editor, *Proceedings of the 35th Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXV)*, volume 347 of *Electronic Notes in Theoretical Computer Science*, pages 179–202, London, UK, 2019.

[Kaa19b] Robin Kaarsgaard. Inversion, iteration, and the art of dual wielding. In Michael Kirkedal Thomsen and Mathias Soeken, editors, *Proceedings of the 11th International Conference on Reversible Computation (RC 2019)*, volume 11497 of *Lecture Notes in Computer Science*, pages 34–50, Lausanne, Switzerland, 2019. Springer.

[KAG17] Robin Kaarsgaard, Holger Bock Axelsen, and Robert Glück. Join inverse categories and reversible recursion. *J. Log. Algebraic Methods Program.*, 87:33–50, 2017.

[Kar19] Martti Karvonen. *The Way of the Dagger*. PhD thesis, 2019.

[Kas79] J. Kastl. Inverse categories. In *Algebraische Modelle, Kategorien und Gruppoide*, Studien zur Algebra und ihre Anwendungen, Band 7, pages 51–60. Berlin, Akademie-Verlag, 1979.

[Kei78] William F Keigher. Symmetric monoidal closed categories generated by commutative adjoint monads. *Cahiers de topologie et géométrie différentielle catégoriques*, 19(3):269–293, 1978.

[Kel65] G.M Kelly. Tensor products in categories. *Journal of Algebra*, 2(1):15–37, 1965.

[Kel82] Max Kelly. *Basic concepts of enriched category theory*, volume 64. CUP Archive, 1982.

[Kel86] G. M. Kelly. A survey of totality for enriched and ordinary categories. *Cahiers de Topologie et Géométrie Différentielle Catégoriques*, 27:109–132, 1986.

[KR21] Robin Kaarsgaard and Mathys Rennela. Join inverse rig categories for reversible functional programming, and beyond. In Ana Sokolova, editor, Proceedings 37th Conference on *Mathematical Foundations of Programming Semantics,* Hybrid: Salzburg, Austria and Online, 30th August - 2nd September, 2021, volume 351 of *Electronic Proceedings in Theoretical Computer Science*, pages 152–167. Open Publishing Association, 2021.

[KV19] Robin Kaarsgaard and Niccolò Veltri. En garde! unguarded iteration for reversible computation in the delay monad. In Graham Hutton, editor, *Proceedings of the 13th International Conference on Mathematics of Program Construction (MPC 2019)*, volume 11825 of *Lecture Notes in Computer Science*, pages 366–384, Porto, Portugal, October 2019. Springer Verlag.

[Lan61] Rolf Landauer. Irreversibility and heat generation in the computing process. *IBM Journal of Research and Development.*, 5(3):183–191, 1961.

[Lei16] Tom Leinster. Basic category theory, 2016.

[LMZ18] Bert Lindenhovius, Michael Mislove, and Vladimir Zamdzhiev. Enriching a linear/non-linear lambda calculus: A programming language for string diagrams. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '18, page 659–668, New York, NY, USA, 2018. Association for Computing Machinery.

[LMZ21] Bert Lindenhovius, Michael Mislove, and Vladimir Zamdzhiev. LNL-FPC: The Linear/Non-linear Fixpoint Calculus. *Logical Methods in Computer Science*, Volume 17, Issue 2, April 2021.

[Luc18] Rory B. B. Lucyshyn-Wright. Commutants for enriched algebraic theories and monads. *Appl. Categorical Struct.*, 26(3):559–596, 2018.

[Lut86] Christopher Lutz. Janus: a time-reversible language. Letter to Rolf Landauer, posted online by Tetsuo Yokoyama on http://www.tetsuo.jp/ref/janus.html, 1986.

[LW18] Rory Lucyshyn-Wright. Convex spaces, affine spaces, and commutants for algebraic theories. *Applied Categorical Structures*, 26, 04 2018.

[Mac95] Ian Mackie. The geometry of interaction machine. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'95*, pages 198–208. ACM Press, 1995.

[Mal13] Octavio Malherbe. *Categorical models of computation: partially traced categories and presheaf models of quantum computation*. PhD thesis, 2013.

[Mar65] J.-M. Maranda. Formal categories. *Canadian Journal of Mathematics*, 17:758–801, 1965.

[MH24] Matthew Di Meglio and Chris Heunen. Dagger categories and the complex numbers: Axioms for the category of finite-dimensional hilbert spaces and linear contractions, 2024.

[ML98] Saunders Mac Lane. *Categories for the Working Mathematician (2nd ed.)*. Springer, 1998.

[MM12] S. MacLane and I. Moerdijk. *Sheaves in Geometry and Logic: A First Introduction to Topos Theory*. Universitext. Springer New York, 2012.

[MM22] Dylan McDermott and Alan Mycroft. Galois connecting call-by-value and call-by-name, 2022.

[MMDPR05] Maria Maietti, Paola Maneggia, Valeria De Paiva, and Eike Ritter. Relating categorical semantics for intuitionistic linear logic. *Applied Categorical Structures*, 13:1–36, 01 2005.

[MMV20] Bassel Mannaa, Rasmus Ejlers Møgelberg, and Niccolò Veltri. Ticking clocks as dependent right adjoints: Denotational semantics for clocked type theory. *Logical Methods in Computer Science*, Volume 16, Issue 4, December 2020.

[Mog89] Eugenio Moggi. Computational lambda-calculus and monads. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science (LICS '89), Pacific Grove, California, USA, June 5-8, 1989*, pages 14–23. IEEE Computer Society, 1989.

[Mog91] Eugenio Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1):55–92, 1991.

[Nak00] H. Nakano. A modality for recursion. In *Proceedings Fifteenth Annual IEEE Symposium on Logic in Computer Science (Cat. No.99CB36332)*, pages 255–266, 2000.

[NC02] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information*. Cambridge University Press, 2002.

[NC10] M. A. Nielsen and I. L. Chuang. *Quantum Computation and Quantum Information: 10th Anniversary Edition*. Cambridge University Press, 2010.

[Plo77] Gordon D. Plotkin. Lcf considered as a programming language. *Theoretical Computer Science*, pages 223–255, 1977.

[Plo85] Gordon D. Plotkin. *Lectures on predomains and partial functions*. Stanford University, 1985.

[PMA+14] Lorenzo Procopio, Amir Moqanaki, Mateus Araújo, Fabio Costa, Irati Calafell, Emma Dowd, Deny Hamel, Lee Rozema, Časlav Brukner, and Philip Walther. Experimental superposition of orders of quantum gates. *Nature Communications*, 6, 12 2014.

[PPRZ20] Romain Péchoux, Simon Perdrix, Mathys Rennela, and Vladimir Zamdzhiev. Quantum programming with inductive datatypes: Causality and affine type theory. In Jean Goubault-Larrecq and Barbara König, editors, *Foundations of Software Science and Computation Structures*, pages 562–581, Cham, 2020. Springer International Publishing.

[PR97] John Power and Edmund P. Robinson. Premonoidal categories and notions of computation. *Math. Struct. Comput. Sci.*, 7:453–468, 1997.

[PSV14] Michele Pagani, Peter Selinger, and Benoît Valiron. Applying quantitative semantics to higher-order quantum computing. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, page 647–658, New York, NY, USA, 2014. Association for Computing Machinery.

[Ric54] H. G. Rice. Recursive real numbers. *Proceedings of the American Mathematical Society*, (5):784–791, 1954.

[RRF+17] Giulia Rubino, Lee A. Rozema, Adrien Feix, Mateus Araújo, Jonas M. Zeuner, Lorenzo M. Procopio, Časlav Brukner, and Philip Walther. Experimental verification of an indefinite causal order. *Science Advances*, 3(3):e1602589, 2017.

[RS18] Mathys Rennela and Sam Staton. Classical control and quantum circuits in enriched category theory. *Electronic Notes in Theoretical Computer Science*, 336:257–279, 2018. The Thirty-third Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXIII).

[RSN55] Frigyes Riesz and Béla Sőkefalvi-Nagy. *Functional Analysis*. Frederick Ungar Publishing Company, 1955.

[Sco55] Dana Scott. Definitions by abstraction in axiomatic set theory. *Bull. Amer. Math. Soc*, 61(442):I955, 1955.

[SHLG94] V. Stoltenberg-Hansen, I. Lindström, and E. R. Griffor. *Mathematical Theory of Domains*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1994.

[SL13] Sam Staton and Paul Blain Levy. Universal properties of impure programming languages. In Roberto Giacobazzi and Radhia Cousot, editors, *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, pages 179–192. ACM, 2013.

[SM13] Mehdi Saeedi and Igor L. Markov. Synthesis and optimization of reversible circuits - a survey. *ACM Computing Surveys*, 45(2):21:1–21:34, 2013.

[SN54] Béla Szőkefalvi-Nagy. Sur les contractions de l'espace de hilbert. *Acta scientiarum mathematicarum*, 15:pp. 87–92, 1954.

[SR20] Víctor Cabezas Sanchez and Fernando Pablo Romo. Moore-penrose inverse of some linear maps on infinite-dimensional vector spaces. *Electronic Journal of Linear Algebra*, 2020.

[Str72] Ross Street. The formal theory of monads. *Journal of Pure and Applied Algebra*, 2(2):149–168, 1972.

[SV09] Peter Selinger and Benoıt Valiron. Quantum lambda calculus. *Semantic techniques in quantum computation*, pages 135–172, 2009.

[SVV18] Amr Sabry, Benoît Valiron, and Juliana Kaizer Vizzotto. From symmetric pattern-matching to quantum control. In Christel Baier and Ugo Dal Lago, editors, *Proceedings of the 21st International Conference on Foundations of Software Science and Computation Structures (FOSSACS'18)*, volume 10803 of *Lecture Notes in Computer Science*, pages 348–364, Thessaloniki, Greece, 2018. Springer.

[TA15] Michael Kirkedal Thomsen and Holger Bock Axelsen. Interpretation and programming of the reversible functional language RFUN. In Ralf Lämmel, editor, *Proceedings of the 27th Symposium on the Implementation and Application of Functional Programming Languages, IFL 2015, Koblenz, Germany, September 14-16, 2015*, pages 8:1–8:13. ACM, 2015.

[TA24] Takeshi Tsukada and Kazuyuki Asada. Enriched presheaf model of quantum fpc. *Proc. ACM Program. Lang.*, 8(POPL), jan 2024.

[Tur37] A. M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2-42(1):230–265, 1937.

[Val08] Benoit Valiron. *Semantics for a Higher-Order Functional Programming Language for Quantum Computation*. PhD thesis, CAN, 2008. AAINR46525.

[Vau09] Lionel Vaux. The algebraic lambda calculus. *Mathematical Structures in Computer Science*, 19(5):1029–1059, 2009.

[VLRH23] Finn Voichick, Liyi Li, Robert Rand, and Michael Hicks. Qunity: A unified language for quantum and classical computing. *Proc. ACM Program. Lang.*, 7(POPL), jan 2023.

[Wra70] Gavin C. Wraith. *Algebraic Theories*. Aarhus universitet, Matematisk institut, 1970.

[WSSD16] Robert Wille, Eleonora Schönborn, Mathias Soeken, and Rolf Drechsler. SyReC: A hardware description language for the specification and synthesis of reversible circuits. *Integration, the VLSI Journal*, 53:39–53, 2016.

[YAG12] Tetsuo Yokoyama, Holger Bock Axelsen, and Robert Glück. Towards a reversible functional language. In Alexis De Vos and Robert Wille, editors, *Revised Papers of the Third International Workshop on Reversible Computation (RC'11)*, volume 7165 of *Lecture Notes in Computer Science*, pages 14–29, Gent, Belgium, 2012. Springer.

[YAG16] Tetsuo Yokoyama, Holger Bock Axelsen, and Robert Glück. Fundamentals of reversible flowchart languages. *Theoretical Computer Science*, 611:87–115, 2016.

[YG07] Tetsuo Yokoyama and Robert Glück. A reversible programming language and its invertible self-interpreter. In G. Ramalingam and Eelco Visser, editors, *Proceedings of the 2007 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation, PEPM 2007, Nice, France, January 15-16, 2007*, pages 144–153, 2007.

[Yok10] Tetsuo Yokoyama. Reversible computation and reversible programming languages. In Irek Ulidowski, editor, *Proceedings of the Workshop on Reversible Computation (RC'09)*, volume 253(6) of *Electronic Notes in Theoretical Computer Science*, pages 71–81, York, UK, 2010. Elsevier.