

Thèse de doctorat en informatique

Towards a Curry-Howard Correspondence for Quantum Computation

Par **Kostia Chardonnet**

Dirigée par

Pablo Arrighi

Professeur à l'Université Paris Saclay

Directeur de thèse

Benoît Valiron

Maître de Conférence à l'École CentraleSupélec

Co-Encadrant de thèse

Alexis Saurin

Chargé de Recherche CNRS à l'Université Paris Cité

Co-Encadrant de thèse

Présentée et soutenue publiquement le 9 Janvier 2023,
devant un jury composé de :

Claudia Faggian

Chargée de recherche CNRS à l'Université Paris Cité

Jury

Delia Kesner

Professeure à l'Université Paris Cité

Jury

Pierre Clairambault

Chargé de recherche CNRS à l'Université d'Aix-Marseille

Jury

Miriam Backens

Lecturer à l'Université de Birmingham

Jury

Laurent Régnier

Professeur à l'Université d'Aix-Marseille

Rapporteur

Emmanuel Jeandel

Professeur à l'Université de Lorraine

Rapporteur

Contents

Introduction	8
Introduction (fr)	13
I. Background	18
1. Reversible & Quantum Computation	19
1.1. Reversible Computation	19
1.1.1. Reversible Primitive Permutations	20
1.1.2. Reversible Turing Machines	22
1.2. Quantum Computation	24
2. Graphical Language for Quantum Computation	27
2.1. The ZX-Calculus	28
2.1.1. Pure Operators	29
2.1.2. Standard Interpretation	30
2.1.3. Properties and structure	31
2.1.4. ZX-diagrams for Mixed Processes	33
3. Proof Theory & The Curry Howard Isomorphism	35
3.1. Logic & Computation	35
3.1.1. The Simply-Typed λ -Calculus	35
3.1.2. Logic	37
3.1.3. Curry-Howard	39
3.2. Linear Logic	39
3.2.1. Proof Nets	43
3.2.2. Geometry of Interaction	44
3.3. Linear Logic & Quantum Computation	45
3.4. Infinitary Linear Logic : μ MALL	45
3.4.1. Background on μ MALL	46
3.4.2. Bouncing Validity	47
3.4.3. Circular Representation of Derivations	52

II. Contributions	54
4. A Curry-Howard Correspondence for Linear, Reversible Computation	55
4.1. Introduction	55
4.2. First-order Isos	57
4.3. Computational Content	67
4.3.1. From RPP to Isos	67
4.4. Proof Theoretical Content	70
4.4.1. Translating isos into μ MALL	70
4.4.2. Pre-Proof Validity	73
4.4.3. Proof Simulation	82
4.5. Removing Exhaustivity	88
4.5.1. Encoding of Reversible Turing Machines	88
4.6. Conclusion	93
5. Geometry of Interaction for ZX Calculus	95
5.1. Introduction	95
5.2. Notions of Graph Theory in ZX	97
5.3. A Token Machine for ZX-diagrams	98
5.3.1. Diffusion and Collision Rules	99
5.3.2. Strong Normalization and Confluence	101
5.3.3. Semantics and Structure of Normal Forms	107
5.3.4. Discussions	114
5.4. Extension to Mixed Processes	115
5.4.1. Token Machine for Mixed Processes	115
5.5. Variations of the Token Machine	117
5.5.1. Pulse Rewriting	117
5.5.2. Synchronicity	118
5.6. Sum Over Path Semantics	119
5.6.1. SOP Token Machine for Pure Operators	119
5.6.2. SOP Token Machine for Mixed Processes	124
5.7. Conclusion and Future Work	127
6. Many Worlds Calculus	128
6.1. Introduction	128
6.2. The Many-Worlds Calculus	130
6.2.1. A First Graphical Language	131
6.3. The Token Machine	132
6.3.1. Tokens and Token States	133
6.3.2. Token Machine's Pulse Rewrite Strategy	134
6.3.3. Token Machine's Asynchronous Rewriting	140
6.3.4. Discussions	148
6.4. Worlds labelling	149
6.5. The Denotational Semantics	151

Contents

6.6. The Equational Theory	153
6.7. Induced Equations	154
6.8. Comparison with Other Graphical Languages	158
6.8.1. ZX-Calculus	159
6.8.2. Tensor-Sum Logic	159
6.8.3. PBS-Calculus	159
6.8.4. LOv-Calculus	160
6.8.5. Path-Calculus	161
6.8.6. Linear Logic's Proof Nets	161
6.9. Representing Computation	161
6.9.1. Syntax of the Language	161
6.9.2. Encoding into the Many-Worlds	163
6.10. Conclusion	169
Conclusion	169
Bibliography	171

Abstract

In this thesis, we are interested in the development of a Curry-Howard correspondence for quantum computing, allowing to represent quantum types and quantum control-flow. In the standard model of Quantum Computation, a classical computer is linked to a quantum coprocessor. The classical computer can then send instructions to allocate, update, or read quantum registers. The programs executed by the coprocessor are represented by a quantum circuit: a sequence of instructions that applies unitary operations to the input quantum bits. While the model is universal, in the sense that it can represent any unitary operations, it stays limited: it lacks of proper representation of non-causal execution flow. Normally, to represent branching, one can use a type system featuring a coproduct, allowing for the choice between two possible executions, but quantum circuits only feature qubits and tensors thereof. On the other hand, types are strongly related to logic via the Curry-Howard correspondence which states that types of programs correspond to formulas and programs to proofs, while the program evaluation is matched with the proof simplification. While this correspondence has been extended to multiple setting in classical computer, it has yet to emerge in quantum computing.

To address those problems we follow two different approaches: the first one, through the development of a linear and reversible programming language, capturing a subset of quantum computing, along with a Curry-Howard correspondence with the logic μ MALL. The language comes in two versions: one representing exactly complete, reversible functions while the other one can represent partial functions. Both version comes with an expressivity result: in the former, one can capture the whole class of Primitive Recursive Functions, while in the later any Turing Machine we show how to capture any Turing Machine. The second approach follows the development of token-based semantics, inspired by Girard's Geometry of Interaction, for graphical language for quantum computation. In this approach, a token-based semantics was given for the ZX-Calculus: a graphical language for quantum computation capable of representing any linear operators. We show how the token-based semantics matches the denotational one. We extend the ZX-Calculus with a coproduct and an explicit tensor in the development of the Many-Worlds Calculus. This new languages comes with a token-based semantics and an equational theory. We show how quantum control can be represented in this system. Finally, the programming language is modified to be able to realize pure quantum computation and the graphical language is then used as a denotational model for it.

Résumé

Dans cette thèse, nous nous intéressons au développement d'une correspondance de Curry-Howard pour l'informatique quantique, permettant de représenter des types quantiques et le flot de contrôle quantique. Dans le modèle standard de l'informatique quantique, un ordinateur classique est lié à un coprocesseur quantique. L'ordinateur classique peut alors envoyer des instructions pour allouer, mettre à jour ou lire des registres quantiques. Les programmes exécutés par le coprocesseur sont représentés par un circuit quantique : une séquence d'instructions qui applique des opérations unitaires aux registres quantiques. Bien que le modèle soit universel, dans le sens où il peut représenter n'importe quelle opérations unitaire, il reste limité : il lui manque une représentation correcte du flot d'exécution non causal. Normalement, pour représenter le branchement, on peut utiliser un système de type contenant un coproduit, permettant le choix entre deux exécutions possibles, mais les circuits quantiques ne contiennent que des qubits et leurs tenseurs. D'autre part, les types sont fortement liés à la logique à travers la correspondance Curry-Howard qui stipule que les types de programmes correspondent aux formules et les programmes aux preuves, tandis que l'évaluation du programme correspond à la simplification de la preuve correspondante. Bien que cette correspondance ait été étendue à des cas multiples en informatique classique, elle n'a pas encore émergée dans l'informatique quantique.

Pour résoudre ces problèmes, nous suivons deux approches différentes : la première, par le développement d'un langage de programmation linéaire et réversible, capturant un sous-ensemble de l'informatique quantique, ainsi qu'une correspondance Curry-Howard avec la logique μ MALL. Le langage existe en deux versions : l'une représentant des fonctions réversibles et totales, tandis que l'autre peut représenter des fonctions partielles. Les deux versions sont accompagnées d'un résultat d'expressivité : dans la première, nous pouvons capturer l'ensemble des fonctions primitive récursives, tandis que dans la deuxième, nous montrons comment capturer n'importe quelle Machine de Turing. La deuxième approche suit le développement d'une sémantique à base de jetons, inspirée de la géométrie de l'interaction de Girard, pour des langages graphique pour le calcul quantique. Dans cette approche, une sémantique à base de jetons a été donnée pour le ZX-Calcul : un langage graphique pour l'informatique quantique capable de représenter n'importe quel opérateur linéaire. Nous montrons comment cette nouvelle sémantique correspond à la sémantique dénotationnelle standard. Nous étendons ensuite le ZX-Calcul avec un coproduit et un tenseur explicite dans le développement du Many-Worlds Calcul. Ce nouveau langage est accompagné sémantique à base de jetons et une théorie équationnelle. Nous montrons comment le contrôle quantique peut être représenté dans

Contents

ce système. Enfin, le langage de programmation est modifié dans le cas quantique pur et nous utilisons le Many-Worlds Calculus comme un modèle dénotationnel pour ce nouveau langage de programmation.

Introduction

Quantum Computation. In quantum computing, one has access to a new kind of data: *quantum bits* (qubits), which consists in *superposition* of the classical bits 0 and 1. The use of quantum bits has allowed for the development of new algorithms enjoying an exponential speedup compared to their classical counter-part. The most prominent examples being Grover’s algorithm, allowing to search for an element in a list of n elements in $O(\sqrt{N})$ [Gro96] and its direct application in Shor’s algorithm, allowing for decomposing a number into its prime factors exponentially faster than known classical algorithms [Sho99]. These algorithms are written using the Quantum Memory (QRAM) model. In this model, a classical computer is linked to a quantum coprocessor. While the classical computer has access to the full expressiveness of types systems and known programming methods, the quantum coprocessor is only able to execute *quantum circuits*, the quantum counterparts of boolean circuits, on some input qubits. A quantum circuit consists in a sequence of unitary operations (called *gates*) which update the states of the input qubits, flowing from the input wires of the circuits, through the output wires, traversing the gates as they move. Then, the result of the execution of the quantum circuit is sent back to the classical computer after the measurement.

From a semantical perspective, the *state* of a quantum circuit consisting of n quantum bits is a vector in a 2^n -dimensional Hilbert space. A (pure) quantum circuit is a linear, sequential description of elementary operations describing a *linear, unitary map* on the state space.

On a formal aspect, quantum circuits have a very rigid structure, allowing for little abstract reasoning on the execution of the circuit: most gates are matrices and in order to prove any property, one has to realize matrices computation, although some formalisms mitigate this difficulty [Amy18]. Graphical languages for quantum computing has been introduced as a way to solve this problem. Coming all the way from Feynman’s diagrams [FH65], graphical languages are commonly used for representing quantum processes. Whether directly based on quantum circuits [Gre+13; Dal17; PRZ17; Cha+] or stemming from categorical analysis such as the ZX-calculus [CK17; CD11], these formal languages are still tied to the quantum coprocessor model in the sense that the only monoidal structure that can be applied to quantum information is the (multiplicative) Kronecker product. Another approach for abstract reasoning on quantum programs would be the development of *typed* quantum programming languages. In this setting, *types* could help us reason about properties of quantum programs. Types systems have been successfully used in classical computing to reason about programs, in particular

through the Curry-Howard Correspondence [Cur34] which states that types of programs correspond to formulas and programs to proofs, while the program evaluation is matched with the proof simplification. This correspondence has been used to mirror first and second-order logics with dependent-type systems [BC13; Ler09], separation logics with memory-aware type systems [Rey02; Jun+17], resource-sensitive logics with differential privacy [Gab+13], logics with monads with reasoning on side effects [Swa+16; Mai+19], classical logic [Gri89], in the development of rich typed programming languages [Nor07; OSG08]. Such a correspondence has yet to emerge in the quantum setting, even though some progress has been made [DD22; DCM22; SVV18], albeit with limited type structures.

Quantum Control Flow. One peculiar feature of quantum computation is *non-causal execution paths*. Indeed, the Janus-faced quantum computational paradigm features two seemingly distinct notions of control structure. On the one hand, a quantum program follows *classical* control: it is hosted on the conventional computer governing the coprocessor, and can therefore only enjoy loops, tests and other regular causally ordered sequences of operations. On the other hand, the lab bench turns out to be more flexible than the rigid coprocessor model, permitting more elaborate *purely quantum* computational constructs than what quantum circuits or ZX-calculus allow.

The archetypal example of a quantum computational behaviour hardly attainable within quantum circuits or ZX-calculus is the *Quantum Switch* [Chi+13]. Consider two quantum bits x and y and two unitary operations U and V acting on y . The problem consists in generating the operation that performs UV on y if x is in state $|0\rangle$ and VU if it is in state $|1\rangle$.

$$\text{QSwitch}(x, U, V) = \begin{cases} \boxed{U} \boxed{V} & \text{if } x = |0\rangle \\ \boxed{V} \boxed{U} & \text{if } x = |1\rangle \end{cases}$$

But, as x can be in superposition, in general the operation is then:

$$(\alpha |0\rangle + \beta |1\rangle) \otimes |y\rangle \mapsto \alpha |0\rangle \otimes (UV |y\rangle) + \beta |1\rangle \otimes (VU |y\rangle).$$

It is a *purely quantum test*: not only can we have values in superposition (here, x) but also *execution orders*. This is in sharp contrast with what happens within the standard quantum coprocessor model. While this program has been shown to be impossible to implement in a quantum circuit with only one instance of U and V [Chi+13], it is nonetheless physically implementable [Abb+20].

While most quantum programming languages are still linked to the QRAM model and features only classical tests and loops [Sin+22; Gre+13], computational models supporting superposition of execution orders have been studied in the literature, such as proposing a suitable extension of quantum circuits [CDP08; Por+17; VKB21; Wec+21]. These approaches typically aim at discussing the notion of quantum channel from a quantum information theoretical standpoint.

Contents

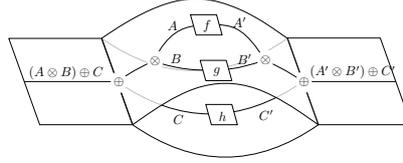


Figure 0.1.: Splits over coproduct and tensor

Quantum Types. Both the QRAM model and the ZX-Calculus only considered tensors of qubits as the types of their input and output data. They do not natively feature coproduct, which could be used for tests, nor richer types structures such as inductive or coinductive types.

The presence of a coproduct could ease the representation of tests, allowing for different execution paths to be considered. This is the approach followed in the PBS-Calculus [CP20], albeit the PBS-Calculus does not feature a tensor: their parallel composition corresponds to the superposition of positions of a single qubit, while in other languages (such as the ZX-Calculus), the parallel composition corresponds to a tensor of qubits.

Typed quantum programming languages featuring both a tensor and a coproduct have been developed [SVV18; VRH22] albeit in these works, no relation with a formal proof system has been established yet. Relating a pure quantum type system with a logical system has been considered in [DD22] in the style of a natural deduction logic through the use of a new connective \odot allowing for the superposition of data, with a studied semantics for a fragment of the language, based on the category of S -semimodule, for S a cancellative commutative semiring [DCM22]. However, their logic is limited to the intuitionistic fragment of linear logic without tensors and the question of quantum control is not discussed.

Graphical languages featuring both a tensor and a coproduct are based on the notion of *tapes*, or *sheets* [Dun09; Mel14]. The coproduct is able to separate a tape in two, making the two part impossible to communicate with one another until they are merged back together, while the tensor is simply the pair of a data, as shown in Figure 0.1. However, these formalisms are not inherently quantum: the splitting and merging of tapes have to be done in a well-bracketed manners making it impossible to represent quantum superposition. From a semantics point of view, pure quantum computation with coproduct and tensors have been studied [DCM22; Cha+22], but it is not clear about inductive types and recursion would fit in those settings and is, so far, the question is left open.

Branching of execution paths can be used to determine whenever a program should terminate or not. The notion of recursion and inductive types in pure quantum computing is yet again lacking: in order to consider recursion, one must make sure that each step of the computation is indeed reversible and that the recursive program is always well-defined, as the whole operation has to be a unitary operation. In [SVV18] the authors

develop a pure quantum programming language with the product and coproduct and the inductive type of lists and with strong constraints ensuring termination and reversibility, which opens the door to further generalization and the development of a proper logic.

Contributions. We try to answer the problem of finding a suitable computation model for pure quantum computation with rich datatype, along with a Curry-Howard correspondence. We offer two approaches:

- The first one, based on a pure quantum programming languages in the style of The-seus [JS12; JS14; SVV18] where we present a linear and reversible programming language with inductive types, together with a Curry-Howard correspondence with the logic μ MALL: linear logic with least and greatest fix points. While [SVV18] extend the language to the quantum case, with the limited type of lists, we stay in the purely classical case. The extension for generalized pure quantum inductive type is left as future work.
- And then using *graphical languages*. First, we define a new semantics for the ZX-Calculus, based on a token-machine inspired from Girard’s Geometry of Interaction, we show how this new semantics capture the usual denotational semantics of the ZX-Calculus. Then, we develop a new graphical language in the style of linear logic proof nets equipped with both a tensor, allowing for handling multiple pieces of information together, and a coproduct, allowing for branching depending on the given input. We develop a token-based semantics on this new language, and then a denotational semantics with an equational theory that is sound and complete.

Plan of the thesis. The thesis is organized as follows: Part I is focused on the mathematical background needed for this thesis. It is split into two parts, Chapter 1 introduce the quantum-theoretical background needed, while Chapter 3 introduce the proof-theoretical formalism of linear logic and μ MALL. Then, Part II focus on my personal contributions. In Chapter 4 we present the linear, reversible language based on pattern-matching. The language features constraints forcing any well-typed function (called iso) to be reversible, and hence an isomorphism. We then show (i) that any primitive recursive function can be encoded as an iso of the language, (ii) how any iso represent a proof-isomorphism in the logic μ MALL, (iii) that by relaxing the constraints on the iso to be able to consider partial maps, the language is Turing Complete. This chapter is a first step towards pure quantum types and quantum control.

Chapter 5 is focused around defining a new kind of semantics for the ZX-Calculus : a graphical language for quantum computation. The semantics is inspired from the token-based geometry of interaction of linear logic, in which tokens move around the graph, capturing the computational content of the graph. We show (i) how the token machine, under some invariants, exactly captures the denotational semantics of the ZX-Diagram, (ii) how it can be adapted to the ZX-Calculus with mixed states, allowing to represent measurement, and (iii) how it can be adapted to another kind of semantics based on

Contents

sum-over-paths. The goal of this chapter is to see how token-based semantics could be adapted to the case of graphical languages. While some previous works have already been done in the quantum case [Dal17], it was still on quantum circuits.

Finally, in Chapter 6 we present a new graphical language called the Many-Worlds Calculus, featuring both products and coproducts. The language allows one to encode all of ZX-Calculus with more primitive constructions and richer types. We give (i) a token-based semantics in the style of Chapter 5, (ii) a denotational semantics deduced from the token machine and (iii) an equational theory that we prove sound and complete with respect to the denotational semantics. Finally, we show how, by adapting a restriction of the language from Chapter 4 to the quantum case, the Many-Worlds Calculus can serve as a model for such a language and as a case study we show how to encode the Quantum Switch into the Many-Worlds with only one occurrence of U and V .

Introduction (fr)

Informatique Quantique. En informatique quantique, nous avons accès à un nouveau type de données : les *bits quantique* (qubits), qui consistent en la *superposition* des bits classiques 0 et 1. L'utilisation de bits quantiques a permis le développement de nouveaux algorithmes bénéficiant d'une vitesse exponentielle par rapport à leurs homologues classiques. Les exemples les plus marquants étant l'algorithme de Grover, permettant de rechercher un élément dans une liste de n éléments en $O(\sqrt{N})$ [Gro96] et son application directe dans l'algorithme de Shor, permettant de décomposer un nombre en ses facteurs premiers exponentiellement plus rapidement que les algorithmes classiques connus [Sho99]. Ces algorithmes sont écrits en utilisant le modèle de mémoire quantique (QRAM). Dans ce modèle, un ordinateur classique est relié à un coprocesseur quantique. Alors que l'ordinateur classique a accès à l'expressivité complète des systèmes de types et aux méthodes de programmation connues, le coprocesseur quantique est seulement capable d'exécuter des *circuits quantiques*, l'homologue quantiques des circuits booléens, sur certains qubits d'entrée. Un circuit quantique consiste en une séquence d'opérations unitaires (appelées *portes*) qui mettent à jour les les états des qubits d'entrée, lorsqu'ils traversent les portes quantiques. Le résultat de l'exécution du circuit quantique est ensuite renvoyé à l'ordinateur classique après avoir effectué une mesure.

D'un point de vue sémantique, les *états* d'une mémoire quantique constituée de n bits quantiques est un vecteur dans un espace de Hilbert de 2^n dimensions. Un circuit quantique est alors une description linéaire et séquentielle d'opérations élémentaires décrivant une *application linéaire, unitaire* sur l'espace d'état.

D'un point de vue formel, les circuits quantiques ont une structure très rigide, ne permettant que peu de raisonnement abstrait sur l'exécution du circuit : la plupart des portes sont des matrices et pour prouver une quelconque propriété il faut réaliser des calculs matriciels, bien que certains formalismes atténuent cette difficulté [Amy18]. Les langages graphiques pour l'informatique quantique ont été présentés comme un moyen de résoudre ce problème. Venant tout droit des diagrammes de Feynman [FH65], les langages graphiques sont couramment utilisés pour représenter les processus quantiques. Qu'ils soient directement basés sur les circuits [Gre+13; Dal17; PRZ17; Cha+] ou issus de l'étude catégorique comme le ZX-calcul [CK17; CD11], néanmoins, ces langages formels sont toujours liés au modèle de coprocesseur quantique dans le sens où la seule structure monoïdale qui peut être appliquée aux qubits est le produit de Kronecker (multiplicatif). Une autre approche pour le raisonnement abstrait sur les programmes quantiques serait le développement de langages de programmation quantiques *typés*.

Dans ce contexte, les *types* pourraient nous aider à raisonner sur les propriétés des programmes quantiques. Les systèmes de types ont été utilisés avec succès en informatique classique pour raisonner sur les programmes, en particulier à travers la correspondance de Curry-Howard [Cur34] qui stipule que les types de programmes correspondent aux formules logiques et les programmes aux preuves, tandis que l'évaluation du programme correspond à la simplification d'une preuve. Cette correspondance a été utilisée pour refléter les logiques du premier et du second ordre avec des types dépendants [BC13; Ler09], des logiques de séparation avec des systèmes de type pouvant gérer la gestion de la mémoire [Rey02; Jun+17], des logiques sensibles aux ressources avec confidentialité différentielle [Gab+13], des logiques avec monades permettant le raisonnement sur les effets de bord [Swa+16; Mai+19], la logique classique [Gri89], dans le développement de langages de programmation avec des systèmes de types riches [Nor07; OSG08]. Néanmoins, une telle correspondance n'a pas encore émergé dans le cadre quantique, même si certains progrès ont été réalisés [DD22; DCM22; SVV18], avec des structures de type limitées.

Quantum Control Flow.

En effet, dans le modèle standard, les opérations sur la mémoire quantique sont séquentielles et non-branchantes. Cela signifie que dans le modèle standard, seul l'ordinateur classique a accès aux boucles, tests, et autres structures de contrôle. Cependant, des expériences montrent que la notion de *contrôle quantique* est réalisable et que le modèle coprocesseur est de ce fait trop rigide et des constructions plus riches peuvent être considérées, ce que les circuits quantiques ou le ZX-Calcul n'autorisent pas.

L'exemple typique d'un comportement de calcul quantique difficilement réalisable dans les circuits quantiques ou le ZX-calcul est le *Quantum Switch* [Chi+13].

Considérons deux bits quantiques x et y et deux opérations unitaires U et V agissant sur y . Le problème consiste à générer l'opération qui exécute UV sur y si x est dans l'état $|0\rangle$ et VU s'il est dans l'état $|1\rangle$.

$$\text{QSwitch}(x, U, V) = \begin{cases} \text{---}U\text{---}V\text{---} & \text{if } x = |0\rangle \\ \text{---}V\text{---}U\text{---} & \text{if } x = |1\rangle \end{cases}$$

Mais, comme x peut être en superposition, l'opération peut donc être décrite par:

$$(\alpha |0\rangle + \beta |1\rangle) \otimes |y\rangle \mapsto \alpha |0\rangle \otimes (UV |y\rangle) + \beta |1\rangle \otimes (VU |y\rangle).$$

Il s'agit d'un test purement quantique : non seulement on peut avoir des valeurs en superposition (ici, x) mais aussi des *ordres d'exécution*. Ceci est en contraste flagrant avec ce qui se passe dans le modèle standard de coprocesseur quantique où la seule source de branchement vient de l'ordinateur classique. Alors que ce programme a

été montré comme étant impossible d’implémenter dans le modèle standard du coprocesseur avec une seule instance de U et de V [Chi+13], il est néanmoins physiquement réalisable [Abb+20].

La plupart des langages de programmation quantique sont encore liés au modèle QRAM et ne comportent que des tests et des boucles classiques [Sin+22; Gre+13], des modèles de calcul supportant la superposition des ordres d’exécution ont été étudiés dans la littérature, tels que ceux proposant une extension des circuits [CDP08; Por+17; VKB21; Wec+21]. Ces approches visent généralement à discuter de la notion de canal quantique d’un point de vue théorique de l’information quantique.

Types Quantique. Le modèle QRAM et le ZX-Calcul ne permettent de représenter que des tenseurs de qubits comme types de données d’entrée et de sortie. Ils ne n’ont pas de coproduit natif, qui pourrait être utilisé pour les tests, ni de structures de types plus riches comme les types inductifs ou coïnductifs.

La présence d’un coproduit pourrait faciliter la représentation des tests, permettant de considérer différents chemins d’exécution. C’est l’approche suivie dans le PBS-Calcul [CP20], bien que le PBS-Calcul ne comporte pas de tenseur : leur composition parallèle correspond à la superposition des positions d’un seul qubit, alors que dans d’autres langages (comme le ZX-Calcul), la composition parallèle correspond à un tenseur de qubits.

Des langages de programmation quantique avec des types, comportant à la fois un tenseur et un coproduit, ont été développés [SVV18; VRH22] bien que dans ces travaux, aucune relation avec un système de preuve formel n’ait encore été établie. La mise en relation d’un système de type quantique pur avec un système logique a été considéré dans [DD22] dans le style d’une logique de déduction naturelle à travers l’utilisation d’un nouveau connecteur \odot permettant la superposition de données, avec une sémantique étudiée pour un fragment de la langage, basée sur la catégorie des S -semimodule, pour S un semiring commutatif annulatif [DCM22]. Cependant, leur logique est limitée au fragment intuitionniste de la logique linéaire sans tenseurs et la question du contrôle quantique n’est pas discutée.

Les langages graphiques comportant à la fois un tenseur et un coproduit sont basés sur la notion de *bandes*, ou de *rubans* [Dum09; Mel14]. Le coproduit est capable de séparer une bande en deux, rendant les deux parties impossible de communiquer l’une avec l’autre jusqu’à ce qu’elles soient fusionnées à nouveau. Tandis que le tenseur est simplement la paire d’une donnée, comme le montre dans Figure 0.2. Cependant, ces formalismes ne sont pas intrinsèquement quantiques : la séparation et la fusion des bandes doivent être effectués d’une manière bien parenthésée rendant impossible la représentation de la superposition quantique. D’un point de vue sémantique, le calcul quantique pure avec coproduit et tenseurs ont été étudiés [DCM22; Cha+22], mais il n’est pas clair comment la notion des types inductifs et la récursion s’adapterait dans ces contextes et est, jusqu’à présent, la question est laissée ouverte.

Contents

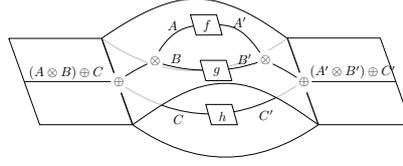


Figure 0.2.: Splits over coproduct and tensor

Le branchement des chemins d'exécution peut être utilisé pour déterminer quand un programme doit se terminer ou non. La notion de récursion et de types inductifs dans l'informatique quantique pure est encore une fois absente : afin de réaliser de la récursion, il faut s'assurer que chaque étape du calcul est réversible et que le programme récursif est toujours bien défini, car l'opération entière doit être une opération unitaire. Dans [SVV18], les auteurs développent un langage de programmation quantique pur avec le produit et coproduit et le type inductif des listes et avec de fortes contraintes assurant la terminaison et la réversibilité, ce qui ouvre la voie à une généralisation et le développement d'une logique propre.

Contributions. Nous essayons de répondre au problème de la recherche d'un modèle de calcul quantique pur avec un système de type de données riches, le tout avec une correspondance Curry-Howard. Nous proposons deux approches :

- Le premier, basé sur un langage de programmation purement quantique dans le style de Theseus [JS12; JS14; SVV18] où nous présentons un langage de programmation linéaire et réversible avec des types inductifs, ainsi qu'une correspondance une correspondance de Curry-Howard avec la logique μ MALL : logique linéaire avec les plus petits et plus grands points fixes. Là où [SVV18] étend le langage au cas quantique, avec le type limité de listes, nous restons dans le cas purement classique. L'extension pour type inductif quantique pur est laissée comme travail futur.
- La seconde consiste à l'utilisation des sémantiques à jetons inspiré de la Géométrie de l'Intéraction de Girard pour des *langages graphiques*. Nous montrons dans un premier temps comment capturer la sémantique dénotationnelle du ZX-Calcul à partir d'une sémantique à jeton. Nous développons ensuite un nouveau langage graphique, inspiré des réseaux de preuves de la logique linéaire, possédant à la fois un tenseur et un coproduit, permettant de gérer à la fois la collections de plusieurs données, mais aussi le branchement d'exécution. Le langage vient avec sa propre sémantique à jeton ainsi qu'une théorie équationnelle.

Plan de la thèse. La thèse est organisée comme en deux partie: la partie I introduit les notions nécessaires à la compréhension de cette thèse. Le chapitre 1 introduit les notions de calcul réversible et quantique nécessaire, tandis que le chapitre 2 introduit le ZX-Calcul : un langage de cordes pour le calcul quantique. Enfin, le chapitre 3 introduit

les notions de théorie de la démonstration nécessaire, en particulier en introduisant le formalisme de μ MALL: logique linéaire avec plus petits et plus grands point fixes.

La partie II est composée de trois chapitres sur mes contributions personnelles. Dans le chapitre 4 nous introduisons un langage linéaire et réversible basé sur le pattern-matching. Le langage est une extension du langage classique présenté dans [SVV18]. Le système de type du langage enforce des contraintes qui nous garantissent que n'importe quelle fonction bien typée (appelées isos) est réversible. Nous montrons (i) comment n'importe quelle fonction primitive récursive peut être encodée comme un iso du langage, (ii) comment n'importe quelle fonction bien typée représente un isomorphisme de preuve dans la logique μ MALL, et enfin (iii) nous montrons comment en relâchant les contraintes du système de type pour considérer des fonctions partielles, nous pouvons encoder les Machines de Turing Réversibles. Les programmes linéaires et réversibles étant un sous-ensemble du calcul quantique, ce chapitre est un premier pas vers des types purement quantique et du contrôle quantique.

Le chapitre 5 s'intéresse à la définition d'une sémantique à jetons pour le ZX-Calcul. La sémantique est inspirée des sémantiques à jetons de la Géométrie de l'Interaction de la logique linéaire, dans lesquels des jetons se déplacent dans le réseaux de preuve, vu comme un graphe, tout en capturant le contenu calculatoire de la preuve. Dans ce chapitre, nous montrons comment notre machine à jetons (i) sous certains invariants, capture exactement la sémantique dénotationnelle du ZX-Calcul, (ii) comment la machine à jetons peut être adaptée pour capturer l'extension du ZX-Calcul avec mesure, et (iii) comment la modifier pour capturer un autre type de sémantique du ZX-Calcul basé sur les sum-over-paths. Le but du chapitre est de voir comment des sémantiques à base de jetons peuvent être adaptées dans le cas des langages graphiques pour le calcul quantique, pour pouvoir par la suite se rapprocher des réseaux de preuves de la logique linéaire. Des travaux existent déjà sur les sémantiques à jetons pour le calcul quantique [Dal17], mais restent dans le cadre des circuits quantiques.

Pour finir, dans le chapitre 6 nous présentons un nouveau langage graphique nommé le Many-Worlds Calculus. A l'inverse du ZX-Calcul qui ne contient qu'un tenseur, le Many-Worlds contient à la fois un tenseur et un coproduit. Il permet d'encoder de manière naturelle l'ensemble du ZX-Calcul, mais possède en plus un système de type plus riche, dû au coproduit, qui nous permet de réaliser des tests quantique. Le chapitre est découpé en plusieurs parties, (i) nous donnons d'abord une sémantique à jetons, adaptée de celle du chapitre précédent, (ii) nous donnons une sémantique dénotationnelle qui a été déduite de la machine à jetons, (iii) nous donnons une théorie équationnelle sur les diagrammes du Many-Worlds Calculus, et nous montrons l'universalité du langage ainsi que sa complétude. Comme étude de cas nous montrons comment encoder le Quantum Switch dans le langage. Enfin, en (iv) nous montrons comment nous pouvons encoder une version quantique, sans récursion, du langage d'isos du chapitre 4 dans le Many-Worlds.

Part I.

Background

Chapter 1.

Reversible & Quantum Computation

1.1. Reversible Computation

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. Indeed, Landauer’s principle states that the erasure of information is linked to the dissipation of energy as heat [Lan61; Bér+12]. This principle was enough to motivate the study of reversible processes. 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 reversible computation, given some process f , there always exists an inverse process f^{-1} such that their composition is equal to the identity: $f \circ f^{-1} = \text{Id} = f^{-1} \circ f$. In programming languages, this is done by ensuring both *forward* and *backward* determinism. Forward determinism is almost always ensured in programming language: 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 checks that given a state, there is only one state that it comes from. Standard programming languages does not ensure backward determinism. On a computational perspective, the Toffoli gate [Tof80], is enough to realize universal classical computation: any boolean function can be implemented by Toffoli gates. The Toffoli gate is a reversible 3-input 3-output gate which flips the 3rd input if and only if the first two are at 1. This also means that when implementing a classical, reversible computation with only Toffoli gates, additional information is necessarily kept. Nevertheless, there exists a way to turn a non-reversible process into a reversible one, without additional information at the end (but with auxiliary wires), albeit with an increase in computational time [Ben73]. All of this led to an interest in reversible computation [Ben00; Ama+20], both with a low-level approach [Car12; Wil+16; 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.

In general, reversible computation captures *partial injective maps* [GKY19] from inputs to outputs. Indeed, from a computational perspective reversibility is understood as a time-local property: if each time-step of the execution of the computation can soundly be reversed, there is no overall condition on the global behaviour of the computation. In particular, this does not say anything about termination: a computation seen as a map from inputs to outputs might very well be partial, as some inputs may trigger a (global) non-terminating behaviour.

The categorical analysis of partial injective maps has been thoroughly analysed since 1979, first by Kastl [Kas79], and then by Cockett and Lack [CL02; CL03; CL07]. This led to the development of *inverse categories*: a category equipped with an inverse operator in which all morphisms have partial inverses and are therefore reversible. The main aspect of this line of research is that partiality can have a purely algebraic description: one can introduce a restriction operator on morphisms, associating to a morphism a partial identity on its domain.

This categorical framework has recently been put to use to develop the semantics of specific reversible programming constructs and concrete reversible languages: analysis of recursion in the context of reversibility [AK16; Kaa19b; KV19], formalization of reversible flowchart languages [GK18; Kaa19a], analysis of side effects [HK15; HKK18], *etc.* Interestingly enough however, the adequacy of the developed categorical constructs with reversible *functional* programming languages has seldom been studied. For instance, if Kaarsgaard *et al.* [KAG17] mention Theseus as a potential use-case, they do not discuss it in detail. So far, the semantics of functional and applicative reversible languages has always been done in *concrete* categories of partial isomorphisms [KV19; KR21; CLV21].

We present two models of reversible computation that will be useful at some point in this thesis, the first one is called RPP [PPR20] (Reversible Primitive Permutations): a set of reversible functions that allows to represent any primitive recursive function, and Reversible Turing Machine [AG11a; MY07].

1.1.1. Reversible Primitive Permutations

RPP is a set of integer-valued functions of variable arity. We define it by arity as follows: we note RPP^k for the set of functions in RPP from \mathbb{Z}^k to \mathbb{Z}^k , it is built inductively on $k \in \mathbb{N}$ by:

- the successor (S), the predecessor (P), the identity (ID) and the sign-change that are part of RPP^1 .
- The swap function (\mathcal{X}) is part of RPP^2 .

$$\begin{array}{cccccc}
 x [S] x + 1 & x [P] x - 1 & x [\text{Sign}] -x & x [\text{Id}] x & x \left[\begin{array}{c} \mathcal{X} \\ y \end{array} \right] y & \\
 & & & & & \\
 & & & & & \\
 x_1 \left[\begin{array}{c} y_1 \\ \vdots \\ f; g \\ y_n \\ x \end{array} \right] & = & x_1 \left[\begin{array}{c} y_1 \\ \vdots \\ f \\ y_n \end{array} \right] \left[\begin{array}{c} y_1 \\ \vdots \\ g \\ y_n \end{array} \right] & & x_1 \left[\begin{array}{c} y_1 \\ \vdots \\ \text{It}[f] \\ y_n \\ x \end{array} \right] & \left. \vphantom{x_1} \right\} = \underbrace{(f; \dots; f)}_{|x|} (x_1, \dots, x_n) \\
 & & & & & \\
 x_1 \left[\begin{array}{c} y_1 \\ \vdots \\ \text{If}[f, g, h] \\ y_n \\ x \end{array} \right] & \left. \vphantom{x_1} \right\} = \begin{cases} f(x_1, \dots, x_n) & \text{if } x > 0 \\ g(x_1, \dots, x_n) & \text{if } x = 0 \\ h(x_1, \dots, x_n) & \text{if } x < 0 \end{cases} & & x_1 \left[\begin{array}{c} y_1 \\ \vdots \\ x_k \text{ } f \parallel g \\ x'_1 \\ \vdots \\ x'_l \end{array} \right] \left[\begin{array}{c} y_1 \\ \vdots \\ y_k \\ y'_1 \\ \vdots \\ y'_l \end{array} \right] & = & x_1 \left[\begin{array}{c} y_1 \\ \vdots \\ f \\ y_n \\ y'_1 \\ \vdots \\ g \\ y'_l \end{array} \right]
 \end{array}$$

Figure 1.1.: Generators of RPP

- For any function $f, g, h \in \text{RPP}^k$ and $j \in \text{RPP}^l$, we can build (i) the sequential composition $f; g \in \text{RPP}^k$, (ii) the parallel composition $f \parallel j \in \text{RPP}^{k+l}$ (iii) the iterator $\text{It}[f] \in \text{RPP}^{k+1}$ and (iv) the selection $\text{If}[f, g, h] \in \text{RPP}^{k+1}$.

The behaviour of the functions is described under a circuit-like form, as in [PPR20], where the left-hand-side variables of the diagram represent the input of the function and the right-hand-side is the output of the function. The functions are shown in Figure 1.1.

Finally, the set of all functions that form RPP is taken as the union for all k all of the RPP^k :

$$\text{RPP} = \bigcup_{k \in \mathbb{N}} \text{RPP}^k$$

Remark 1.1.1. In their paper [PPR20], the authors make use of two other constructors: generalized permutations over \mathbb{Z}^k and weakenings of functions, but those can actually be defined from the other constructors [PPR20, Section 3] so that in the following section we do not give their encoding.

Then, if $f \in \text{RPP}^k$ we can define an inverse f^{-1} :

Definition 1.1.2 (Inversion). *The inversion is defined as follows:*

$$\begin{array}{lll} \text{Id}^{-1} = \text{Id} & \text{S}^{-1} = \text{P} & \text{P}^{-1} = \text{S} \\ \text{Sign}^{-1} = \text{Sign} & \mathcal{X}^{-1} = \mathcal{X} & (g; f)^{-1} = f^{-1}; g^{-1} \\ (f \parallel g)^{-1} = f^{-1} \parallel g^{-1} & (\text{It}[f])^{-1} = \text{It}[f^{-1}] & (\text{If}[f, g, h])^{-1} = \text{If}[f^{-1}, g^{-1}, h^{-1}] \end{array}$$

Proposition 1.1.3 (Inversion defines an inverse [PPR20]). *Given $f \in \text{RPP}^k$ then $f; f^{-1} = \text{Id} = f^{-1}; f$.* \square

Theorem 1.1.4 (Soundness & Completeness [PPR20]). *RPP is PRF-Complete and PRF-Sound: it can represent any Primitive Recursive Function and every function in RPP can be represented in PRF.* \square

1.1.2. Reversible Turing Machines

We start by introducing Reversible Turing Machine, following the formalism from [MY07; Ben73].

Definition 1.1.5 (Turing Machine). *We define the notion of a Turing Machine (TM) T as a tuple $(Q, \Sigma, \delta, b, q_s, q_f)$ where Q is a finite set of states, Σ a finite set of tape symbols, $b \in \Sigma$, the blank symbol and $\delta \subseteq \Delta = (Q \times ((\Sigma \times \Sigma) \cup \{\leftarrow, \downarrow, \rightarrow\})) \times Q$ is a partial relation defining the transition relation.*

*The states q_s and q_f are the starting and final state. There must be **no** transition leading out of q_f and no relation leading into q_s .*

Definition 1.1.6 (Configuration). *The configuration of a TM is a tuple $(q, (l, s, r)) \in Q \times (\Sigma^* \times \Sigma \times \Sigma^*)$ 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 TM T in configuration $C = (q, (l, s, r))$ leads to configuration $C' = (q', (l', s', r'))$, written as $T \vdash C \rightsquigarrow C'$ in a single computation step if there exists a transition $(q, a, q') \in \delta$ where a is either (s, s') , in which case $l = l'$ and $r = r'$ or $a \in \{\leftarrow, \downarrow, \rightarrow\}$ in which case 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 $r' = r$.

Definition 1.1.7 (Local forward/backward determinism). *A TM T is **local forward deterministic** if and only if for any distinct pair 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 $s_1 \neq s_2$.*

*A TM T is **local backward deterministic** if and only if for any distinct pair 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 1.1.8 (Reversible Turing Machine). *We say that a Turing Machine is reversible if and only if it is locally forward and backward deterministic.*

The transition relation works as follow:

If the transition is done by an element $(q, (s, s'), q') \in \delta$, then it means that if we are in state q and read s , we write s' and go in state q' . If the transition is (q, d, q') it means that if we are in state q , we move by direction d and go into state q' .

Definition 1.1.9 (String Semantics). *The semantic $\llbracket T \rrbracket$ of a TM is given by:*

$$\llbracket T \rrbracket = \{(s, s') \in ((\Sigma \setminus \{b\})^* \times (\Sigma \setminus \{b\})^*) \mid T \vdash (q_s, (\epsilon, b, s)) \rightsquigarrow^* (q_f, (\epsilon, b, s'))\}$$

The computation is as follows: from starting state q_s with input s , in a standard configuration $(q_s, (\epsilon, b, s))$ run the machine until it halts in a standard configuration $(q_f, (\epsilon, b, s'))$ with output s' , or diverges.

We say that T computes function f if and only if $\llbracket T \rrbracket = f$.

Theorem 1.1.10 (RTMs are injective [Ben73]). *If T is a RTM, then $\llbracket T \rrbracket$ is an injective function.*

Lemma 1.1.11 (RTM Inversion [Ben73]). *Given a RTM $T = (Q, \Sigma, \delta, b, q_s, q_f)$, define $T^{-1} = (Q, \Sigma, \text{inv}(\delta), b, q_f, q_s)$ as the inverse Turing Machine, where $\text{inv}(\delta)$ is defined as:*

- $\text{inv}(q, (s, s'), s') = (q', (s', s), q)$
- $\text{inv}(q, \leftarrow, q') = (q', \rightarrow, q)$
- $\text{inv}(q, \downarrow, q') = (q', \downarrow, q)$
- $\text{inv}(q, \rightarrow, q') = (q', \leftarrow, q)$

T^{-1} compute the inverse function, i.e $\llbracket T^{-1} \rrbracket = \llbracket T \rrbracket^{-1}$.

Finally, an important result is that any Turing Machine can be turned into a Reversible Turing Machine while preserving the semantics:

Theorem 1.1.12 (Bennett's method [Ben73]). *Given a 1-tape Turing Machine T , there exists a 3-tape reversible Turing Machine $B(T)$, such that $\llbracket B(T) \rrbracket (x) = (x, \llbracket T \rrbracket (x))$.*

Notice that T and $B(T)$ does not have the same exact the same semantics: in the output of $B(T)$ the initial input x is still present. As we stated earlier, reversible computation always come at the cost of having to keep in memory additional information to ensure reversibility.

Finally, for robustness, we will state that any k -tape RTM can be turned into a 1-tape RTM [AG11b].

1.2. Quantum Computation

Reversible computation makes an important subset of quantum computation without measurement. In both setting, all operations are reversible. In the quantum case, a reversible operation send basis vectors to basis vectors: it is a unitary operation. Quantum computation also requires operations to be linear, which can be considered in a linear, reversible computational model. The main difference arise in the data they handle: quantum computing have access to a new kind of data, *quantum bits*.



Figure 1.2.: The QRAM Model

The QRAM Model In general quantum computation, one has access to a coprocessor holding a “quantum” memory. This memory consists of “quantum” bits, the basic unit of information in quantum computation, having a peculiar property: their state cannot be duplicated, and the operations one can perform on them are unitary, reversible operations. The coprocessor comes with an interface to which one can send instructions to allocate, update or read quantum registers. Quantum memories can be used to solve classical problems faster than with purely conventional means. The most prominent examples being Grover’s algorithm, allowing to search for an element in a list of n element in $O(\sqrt{N})$ [Gro96] and its direct application in Shor’s algorithm, allowing for decomposing a number into its prime factor exponentially faster than classical algorithm [Sho99]. Quantum programming languages are nowadays pervasive [FBW18; VRH22] and several formal approaches based on logical systems have been proposed to relate to this model of computation [SV06; PRZ17; RS17]. However, all of these languages rely on a purely *classical* control-flow: quantum computation is reduced to describing a list of instructions—a quantum circuit—to be sent to the coprocessor. In particular, in this model, operations performed on the quantum memory only act on quantum bits and tensors thereof, while the classical computer enjoys the manipulation of any kind of data with the help of rich type systems. Reversible computation can be seen as a subcase of quantum computation. While reversible computation allows for the duplication and erasure of data, this is not the case in quantum computation, but in both cases the operations are always reversible.

Dirac Notation Dirac’s notation is a way to represent matrices and vectors in a more compact way. In Dirac notation [NC02], vectors of the form $|\cdot\rangle$ (called “*kets*”) are considered as column vector, and therefore $|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$, $|1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$, and given α, β complex numbers, $\alpha|0\rangle + \beta|1\rangle = \begin{pmatrix} \alpha \\ \beta \end{pmatrix}$. The tensor product of spaces \mathcal{V} and \mathcal{W} whose bases are respectively $\{v_i\}_{i \in I}$ and $\{w_j\}_{j \in J}$ is the vector space of basis $\{v_i \otimes w_j\}_{i,j \in I \times J}$, where $v_i \otimes w_j$

is a formal object consisting of a pair of v_i and w_j . We denote $|x\rangle \otimes |y\rangle$ as $|xy\rangle$ and $|0^m\rangle$ to represent an m -fold tensor of $|0\rangle$. As a shortcut notation, we write $|\phi\rangle$ for column vectors consisting of a linear combination of kets. Dirac also introduced the notation “bra” $\langle x|$, standing for a row vector. So for instance, $\alpha \langle 0| + \beta \langle 1|$ is $(\alpha \ \beta)$. If $|\phi\rangle = \alpha |0\rangle + \beta |1\rangle$, we then write $\langle \phi|$ for the vector $\bar{\alpha} \langle 0| + \bar{\beta} \langle 1|$ (with $\bar{(\cdot)}$ the complex conjugation). The notation for tensors of bras is similar to the one for kets. For instance, $\langle x| \otimes \langle y| = \langle xy|$. Using this notation, the scalar product is transparently the product of a row and a column vector: $\langle \phi | \psi \rangle$, and matrices can be written as sums of elements of the form $|\phi\rangle\langle\psi|$. For instance, the identity on \mathbb{C}^2 is $\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} + \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \begin{pmatrix} 1 & 0 \end{pmatrix} + \begin{pmatrix} 0 \\ 1 \end{pmatrix} \begin{pmatrix} 0 & 1 \end{pmatrix} = |0\rangle\langle 0| + |1\rangle\langle 1|$.

Quantum State As quantum computation works with vectors and matrices, we take the two column vectors $|0\rangle$ and $|1\rangle$ as the counterpart of the classical bits 0 and 1. Richer states of multiple bits are built using tensor products of states. Remember that the tensor product between matrices is defined as:

$$A \otimes B = \begin{pmatrix} a_{00}B & a_{01}B & \cdots \\ a_{10}B & \ddots & \\ \vdots & & \end{pmatrix}$$

and hence $|00\rangle = |0\rangle \otimes |0\rangle = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}$. Quantum state can be in *superposition* by considering a linear combination of classical bits, written $\alpha |0\rangle + \beta |1\rangle$, and where $\alpha, \beta \in \mathbb{C}$. Quantum computation asks for the state to be *normalized*, meaning that $|\alpha|^2 + |\beta|^2 = 1$. In general, a quantum state can be written as $\sum_{i \in I} \alpha_i |i\rangle$ with $\sum_{i \in I} |\alpha_i|^2 = 1$.

Not all quantum states can be written as a tensor of smaller states. Those inseparable states are called *entangled state*. For example, the *Bell State* $\frac{|00\rangle + |11\rangle}{\sqrt{2}}$ cannot be written as the tensors of two other states.

While $|0\rangle, |1\rangle$ form a computational basis, we could also consider the basis $|+\rangle, |-\rangle$ defined as $|+\rangle = \frac{|0\rangle + |1\rangle}{\sqrt{2}}$ and $|-\rangle = \frac{|0\rangle - |1\rangle}{\sqrt{2}}$. One way to go from one basis to the other is via the Hadamard gate.

Operations & Quantum Circuits Quantum computation is restricted to a particular kind of operations called *unitaries*: matrices whose conjugate transpose U^* is also its inverse, U^{-1} . Any unitary can be written as $\sum_{i,j \in B} \alpha_{i,j} |i\rangle\langle j|$ for B an orthogonal basis of the state space under consideration.

A quantum circuit is the quantum counterpart of boolean circuits: a graphical language where wires represent qubits and gates represent unitary operations. Qubits flow from the left of the circuit towards the right, updating their state as they traverse quantum

gates. A quantum circuit is then a single unitary operation which is made by the composition of the smaller gates inside it.

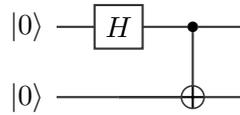


Figure 1.3.: Bell State's quantum circuit

A standard universal set consists of the one-input, one-output Pauli-X, -Y and -Z gates, the Hadamard gate and the Phase Shift $P(\phi)$ gate, represented in a circuit by the diagram $\boxed{\#}$ with $\# \in \{X, Y, Z, H, P(\phi)\}$ and corresponding respectively to the matrices $\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}, \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}, \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}, \begin{pmatrix} 1 & 0 \\ 0 & e^{i\phi} \end{pmatrix}$. Along with the two-input two-output CNOT gate $\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$ represented by

For example, the circuit in Figure 1.3 takes for input two qubits initialized at $|0\rangle$, apply the Hadamard gate on the first qubit and then applies a CNOT in order to produce the Bell State $\frac{|00\rangle + |11\rangle}{\sqrt{2}}$.

Measurement One distinct feature of quantum computation is *measurement*: the collapse of a superposition of states into a single one with some probability. For instance, in the qubit case, applying a measurement on the state $\alpha |0\rangle + \beta |1\rangle$ give $|0\rangle$ (resp. $|1\rangle$) with probability $|\alpha|^2$ (resp. $|\beta|^2$). More generally, given a quantum state $\sum_{i \in B} \alpha_i |i\rangle$ with B a orthogonal basis, a measurement on such a state gives us the state $|i\rangle$ with probability $|\alpha_i|^2$. The standard semantics for measurement is to work with *density matrices* and *completely positive maps* (CPM).

In a quantum circuit the measurement can *always* be done at the end. This is called the Deferred Measurement Principle [NC02]. Therefore, for this thesis we will focus purely on the unitary part and ignore the measurement.

More information and details can be found in [NC02].

Chapter 2.

Graphical Language for Quantum Computation

The use of graphical representations for computation can be found in many fields of computer science, the most basic example being the control flow of a program, where one can initialize a *state* of the machine, in which each variable is given a value of its type, and look at the order of instructions that are being executed by the program. A more formal one would be boolean circuits: graphical representations of an assembly-like language in which the only connectives are the usual logical ones (negation, conjunction and disjunction), read from left to right in which the semantics is given by considering some *tokens* (the bits) that flow from the input of the diagram all the way to the output and whose values change according to the gate they enter. Similarly, quantum circuits (quantum counterparts of boolean circuits) work in the same way, with qubits instead of bits flowing through the circuit. Although successful, those formalisms are akin to assembly-like languages which makes it hard to use and to reason about.

Diagrammatic languages for describing the mathematical behaviour of quantum processes were already present in the 1940's with Feynman's diagrams but, in this section, we are interested in a particular kind of graphical languages: *string diagrams*. String diagrams are a graphical formalism that allows for a 2-dimensional representation of categorical syntax. Coming all the way from Feynman diagrams [FH65], a graphical language is an alternative syntax to the usual matrix representation of quantum computation. String diagrams for quantum computation emerged from the Categorical Quantum Mechanics program by Abramsky and Coecke [AC04; AC09]. Their goal was to be able to study properties of quantum processes in a more abstract way through the use of category theory. This led to the development of the ZX-Calculus [CD11], a graphical language for quantum computation. On a formal level, a graphical language is a PROP [Lac04], that is, a symmetric, strict monoidal structure $(\mathcal{C}, \top, \boxtimes)$ whose objects are of the form $W \boxtimes \dots \boxtimes W$. The object W is a “wire”, and any object stands for a bunch of wires. The monoidal structure formalizes how the bunching of wires behaves. For the purpose of this thesis, we introduce the ZX-Calculus purely as a graphical language. For further bibliography on the subject we refer to [Vil19; Sel10; JS91; CD11; CK17].

2.1. The ZX-Calculus

Although the pervasive model for quantum computation, quantum circuits, only have an informal semantics: qubits travelling from the inputs wires to the outputs wires, changing their state as they pass through the gates of the circuit. A quantum circuit is understood as some sequential, low-level assembly language where quantum gates are opaque black boxes. In particular, quantum circuits do not natively feature any formal operational semantics giving rise to abstract reasoning or well-founded rewriting system, and did not, until recently [Clé+22a], feature an equational theory, even though the current one is not very usable.

From a denotational perspective, quantum circuits are literal descriptions of tensors and applications of linear operators. These can be described with the original matrix interpretation [NC02], or with the more recent sum-over-path semantics [Amy18; Cha+]—this can be regarded as a *wave-style semantics*. In such a semantics, the state of all of the quantum bits of the memory is mathematically represented as a vector in a (finite dimensional) Hilbert space: the set of quantum bits is a *wave* flowing in the circuit, from the inputs to the output, while the computation generated by the list of quantum gates is a linear map from the Hilbert space of inputs to the Hilbert space of outputs.

In recent years, an alternative model of quantum computation with better formal properties than quantum circuits have emerged: the ZX-Calculus [CD11]. Originally motivated by a categorical interpretation of quantum theory, the ZX-Calculus is a graphical language that represents linear maps as special kinds of graphs called *diagrams*. Unlike the quantum circuit framework, the ZX-Calculus comes with a sound and complete [Vil19], well-defined equational theory on a small set of canonical generators making it possible to reason on quantum computation by means of local graph rewriting.

The canonical semantics of a ZX diagram consists in a linear operator. This operator can be represented as a matrix or through the more recent sum-over-path semantics [Vil20]. But in both cases, these semantics give a purely functional, *wave-style* interpretation to the diagram. Nonetheless, this graphical language—and its equational theory—has been shown to be amenable to many extensions and is being used in a wide spectrum of applications ranging from quantum circuit optimization [Dun+20; Bac+20], verification [Hil11; DL14; DG18] and representation such as MBQC patterns [DP10] or error-correction [BH20; Bea+19].

The ZX-Calculus is a powerful graphical language for reasoning about quantum computation introduced by Bob Coecke and Ross Duncan [CD11]. A term in this language is a graph—called a *string diagram*—built from a core set of primitives. In the standard interpretation of ZX-Calculus, a string diagram is interpreted as a matrix. The language is equipped with an equational theory preserving the standard interpretation.

2.1.1. Pure Operators

The so-called *pure* ZX-diagrams are generated from a set of primitives, given on the right: the Identity, Swap, Cup, Cap, Green-spider and H-gate:

$$\left\{ \text{Identity}, \text{Swap}, \text{Cup}, \text{Cap}, \text{Green-spider}(n, m, \alpha), \text{H-gate} \right\}_{\substack{n, m \in \mathbb{N} \\ \alpha \in \mathbb{R}}}$$

The real number α attached to the green spiders is called the *angle*, we generally write $Z_m^n(\alpha)$ for a green-spider with n inputs, m outputs and angle α . ZX-diagrams are read top-to-bottom: dangling top edges are the *input edges* and dangling edges at the bottom are *output edges*. For instance, Swap has 2 input and 2 output edges, while Cup has 2 input edges and no output edges. ZX-primitives can be composed either sequentially or in parallel:

$$D_2 \circ D_1 := \begin{array}{c} \dots \\ \boxed{D_1} \\ \dots \\ \boxed{D_2} \\ \dots \end{array} \qquad D_1 \otimes D_2 := \begin{array}{cc} \dots & \dots \\ \boxed{D_1} & \boxed{D_2} \\ \dots & \dots \end{array}$$

We write \mathbf{ZX} for the set of all ZX-diagrams. Notice that when composing diagrams with $(_ \circ _)$, we “join” the outputs of the top diagram with the inputs of the bottom diagram. This requires that the two sets of edges have the same cardinality.

Convention 2.1.1. We define a second spider, red this time, by composition of Green-spiders and H-gates, as shown on the right, similarly for the green-spider, we write the red-spider as $X_m^n(\alpha)$.

$$\begin{array}{c} (\dots) \\ \text{red spider} \\ (\dots) \end{array} \alpha := \begin{array}{c} \dots \\ \text{H-gate} \\ \dots \\ \text{Green-spider} \\ \dots \end{array} \alpha$$

Convention 2.1.2. We write σ for a permutation of wires, i.e any diagram generated by $\left\{ \text{Identity}, \text{Swap} \right\}$ with sequential and parallel composition. The Cap and Cup are written respectively as η and ϵ . We write $Z_k^n(\alpha)$ (resp, $X_k^n(\alpha)$) for the green-node (resp, red-node) of n inputs, k outputs and parameter α and H for the H-gate. Finally, by abuse of notation a green or red node with no explicit parameter holds the angle 0:

$$\begin{array}{c} (\dots) \\ \text{green node} \\ (\dots) \end{array} := \begin{array}{c} (\dots) \\ \text{green node} \\ (\dots) \end{array} 0 \quad \text{and} \quad \begin{array}{c} (\dots) \\ \text{red node} \\ (\dots) \end{array} := \begin{array}{c} (\dots) \\ \text{red node} \\ (\dots) \end{array} 0.$$

Formally, the ZX-Calculus is a *dagger compact category* and is in particular a *PROP*: objects are natural numbers, and morphisms are the diagrams [CD11]. For more information on the categorical background of the ZX-Calculus and other graphical languages we refer to [Sel10; CK17].

2.1.2. Standard Interpretation

We understand ZX-diagrams as linear operators through the *standard interpretation*. Informally, wires are interpreted with the two-dimensional Hilbert space, with orthonormal basis $\{|0\rangle, |1\rangle\}$. In the standard interpretation [CD11], a diagram D is mapped to a finite dimensional Hilbert space of dimension some powers of 2: $\llbracket D \rrbracket \in \mathbf{Qubit} := \{\mathbb{C}^{2^n} \rightarrow \mathbb{C}^{2^m} \mid n, m \in \mathbb{N}\}$.

If D has n inputs and m outputs, its interpretation is a map $\llbracket D \rrbracket : \mathbb{C}^{2^n} \rightarrow \mathbb{C}^{2^m}$ (by abuse of notation we shall use the notation $\llbracket D \rrbracket : n \rightarrow m$). It is defined inductively as follows.

$$\begin{aligned} \llbracket \begin{array}{|c|} \hline \dots \\ \hline D_1 \\ \hline \dots \\ \hline D_2 \\ \hline \dots \\ \hline \end{array} \rrbracket &= \llbracket \begin{array}{|c|} \hline \dots \\ \hline D_2 \\ \hline \dots \\ \hline \end{array} \rrbracket \circ \llbracket \begin{array}{|c|} \hline \dots \\ \hline D_1 \\ \hline \dots \\ \hline \end{array} \rrbracket & \llbracket \begin{array}{|c|c|} \hline \dots & \dots \\ \hline D_1 & D_2 \\ \hline \dots & \dots \\ \hline \end{array} \rrbracket &= \llbracket \begin{array}{|c|} \hline \dots \\ \hline D_1 \\ \hline \dots \\ \hline \end{array} \rrbracket \otimes \llbracket \begin{array}{|c|} \hline \dots \\ \hline D_2 \\ \hline \dots \\ \hline \end{array} \rrbracket \\ \llbracket \begin{array}{|c|} \hline | \\ \hline \end{array} \rrbracket &= id_{\mathbb{C}^2} = |0\rangle\langle 0| + |1\rangle\langle 1| & \llbracket \begin{array}{|c|} \hline \times \\ \hline \end{array} \rrbracket &= \sum_{i,j \in \{0,1\}} |ji\rangle\langle ij| \\ \llbracket \begin{array}{|c|} \hline \cap \\ \hline \end{array} \rrbracket &= \llbracket \begin{array}{|c|} \hline \cup \\ \hline \end{array} \rrbracket^\dagger = |00\rangle + |11\rangle & \llbracket \begin{array}{|c|} \hline \oplus \\ \hline \end{array} \rrbracket &= |+\rangle\langle 0| + |-\rangle\langle 1| \\ \llbracket \begin{array}{|c|} \hline \overset{n}{\underbrace{\dots}} \\ \hline \bullet \\ \hline \underbrace{\dots}{m} \\ \hline \end{array} \rrbracket &= |0^m\rangle\langle 0^n| + e^{i\alpha} |1^m\rangle\langle 1^n| & \llbracket \begin{array}{|c|} \hline \overset{n}{\underbrace{\dots}} \\ \hline \bullet \\ \hline \underbrace{\dots}{m} \\ \hline \end{array} \rrbracket &= |+\rangle\langle +^n| + e^{i\alpha} |-\rangle\langle -^n| \end{aligned}$$

Intuitively:

- The Hadamard gate is the standard one from quantum computation: $\frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$, that sends the state $|0\rangle$ to $\frac{|0\rangle+|1\rangle}{\sqrt{2}}$ and $|1\rangle$ to $\frac{|0\rangle-|1\rangle}{\sqrt{2}}$.
- The cap creates an entangled pair of $|00\rangle$ and $|11\rangle$, while the cup is its dual: asking both element of the $|xy\rangle$ to be the same, either 0 or 1.
- The green-spider can be seen as a map that checks that all its input are the same, if its input is $|0^n\rangle$ it returns $|0^m\rangle$, if they are all $|1^n\rangle$ it returns $e^{i\alpha} |1^m\rangle$ and otherwise returns the 0 vector.
- The red-spider is the same but on the $|+\rangle, |-\rangle$ basis.

Example 2.1.3. The states $|0\rangle$ and $|1\rangle$ can be encoded as:  and 

A maybe more intuitive way to look at it is when considering a simpler green and red-spider where the green-spider has one input and n outputs and the red-spider have n inputs and one output. Then, the green spider can be seen as a copy operation on classical data while the red spider can be seen as a XOR:

$$\begin{aligned} \begin{array}{|c|} \hline \bullet \\ \hline \dots \\ \hline \end{array} \text{ s.t. } \begin{array}{|c|} \hline \bullet \\ \hline \dots \\ \hline \end{array} \overset{k\pi}{=} \begin{array}{|c|c|} \hline \bullet & \bullet \\ \hline \dots & \dots \\ \hline \end{array} & \begin{array}{|c|} \hline \dots \\ \hline \bullet \\ \hline \dots \\ \hline \end{array} \text{ s.t. } \begin{array}{|c|c|c|} \hline \overset{k_1\pi}{\bullet} & \dots & \overset{k_n\pi}{\bullet} \\ \hline \bullet & & \\ \hline \end{array} \overset{\sum_j k_j\pi}{=} \begin{array}{|c|} \hline \bullet \\ \hline \dots \\ \hline \end{array} \quad \text{for } k, k_1, \dots, k_n \in \{0, 1\} \end{aligned}$$

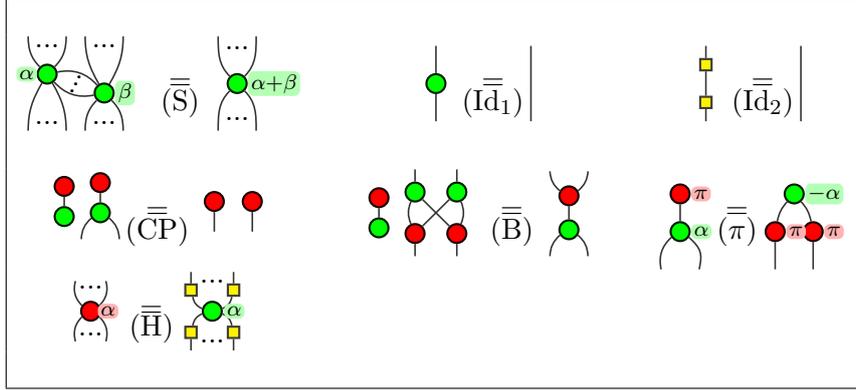
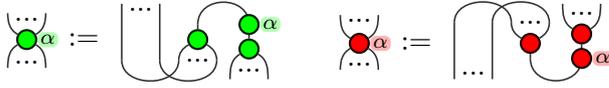


Figure 2.1.: Minimal equational theory of the ZX-Calculus

From those, we can recover the general green and red spider:



Example 2.1.4. *The CNOT gate (up to some scalar) in the ZX-Calculus can be defined as:*

$$\left[\begin{array}{c} \text{green spider} \\ \text{red spider} \end{array} \right] = \left(\left[\begin{array}{c} | \\ | \end{array} \right] \otimes \left[\begin{array}{c} \text{red spider} \end{array} \right] \right) \circ \left(\left[\begin{array}{c} \text{green spider} \end{array} \right] \otimes \left[\begin{array}{c} | \\ | \end{array} \right] \right) = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}.$$

2.1.3. Properties and structure

In this section, we list several definitions and known results that we shall use in the remainder of the thesis. See e.g. [Vil19] for more information.

Universality ZX-diagrams are *universal* [CD11] in the sense that for any linear map $f : n \rightarrow m$, there exists a diagram D of **ZX** such that $\llbracket D \rrbracket = f$.

The price to pay for universality is that different diagrams can possibly represent the same quantum operator, for instance we have that $\left[\begin{array}{c} | \\ | \end{array} \right] = \left[\begin{array}{c} | \\ | \end{array} \right]$. There exists however a

way to deal with this problem: an equational theory. We give in Figure 2.1 a complete axiomatization of the standard ZX-Calculus. This equational theory is not the only one that exists for the ZX-Calculus, also several equational theories have been designed for various fragments of the language [Bac14; JPV18a; HNW18; JPV18b; JPV19; Vil19].

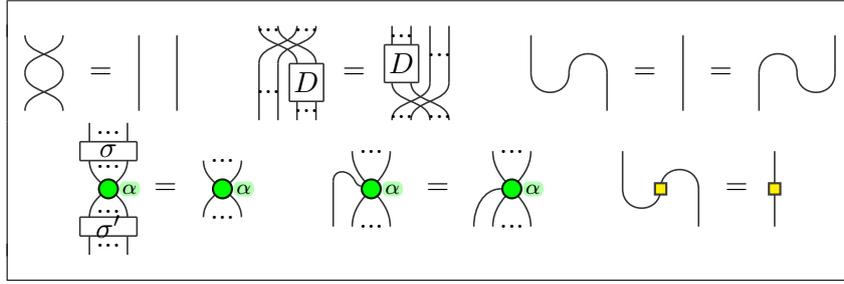
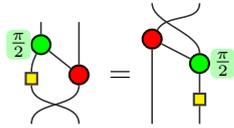


Figure 2.2.: Connectivity rules. D represents any ZX-diagram, and σ, σ' any permutation of wires.

One important aspect of the equational theory is that ZX-diagrams can be seen as open graphs. Therefore, any graph isomorphism is a valid derivation in the equational theories. For example:



Core axiomatization Despite this variety, any ZX axiomatization builds upon the core set of equations provided in Figure 2.2, meaning that edges really behave as wires that can be bent, tangled and untangled. They also enforce the irrelevance on the ordering of inputs and outputs for spiders. Most importantly, these rules preserve the standard interpretation given in Section 2.1.2. These rules are sometimes referred to as “*only connectivity matters*”, and they preserve the semantics, most of the time people consider diagrams modulo those rules. Those rules are the one given by the underlying PROP structure.

Completeness The ability to transform a diagram D_1 into a diagram D_2 using the rules of some axiomatization zx (e.g. the core one presented in Figure 2.1) is denoted $\text{zx} \vdash D_1 = D_2$.

The axiomatization is said to be *complete* whenever any two diagrams representing the same operator can be turned into one another using this axiomatization. Formally:

$$\llbracket D_1 \rrbracket = \llbracket D_2 \rrbracket \iff \text{zx} \vdash D_1 = D_2$$

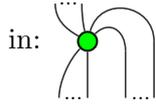
The first complete axiomatization of the ZX-Calculus was provided in [Vil19].

It is common in quantum computing to work with restrictions of quantum mechanics. Such restrictions translate to restrictions to particular sets of diagrams – e.g., the $\frac{\pi}{4}$ -fragment which consists of all ZX-diagrams where the angles are multiples of $\frac{\pi}{4}$. There exist axiomatizations that were proven to be complete for the corresponding fragment (all of the aforementioned references tackle the problem of completeness).

Input and output wires An important result from quantum computation that translates nicely is the ZX-Calculus is the following:

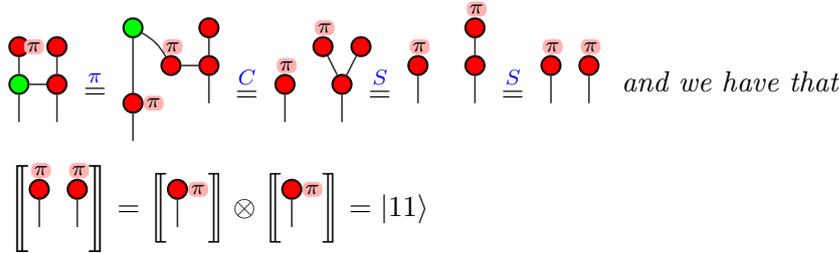
Theorem 2.1.5 (Choi-Jamiołkowski). *There are isomorphisms between $\{D \in \mathbf{ZX} \mid D : n \rightarrow m\}$ and $\{D \in \mathbf{ZX} \mid D : n - k \rightarrow k + m\}$ (when $k \leq n$).* \square

To see how this can be true, simply add cups or caps to turn input edges to output edges (or vice versa), and use the fact that we work modulo the rules of Figure 2.2 as



When $k = n$, this isomorphism is referred to as the *map/state duality*. A related but more obvious isomorphism between ZX-diagrams is obtained by permutation of input wires (resp. output wires).

Example 2.1.6. *The CNOT diagram composed with the state $|10\rangle$ and its reduction in the ZX-Calculus:*



2.1.4. ZX-diagrams for Mixed Processes

While the ZX-Calculus allows to represent *pure* quantum computation, it is possible to consider its extension to mixed processes, allowing to represent measurement by adding a unary generator $\underline{\underline{=}}$ to the language [CP12; Car+19], that intuitively enforces the state of the wire to be classical. We denote with \mathbf{ZX}^{\neq} the set of diagrams obtained by adding the $\underline{\underline{=}}$ generator.

Similar to what is done in quantum computation, the standard interpretation $\llbracket \cdot \rrbracket^{\neq}$ for \mathbf{ZX}^{\neq} maps diagrams to CPMs. If $D \in \mathbf{ZX}$ we define $\llbracket D \rrbracket^{\neq}$ as $\rho \mapsto \llbracket D \rrbracket^{\dagger} \circ \rho \circ \llbracket D \rrbracket$, and we set $\llbracket \underline{\underline{=}} \rrbracket^{\neq}$ as $\rho \mapsto \text{Tr}(\rho)$, where $\text{Tr}(\rho)$ is the trace of ρ .

There is a canonical way to map a \mathbf{ZX}^{\neq} -diagram to a \mathbf{ZX} -diagram in a way that preserves the semantics: the so-called CPM-construction [Sel07]. We define the map (conveniently named) CPM as the map that preserves compositions $(-\circ-)$ and $(-\otimes-)$ and such that:

$$\begin{aligned} \text{CPM} \left(\begin{array}{|c|} \hline \dots \\ \hline D_1 \\ \hline \dots \\ \hline D_2 \\ \hline \dots \\ \hline \end{array} \right) &= \text{CPM} \left(\begin{array}{|c|} \hline \dots \\ \hline D_2 \\ \hline \dots \\ \hline \end{array} \right) \circ \text{CPM} \left(\begin{array}{|c|} \hline \dots \\ \hline D_1 \\ \hline \dots \\ \hline \end{array} \right) \\ \text{CPM} \left(\begin{array}{|c|} \hline \dots \\ \hline D_1 \\ \hline \dots \\ \hline \end{array} \right) \begin{array}{|c|} \hline \dots \\ \hline D_2 \\ \hline \dots \\ \hline \end{array} &= \text{CPM} \left(\begin{array}{|c|} \hline \dots \\ \hline D_1 \\ \hline \dots \\ \hline \end{array} \right) \otimes \text{CPM} \left(\begin{array}{|c|} \hline \dots \\ \hline D_2 \\ \hline \dots \\ \hline \end{array} \right) \\ \text{CPM} (|) &= || \qquad \text{CPM} (\times) = \times \\ \text{CPM} (\cup) &= \cup \qquad \text{CPM} (\cap) = \cap \qquad \text{CPM} (\underline{=}) = \cup \\ \text{CPM} \left(\begin{array}{|c|} \hline \dots \\ \hline \alpha \\ \hline \dots \\ \hline \end{array} \right) &= \begin{array}{|c|} \hline \dots \\ \hline \alpha \\ \hline \dots \\ \hline \end{array} \begin{array}{|c|} \hline \dots \\ \hline -\alpha \\ \hline \dots \\ \hline \end{array} \qquad \text{CPM} \left(\begin{array}{|c|} \hline \dots \\ \hline \alpha \\ \hline \dots \\ \hline \end{array} \right) = \begin{array}{|c|} \hline \dots \\ \hline \alpha \\ \hline \dots \\ \hline \end{array} \begin{array}{|c|} \hline \dots \\ \hline -\alpha \\ \hline \dots \\ \hline \end{array} \qquad \text{CPM} (\square) = \square \end{aligned}$$

CPM(D) has to be understood as two copies of D where $\underline{=}$ is replaced by \cup and where every angle α is changed to $-\alpha$ in the second copy. Indeed, a CPM operation from $A \rightarrow B$ can be seen as a unitary operation from $A \otimes A^* \rightarrow B \otimes B^*$, this is what is done in this transformation.

In the general ZX-Calculus, it has been shown that the axiomatization itself could be extended to a complete one by adding only four axioms and is sound and complete [Car+19].

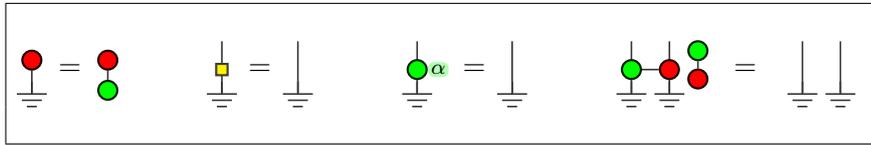
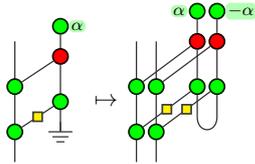


Figure 2.3.: Additional rules for $\underline{=}$. Together with zx, they form the equational theory $\text{zx}^{\underline{=}}$.

Example 2.1.7. A $\text{ZX}^{\underline{=}}$ -diagram and its associated CPM construction.



Chapter 3.

Proof Theory & The Curry Howard Isomorphism

3.1. Logic & Computation

In this chapter, we give a short background on the typed lambda-calculus whose notions we will use in Chapter 4, and logic, and see how both notions are related by the Curry-Howard Correspondence. We also give a extension introduction to the logic μ MALL, an extension of linear logic with least and greatest fixed point.

3.1.1. The Simply-Typed λ -Calculus

Syntax Among the many computational models that exists, the one that impacted the most the theory and development of programming languages is without a doubt the λ -calculus. Developed by Alonzo Church in the 1930s [Chu36; Chu41]. The λ -calculus is a model of higher-order computation that manipulates λ -terms built upon the following syntax:

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

where x is a variable, taken from an infinite set of variables \mathcal{V} , the term $(t u)$ is the application of a term t (considered as a function) to another term u (the input of said function) and $\lambda x.t$ is a function declaration (also called a λ -abstraction) which binds the variable x inside t . Intuitively $\lambda x.t$ can be seen as an unnamed function $x \rightarrow t$. For instance $\lambda x.x$ can be seen as the identity function while $\lambda x.y$ as a constant function that always returns y .

Scope & The α -equivalence As said, $\lambda x.t$ acts as a binder that binds the variable x inside t , and so we say that x is *bound* inside t . On the opposite, we say that x is *free* inside t if it is not bound by an abstraction. More formally we can define the set of free-variables inside a term t by induction on t as $\text{FV}(x) = x$, $\text{FV}(t t') = \text{FV}(t) \cup \text{FV}(t')$, $\text{FV}(\lambda x.t) = \text{FV}(t) \setminus \{x\}$. A term with no free variable is said to be *closed* (also called a *combinator*).

Contexts A context is a term with a hole in it. If C is a context, then $C[t]$ is the result of filling the hole with t . For example, if we have the context $C = \lambda x. \square$ then $C[t] = \lambda x.t$. Contexts can be used when we wish to focus on one position in a term, they can be used in defining particular evaluation order. For instance, we can formulate the arbitrary context $C ::= \square \mid C M \mid M C \mid \lambda x.C$. Contexts are used to define how a term valuate through the β -reduction as we will see shortly.

Types Introduced by Church [Chu40], the Simply Typed λ -Calculus is an extension of the base λ -calculus in which terms are *typed*: annotated with some types, according to some *typing system*. A type system is a set of deduction rules allowing us to build *typing derivations*. Type systems usually use *typing context*: a set of pairs of term-variable and a type defined as $\Delta ::= \emptyset \mid x : A, \Delta$ where the comma represents a union and A is a type. In the simply typed λ -calculus, the types are defined by the grammar $o ::= o \mid o_1 \rightarrow o_2$ and a typing judgement is of the form $\Delta \vdash t : o$, which can be understood as: under context Δ , the term t has type o . The typing system is formally defined by the rules:

$$\frac{}{x : o, \Delta \vdash x : o} \quad \frac{x : o_1, \Delta \vdash t : o_2}{\Delta \vdash \lambda x.t : o_1 \rightarrow o_2} \quad \frac{\Delta \vdash t : o_1 \rightarrow o_2 \quad \Delta \vdash t' : o_1}{\Delta \vdash t t' : o_2}$$

Then, one can decide to only consider *well-typed terms*: terms who have a typing derivation defined inductively by the rules given above.

Computation Computation inside the λ -calculus works with β -reduction: when an abstraction $(\lambda x.t)$ is applied to some argument u , the β -reduction will produce the term t in which all occurrence of the free variable x inside t has been replaced by u , noted $t[x \leftarrow u]$. In the way we described the β -reduction we may end up with a problem called *variable capture*: consider $(\lambda x.y)[y \leftarrow x] = \lambda y.y$. We went from the constant functions that always returns y to the identity function. To avoid this problem, we consider the α -equivalence which put into relation terms with different bounded variables, for instance we can say that $\lambda x.x \equiv_\alpha \lambda z.z$ or that $(\lambda x.t) \equiv_\alpha (\lambda y.t[x \leftarrow y])$. It allows us to work up to renaming of bound variables. 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, even when this remains implicit. The β -reduction is then defined using the evaluation context as $C[(\lambda x.t)u] \rightarrow_\beta C[t[x \leftarrow u]]$. The λ -calculus, with β -reduction is Turing Complete.

An important property is that the β -reduction is *confluent*: if a term t can be reduced in two different ways by the β -reduction into terms t_1, t_2 , there always exists another term t' such that t_1 and t_2 reduce to t_3 . The typing system ensures us that when a term reduces, it keeps the same type and that well-typed terms always terminate: there is no infinite sequence of reduction. For instance, the term $\Omega = (\lambda x.xx)(\lambda x.xx)$ is *ill-typed* and its reduction does not terminate: $\Omega \rightarrow_\beta xx[x \leftarrow (\lambda x.xx)] = (\lambda x.xx)(\lambda x.xx)$

Example 3.1.1. Take the function that takes two arguments x and y and returns the second argument while erasing the first one: $\lambda x.\lambda y.y$ and applying it to any two arguments t, t' , we get: $((\lambda x.\lambda y.y)t)t' \rightarrow_{\beta} (\lambda y.y[x \leftarrow t])t' = (\lambda y.y)t' \rightarrow_{\beta} y[x \leftarrow t] = t'$

Example 3.1.2 (Booleans values in the λ -calculus). We can define the two truth values True (\mathbf{tt}) and False (\mathbf{ff}) as $\lambda x.\lambda y.x$ and $\lambda x.\lambda y.y$ and the Boolean operation AND as $\lambda x.\lambda y.x y \mathbf{ff}$.

To give a few intuitions, consider AND $\mathbf{tt} \mathbf{ff}$: then

$$\begin{aligned} (\lambda x.\lambda y.x y \mathbf{ff}) \mathbf{tt} \mathbf{ff} &\rightarrow_{\beta} (\lambda y.\mathbf{tt} y \mathbf{ff}) \mathbf{ff} \rightarrow_{\beta} \mathbf{tt} \mathbf{ff} \mathbf{ff} \\ &= (\lambda x.\lambda y.x) \mathbf{ff} \mathbf{ff} \rightarrow_{\beta} (\lambda y.\mathbf{ff}) \mathbf{ff} \rightarrow_{\beta} \mathbf{ff} \end{aligned}$$

and AND $\mathbf{tt} \mathbf{tt}$:

$$\begin{aligned} (\lambda x.\lambda y.x y \mathbf{ff}) \mathbf{tt} \mathbf{tt} &\rightarrow_{\beta} (\lambda y.\mathbf{tt} y \mathbf{ff}) \mathbf{tt} \rightarrow_{\beta} \mathbf{tt} \mathbf{tt} \mathbf{ff} \\ &= (\lambda x.\lambda y.x) \mathbf{tt} \mathbf{ff} \rightarrow_{\beta} (\lambda y.\mathbf{tt}) \mathbf{ff} \rightarrow_{\beta} \mathbf{tt} \end{aligned}$$

Other Boolean operations such as the disjunction, negation, etc. can also be defined in the λ -calculus.

3.1.2. Logic

Logic and Proof Theory is the study of mathematical proof and mathematical reasoning. A logical system is made of *formulas*, allowing one to express logical statements and of *inference rules*, allowing to reason, and prove, said statements. In a logical system, one is interested in *proof trees*. A proof tree is a tree where the leafs are formulas and the internal nodes are inference rules. The root of the tree is the formula we want to prove and the leafs are formulas which are always considered provable, called *axioms*. In this chapter, we present a fragment of the intuitionistic sequent calculus LJ, that we will also call LJ by abuse of notation. Formulas are builds upon connectives between atoms. Usual connectives feature the negation, conjunction, disjunction and implication, respectively noted $\neg, \wedge, \vee, \rightarrow$. Hence, given an infinite set of atoms $\{p, q, \dots\}$ the syntax of formulas is defined as:

$$A, B ::= p \mid \neg A \mid A \wedge B \mid A \vee B \mid A \rightarrow B$$

Then, a proof of some formula A , under a set of assumptions A_1, \dots, A_n is noted as $A_1, \dots, A_n \vdash A$. The left-hand side of the sequent (\vdash) is called the *context* and is defined as $\Delta ::= \emptyset \mid A, \Delta$ where the comma stands for the union. We only allow to have at most one formula on the right-hand-side of the sequent. A sequent $A_1, \dots, A_n \vdash B$ should be read as $A_1 \wedge \dots \wedge A_n \rightarrow B$, meaning that the formulas on the left side of the

$$\begin{array}{c}
 \frac{}{\Delta, A \vdash A} \text{ ax} \quad \frac{\Delta \vdash B \quad \Delta, B \vdash A}{\Delta \vdash A} \text{ cut} \\
 \\
 \frac{\Delta, A_i \vdash C}{\Delta, A_1 \wedge A_2 \vdash C} \wedge_L^i, i \in \{1, 2\} \quad \frac{\Delta \vdash A \quad \Delta \vdash B}{\Delta \vdash A \wedge B} \wedge_R \\
 \\
 \frac{\Delta, A \vdash C \quad \Delta, B \vdash C}{\Delta, A \vee B \vdash C} \vee_L \quad \frac{\Delta \vdash A}{\Delta \vdash A \vee B} \vee_R^1 \quad \frac{\Delta \vdash B}{\Delta \vdash A \vee B} \vee_R^2 \\
 \\
 \frac{\Delta \vdash A \quad \Delta, B \vdash C}{\Delta, A \rightarrow B \vdash C} \rightarrow_L \quad \frac{\Delta, A \vdash B}{\Delta \vdash A \rightarrow B} \rightarrow_R^1 \quad \frac{\Delta \vdash B}{\Delta \vdash A \rightarrow B} \rightarrow_R^2 \\
 \\
 \frac{\Delta \vdash A}{\Delta, \neg A \vdash} \neg_L \quad \frac{\Delta, A \vdash}{\Delta \vdash \neg A} \neg_R
 \end{array}$$

Figure 3.1.: Rules of LJ

sequent are put in conjunction and should imply the formula from the right hand side of the sequent.

The rules of the logic are given in Figure 3.1

Among the rules, the one of particular interest is the *cut* rule, it says that in order to prove some formula A , one can first start by proving B , and then use the hypothesis B in order to prove A . It corresponds to the use of a lemma in a mathematical proof. However, the use of the *cut* rule can lead to superfluous information. Consider a proof π of some formula A , one can then consider the proof:

$$\frac{\frac{\pi}{\vdash A} \quad \frac{}{A \vdash A} \text{ ax}}{\vdash A} \text{ cut}$$

The use of the cut and axiom rules are superfluous: they make a useless detour in order to prove A . This led to the notion of *proof simplification*, or *cut-elimination* by Gentzen [Gen35]: if there exists a proof π of some $\Delta \vdash A$, then there exists a proof π' of $\Delta \vdash A$ without cuts. This is an important result: it implies consistency, which says that one cannot prove the empty statement.

For instance, a rule of the cut-elimination is: $\frac{\frac{\pi}{\vdash A} \quad \overline{A \vdash A}}{\vdash A} \text{cut} \rightsquigarrow \frac{\pi}{\vdash A}$

3.1.3. Curry-Howard

$$\frac{\begin{array}{c} s \\ \vdots \\ A \vdash B \end{array} \quad \begin{array}{c} t \\ \vdots \\ \vdash A \end{array}}{\vdash B} \text{cut}$$

Figure 3.2.: Modus Ponens

Computation and logic are two faces of the same coin. For instance, consider a proof s of $A \rightarrow B$ and a proof t of A . With the logical rule *Modus Ponens* one can construct a proof of B : Figure 3.2 features a graphical presentation of the corresponding proof. In the *Curry-Howard correspondence* [Cur34; How80] types correspond to formulas and programs (terms) to proofs, while program evaluation is mirrored with proof simplification (the so-called cut-elimination).

The Curry-Howard correspondence formalizes the fact that the proof s of $A \rightarrow B$ can be regarded as a *function* —parametrized by an argument of type A — that produces a proof of B whenever it is fed with a proof of A . Therefore, the computational interpretation of Modus Ponens corresponds to the *application* of an argument (i.e. t) of type A to a function (i.e. s) of type $A \rightarrow B$. When computing the corresponding program, one substitutes the parameter of the function with t and get a result of type B . On the logical side, this corresponds to substituting every axiom introducing A in the proof s with the full proof t of A . This yields a direct proof of B without any invocation of the “lemma” $A \rightarrow B$.

Paving the way toward the verification of critical software, the Curry-Howard correspondence provides a versatile framework. It has been used to mirror first and second-order logics with dependent-type systems [BC13; Ler09], separation logics with memory-aware type systems [Rey02; Jun+17], resource-sensitive logics with differential privacy [Gab+13], logics with monads with reasoning on side effects [Swa+16; Mai+19], classical logic [Gri89], etc.

3.2. Linear Logic

Linear Logic, introduced by Girard [Gir87] is a resource sensitive logic in which formulas cannot be duplicated nor erased at will and which embed both classical and intuitionistic logic. Linear Logic was discovered by the study of the semantics of system F with coherent spaces, where the intuitionistic arrows $A \rightarrow B$ was decomposed into $!A \multimap B$.

In this decomposition, $!A$ (of course A) tells you that you can duplicate A as many times as you want and $A \multimap B$ is a function that uses its argument exactly once.

The syntax of the formulas of Linear Logic is given by:

$$\begin{array}{ll}
 A, B ::= \mathbf{1} \mid \perp \mid A \otimes B \mid A \wp B & \text{Multiplicative Fragment} \\
 \mathbf{0} \mid \top \mid A \oplus B \mid A \& B & \text{Additive Fragment} \\
 !A \mid ?A & \text{Exponential Fragment}
 \end{array}$$

Linear Logic admits several fragments of interests among which:

- Multiplicative Linear Logic (MLL): consists of the syntax whose connectors are \otimes, \wp and its units $\mathbf{1}$ and \perp .
- MALL (Multiplicative Linear Logic): is MLL with the additives connectives \oplus and $\&$ and their unit $\mathbf{0}, \top$.
- MELL (Multiplicative Exponentials Linear Logic): consist of MLL with the exponential connectives $!, ?$.

Finally, the formulas come with an involution operation for negation, noted A^\perp defined by:

$$\begin{array}{ll}
 \mathbf{1}^\perp = \perp & (A \otimes B)^\perp = A^\perp \wp B^\perp \\
 \mathbf{0}^\perp = \top & (A \oplus B)^\perp = A^\perp \& B^\perp
 \end{array}$$

$$\begin{array}{c}
 \frac{}{\vdash A^\perp, A} \textit{id} \qquad \frac{\vdash \Delta, A \quad \vdash \Gamma, A^\perp}{\vdash \Delta, \Gamma} \textit{cut} \\
 \\
 \frac{\vdash \Delta}{\vdash \perp, \Delta} \perp \qquad \frac{}{\vdash \mathbf{1}} \mathbf{1} \qquad \frac{\vdash A, B, \Delta}{\vdash A \wp B, \Delta} \wp \qquad \frac{\vdash A, \Delta \quad \vdash B, \Gamma}{\vdash A \otimes B, \Delta, \Gamma} \otimes \\
 \\
 \frac{\vdash A, \Delta \quad \vdash B, \Delta}{\vdash A \& B, \Delta} \& \qquad \frac{\vdash A_i, \Delta}{\vdash A_1 \oplus A_2, \Delta} \oplus^i \ i \in \{1, 2\} \qquad \frac{\Delta}{\vdash \Delta, \perp} \perp \\
 \\
 \frac{\vdash A, ?\Delta}{\vdash !A, ?\Delta} ! \qquad \frac{\vdash A, \Delta}{\vdash ?A, \Delta} ?d \qquad \frac{\vdash ?A, ?A, \Delta}{\vdash ?A, \Delta} ?c \qquad \frac{\vdash \Delta}{\vdash ?A, \Delta} ?w
 \end{array}$$

Figure 3.3.: Rules of Linear Logic.

Remark 3.2.1. Usually, Linear Logic is presented in a two-sided way where sequents are of the form $\Delta \vdash \Gamma$ and come with twice as many rules (for rules that are applied on the right-hand-side or on the left-hand-side of the sequent). This is closer to the formalism of LJ. One can go from the two-sided representation to the one-sided representation by applying the negation on the context Δ , i.e.: $\Delta \vdash \Gamma \rightsquigarrow \vdash \Gamma, \Delta^\perp$.

For example, in the two-sided version of Linear Logic, the rules for the $\&$ becomes:

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \& B} \&_R \quad \frac{\Gamma, A_i \vdash C}{\Gamma, A_1 \& A_2 \vdash C} \&_L^i, i \in \{1, 2\}$$

which matches the rule \wedge_R and \vee_L^i . This comparison can be done with the other additive connectives of Linear Logic. For the multiplicative connectives, there exists a multiplicative version of LJ that requires structural rules, that are matched by the exponential connectives. More information can be found in [Lau11].

Finally, Linear Logic comes with cut-elimination rules (also called proof simplification) given in Figure 3.4 where a double bar means that we apply the same rule multiple times. The cut-elimination also comes with some commutation rules which allows to commute some inference rules below a cut. We do not give all the rules, but for example we have:

$$\frac{\frac{\frac{\vdash C, A, B, \Delta}{\vdash C, A \wp B, \Delta} \wp}{\vdash A \wp B, \Delta, \Gamma} \text{cut} \quad \vdash C^\perp, \Gamma}{\vdash A \wp B, \Delta, \Gamma} \text{cut} \rightsquigarrow \frac{\frac{\vdash C, A, B, \Delta \quad \vdash C^\perp, \Gamma}{\vdash A, B, \Delta, \Gamma} \text{cut}}{\vdash A \wp B, \Delta, \Gamma} \wp$$

Linear Logic enjoys the cut-elimination theorem: given some proof $\frac{\pi}{\vdash \Delta}$ with cuts, there exists a proof $\frac{\pi'}{\vdash \Delta}$ without cuts.

From a Curry-Howard point of view, the Multiplicative fragment of Linear Logic (MLL) correspond to the linear λ -calculus while MELL correspond to the typed λ -calculus [Gir87]. One can then translate a typing derivation $x_1 : A_1, \dots, x_n : A_n \vdash t : B$ into the formula $?(A_1^*)^\perp, \dots, ?(A_n^*)^\perp, B^*$, where A^* is defined on the base type as the identity and $(A \rightarrow B)^* = !A^* \multimap B^*$. Following this perspective, linear logic is a good fit for the study of λ -calculus and their extensions with a well-studied semantics [Mel09].

$$\begin{array}{c}
 \frac{}{\vdash A, A^\perp} \text{ax} \\
 \frac{}{\vdash A, \Gamma} \text{cut} \rightsquigarrow \vdash A, \Gamma
 \end{array}$$

$$\frac{\frac{\vdash \Gamma, A \quad \vdash \Delta, B}{\vdash \Gamma, \Delta, A \otimes B} \otimes \quad \frac{\vdash \Xi, A^\perp, B^\perp}{\vdash \Xi, A^\perp \wp B^\perp} \wp}{\vdash \Gamma, \Delta, \Xi} \text{cut} \rightsquigarrow \frac{\frac{\vdash \Delta, B \quad \vdash \Xi, A^\perp, B^\perp}{\vdash \Delta, \Xi, A^\perp} \text{cut}}{\vdash \Gamma, \Delta, \Xi} \text{cut}$$

$$\frac{\frac{\vdash \Gamma, A_1 \quad \vdash \Gamma, A_2}{\vdash \Gamma, A_1 \& A_2} \& \quad \frac{\vdash \Delta, A_k^\perp}{\vdash \Delta, A_1^\perp \oplus A_2^\perp} \oplus_k}{\vdash \Gamma, \Delta} \text{cut} \rightsquigarrow \frac{\vdash \Gamma, A_k \quad \vdash \Delta, A_k^\perp}{\vdash \Gamma, \Delta} \text{cut for } k \in \{0, 1\}$$

$$\frac{\frac{\vdash ?\Gamma, A}{\vdash ?\Gamma, !A} ! \quad \frac{\vdash \Delta}{\vdash \Delta, ?A^\perp} \text{w}}{\vdash ?\Gamma, \Delta} \text{cut} \rightsquigarrow \frac{\vdash \Delta}{\vdash ?\Gamma, \Delta} \text{w}$$

$$\frac{\frac{\vdash ?\Gamma, A}{\vdash ?\Gamma, !A} ! \quad \frac{\vdash \Delta, A^\perp}{\vdash \Delta, ?A^\perp} \text{d}}{\vdash ?\Gamma, \Delta} \text{cut} \rightsquigarrow \frac{\vdash ?\Gamma, A \quad \vdash \Delta, A^\perp}{\vdash ?\Gamma, \Delta} \text{cut}$$

$$\frac{\frac{\vdash ?\Gamma, A}{\vdash ?\Gamma, !A} ! \quad \frac{\vdash \Delta, ?A^\perp, ?A^\perp}{\vdash \Delta, ?A^\perp} \text{c}}{\vdash ?\Gamma, \Delta} \text{cut} \rightsquigarrow \frac{\frac{\frac{\vdash ?\Gamma, A}{\vdash ?\Gamma, !A} ! \quad \vdash \Delta, ?A^\perp, ?A^\perp}{\vdash ?\Gamma, \Delta, ?A^\perp} \text{cut}}{\frac{\vdash ?\Gamma, ?\Gamma, \Delta}{\vdash ?\Gamma, \Delta} \text{c}} \text{cut}$$

$$\frac{\frac{\frac{\vdash ?\Gamma, A}{\vdash ?\Gamma, !A} ! \quad \frac{\vdash ?\Delta, ?A^\perp, B}{\vdash ?\Delta, ?A^\perp, !B} !}{\vdash ?\Gamma, ?\Delta, !B} \text{cut}}{\vdash ?\Gamma, ?\Delta, !B} \rightsquigarrow \frac{\frac{\frac{\vdash ?\Gamma, A}{\vdash ?\Gamma, !A} ! \quad \vdash ?\Delta, ?A^\perp, B}{\vdash ?\Gamma, ?\Delta, B} \text{cut}}{\vdash ?\Gamma, ?\Delta, !B} !$$

$$\frac{}{\vdash 1} \perp \quad \frac{\vdash \Gamma}{\vdash \Gamma, \perp} \perp}{\vdash \Gamma} \text{cut} \rightsquigarrow \vdash \Gamma$$

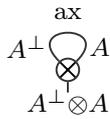
Figure 3.4.: Cut-elimination rules

3.2.1. Proof Nets

In Linear Logic, some rules are said to be *invertible*: they do not change the provability of the sequent. An instance of such a rule is the \wp : a \wp rule can always be applied before, or after, another rule. This cause of problem of identifying multiple proofs of the same formula that are equivalent up to some rule permutation. To solve this problem, Linear Logic comes with another syntax for proofs: a graphical language called *proof nets* [Gir87]. Proofs nets are defined in two steps: first, the general graphical language call *proof structures* and then a validity criterion that restraint the proof structure to only those that correspond to real proofs of linear logic. As with linear logic, there exists multiple kinds of proof nets, depending on which fragment of linear logic we are interested in. Most notably, the additive fragment as with the units have always created problems in defining a proper validity criterion that matches properly the cut-elimination procedure of their sequent-calculus counterpart. For the multiplicative fragment, the proof-structures are defined as:

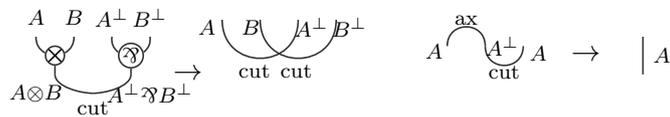
$$\left\{ \begin{array}{c} \text{ax} \\ A^\perp \curvearrowright A, \text{cut}, \begin{array}{c} A \quad B \quad A \quad B \\ \otimes \quad \wp \\ A \otimes B \quad A \wp B \end{array} \end{array} \right\}$$

The constructions of the proof-structures correspond respectively to the axiom, cut, tensor and par rule of MLL. Then, each derivation judgement from MLL can be sent into a proof structure. As discussed, not every proof structure is a proof net, for instance the following proof structure does not correspond to any proof of linear logic:



This is due to the fact that the tensor has two premisses and in this proof structure it only has one. Therefore, proof nets come with a validity criterion. We do not go into all the details as it is not necessary for this thesis, but intuitively the validity condition considers a switching function $\mathcal{S} : \wp \rightarrow \{l, r\}$ which consist in removing one of the two input wire of every \wp node in the proof structure. The obtained graph is called a *switching graph* and then, the validity criterion require that every switching graph that can be obtained from a proof structure is a *connected tree*.

Similarly, the cut-elimination procedure can also be defined inside proof nets:



Finally, one can show that cut-elimination holds inside proof nets and that any proof of linear logic gives a proof net, and that every proof net is the image of a proof [Gir87; Lau11]. More details can be found in [Lau11].

MELL Proof Nets MELL allows one to encode all of the simply typed λ -calculus, duplication of a λ -term is represented using exponentials, which also handle erasure. To make proof nets for MELL, one then needs a tool to represent what can, and cannot, be duplicated or erased. For that, the most common tool is *exponential boxes*. In MELL Proof Nets, proofs nets can be put inside boxes, which indicate that they can be duplicate or erased under certain conditions. The erasure is made through *weakening* and may create a situation where the proof net is disconnected. This creates problems in defining the validity condition. One solution would be to consider MELL + Mix, another way is through the use of *jumps*, connecting weakening to a correct proof net. Once again, more details can be found in [Lau11].

MALL Proof Nets On the additive side, correctness criterion becomes much harder. The main criterion of MALL Proof Nets [HVG03] works with additive resolutions: deleting one branch of each $\oplus, \&$ -node and through a notion of *toggling* of $\&$ -node. A toggle- $\&$ creates some kind of dependency between himself and other part of the proof-nets through jump-edges. One then needs to check a similar criterion to the MLL criterion: toggled $\&$ must not be present in a cycle of the proof-net with the added jump-edges.

3.2.2. Geometry of Interaction

Proof Nets, in particular MLL ones, comes with another form of semantics called the *Geometry of Interaction* (GoI) [Gir89b; Gir89a; Gir88; Gir95; Gir06; Gir11; Gir13]. The Geometry of Interaction can be seen in two ways:

- As a *wave-style semantics* where the proof net is seen as a global operator;
- as a *particle-style semantics* (also called *token-machine*) where *tokens* travel through the proof net, computing a global operator in a local manner.

Both view are related through the *Execution Formula*: a new, purely computational way to view the cut-elimination in an abstract way. In this thesis, we follow the particle-style approach of GoI, following the notion of GoI has a token-based automaton [DR99; AL95]. In this setting, the flow of tokens inside a proof net, seen as a graph, characterizes an invariant of the proof: its computational content. A token carry a *state*, which is being updated as the token goes up or down through nodes of the proof net. In this setting, applying the execution formula n times can be seen as moving the token n times through the proof net, and hence, by iterating the execution formula, the token always ends up leaving the proof net. Then, proof nets are seen as a global operation on the state space of tokens. For the case of MLL, this corresponds simply to a permutation. Due to the linearity of the logic and the fact that each operation is forward and backward deterministic (each step corresponding to simply going down, or up, on the right, or left, side of a node), each step of the token machine is reversible.

3.3. Linear Logic & Quantum Computation

The fact that linear logic doesn't allow for duplication or erasure of data is very reminiscent of the constraints of quantum computation. Both linear logic and quantum computation features a tensor, and the exponential can be used for typing classical data, which can be duplicated and erased at will. Multiple typed languages have been developed, along with their semantics in such a setting [PSV14; Gre+13; SV+09; Lee+21]. More related to proof nets, major works in quantum computation has taken inspiration from MLL Proof Nets, most notably the *Categorical Quantum Theory* [] and the *ZX-Calculus* [CD11] that grew out of it. However, the connection is not perfect as in quantum computation the tensor is self-dual and so the correctness condition of proof nets is no longer relevant, also there is no reason for the obtained graph to be acyclic as the trace is a fundamental structure. Finally, proof nets are oriented, which is no longer the case in the ZX-Calculus as all objects are self-dual. More details can be found in [AD06; Dun06; Dun09; Dun04]. Geometry of Interaction has also been applied in the context of quantum computation, both in a categorical setting [HH16] or in token-based-GoI setting such as in [Dal17; LZ15; DLF11], where the tokens are seen as qubits flowing inside a higher-order term, computing a quantum circuit. In this setting, tokens require a notion of synchronization: a token arriving at an input of a gate is blocked until all the inputs of the gates are populated by a token, at which point all the tokens go through at once (while obviously changing the state).

3.4. Infinitary Linear Logic : μ MALL

Functional programming languages often feature the ability to encode *inductive types* and *coinductive types* as data structure. On a Curry-Howard point of view, how inductive and coinductive types and reasoning are related to Linear Logic is not directly clear.

From a proof theory point of view, inductive and coinductive reasoning have been studied for a long time. Most notably by the μ -calculus [DR79; Par69; Koz83]: an extension of modal logic with fixed point operator. Modal logic was extended with a new formula of the form $\mu X.A$ where A is a formula, along with the dual operator $\nu X.A$. Already at this point, the μ and ν operator described the least and greatest fixed point operator. By the Curry-Howard correspondence, those new logics helped in modelling possibly infinite computation. Among those logics and derivations, *circular proofs* [San02; FS13], infinite proofs with finitely many subtrees, are in particular interest to represent recursive programs. As we are concerned with the particular case of Linear Logic, we look at the logic μ MALL. In his PhD, Baelde [Bae08] looked at adapting linear logic with formulas with fixed points and showed the cut-elimination and the equivalence of provability with regards to higher order linear logic. Finally, Amina et al [BDS16; Dou17; BDS16] looked more precisely at the infinitary aspects of the derivation with a proof-theoretical approach, defining validity criterions and their properties.

$$\frac{\frac{\vdots}{\vdash \nu X.X} \nu \quad \frac{\vdots}{\mu X.X \vdash \psi} \mu}{\vdash \psi} \text{cut}$$

Figure 3.5.: Degenerated proof

3.4.1. Background on μ MALL

The logic μ MALL [Bae12; BDS16] is an extension of the additive and multiplicative fragment of linear logic [Gir87]. The syntax of linear logic, where formulae are denoted by ψ, ϕ , is extended with the formulae $\mu X.\psi$ and its dual $\nu X.\psi$ (where X is a type variable occurring in ψ), which can be understood at the least and greatest fixed points of the operator $X \mapsto \psi$. These permits inductive and coinductive statements. One can for instance define the type of natural numbers as $\mu X.\mathbf{1} \oplus X$ or of lists of type ψ as $[\psi] = \mu X.\mathbf{1} \oplus (\psi \otimes X)$ or of streams of type ψ as $\nu X.\psi \otimes X$. Note that our system only deals with closed formulae. The syntax of formulae is $\psi, \phi ::= \alpha \mid \alpha^\perp \mid \mu X.\psi \mid \nu X.\psi \mid X \mid \perp \mid \mathbf{1} \mid \psi \wp \phi \mid \psi \otimes \phi \mid \mathbf{0} \mid \top \mid \psi \oplus \phi \mid \psi \& \phi$.

Where $\alpha \in \mathfrak{A}$, an infinite set of atoms and $X, Y, \dots \in \mathfrak{V}$ an infinite set of fixed point variables.

Definition 3.4.1 (Negation). *The negation of a formula, ψ^\perp is the involution on formulae satisfying $\alpha^{\perp\perp} = \alpha$, $(\psi \wp \phi)^\perp = (\psi^\perp \otimes \phi^\perp)$, $(\psi \oplus \phi)^\perp = \psi^\perp \& \phi^\perp$, $\perp^\perp = \mathbf{1}$, $\top^\perp = \mathbf{0}$, $(\nu X.\psi)^\perp = \mu X.\psi^\perp$, $X^\perp = X$. Having $X^\perp = X$ is harmless since we only deal with closed formulae.*

Following the notation of one-sided sequents, the rules for the μ and ν connectives are:

$$\frac{\vdash \psi[X \leftarrow \mu X.\psi], \Delta}{\vdash \mu X.\psi, \Delta} \mu \quad \frac{\vdash \psi[X \leftarrow \nu X.\psi], \Delta}{\vdash \nu X.\psi, \Delta} \nu$$

The logic allows for the constructions of infinite derivations, called *pre-proofs*. Their name comes from the fact that, if we consider that any derivation is a *proof*, then we can prove any statement ψ using the cut-rule, as shown in Figure 3.5. This is why μ MALL comes with a validity criterion, separating *pre-proofs* from actual *proofs*. μ MALL also comes with a boolean semantics [BDS16; Dou17] on formulae, noted $\llbracket - \rrbracket$, giving us information on whenever or not a formula can be deemed provable. They proved that if a sequent $\vdash \Gamma$ is provable then $\llbracket F \rrbracket = \mathfrak{tt}$ for some formula $F \in \Gamma$.

In order to distinguish between pre-proofs and actual proofs, μ MALL comes with *validity criterion* on derivations: mainly, whether or not each infinite branch can be justified by

a form of coinductive reasoning. The criterion also ensures that the cut-elimination procedure holds. The criterion is based on Girard's Geometry of Interaction where some data (namely a *thread*) move through the derivation, following a subformula and collecting information (its *weight*). Then, one can analyse the collected information and determine whenever or not it is *valid*. More formally, a *thread* [BDS16; Bae+20] is an infinite sequence of tuples of formulas, sequents and directions (either up or down) written $(F; \vdash \Delta; d)$. Intuitively, these threads follow some formula starting from the root of the derivation and start by going up. The thread has the possibility to *bounce* on axioms and cuts and change its direction, either going back-down on an axioms or back-up on a cut. A thread will be called *valid* when it is non-stationary (does not follow a formula that is never a principal formula of a rule), and when in the set of formulas appearing infinitely often, the minimum formula (according to the subformula ordering) is a ν formula. For the multiplicative fragment, we say that a pre-proof is valid if for all infinite branches, there exists a valid thread, while for the additive part, we require a notion of *additive slices* and *persistent slices*, and we ask that all persistent slices are valid in the sence of the multiplicative fragment.

Multiple validity criterion exists, some covering more derivation than others. One thing that has to be noticed is that, even though a valid proof is productive in the sense of the cut-elimination, not all productive derivation are valid proofs. One on hand [NST18] introduced a local condition for circular proofs validity. In their system $\mu\text{MALL}^{\curvearrowright}$, every valid proof is a valid proof in μMALL , while the conserve is not true: interleaving of fixed-point and back-edge are not captured by their system. The system of thread previously mentioned is developed in [BDS16] and extended in [Bae+20] and called the *bouncing-thread-validity*. Compare to [NST18] the criterion is a global one that looks at infinite threads. This is the criterion that we present here and that we will use in Chapter 4.

3.4.2. Bouncing Validity

We consider sequents to work with sets of named formulas, also called *formula occurrences*. The idea is that each formula is attached a unique address, when a rule is applied to said formula, its subformulas will extend their addresses, corresponding to their provenance by $\{l, r, i\}$ (for left, right, inside).

Definition 3.4.2 (Addresses). *Let \mathfrak{A}_{fresh} be an infinite set of atomic addresses and $\mathfrak{A}_{fresh}^{\perp} = \{\alpha^{\perp} \mid \alpha \in \mathfrak{A}_{fresh}\}$ and $\Sigma = \{l, r, i\}$. An address is a word of the form $\alpha \cdot x$ where $\alpha \in \mathfrak{A}_{fresh} \cup \mathfrak{A}_{fresh}^{\perp}$ and $x \in \Sigma^*$. Given two addresses α', α we say that α' is a sub-address of α when α is a prefix of α' , noted $\alpha \sqsubseteq \alpha'$. Two addresses are disjoint if they are incompatible with regard to \sqsubseteq .*

We can now define the notion of *formula occurrence* F, G, \dots as a simple pair (ψ, α) where ψ is a formula and α is an address, noted $F = \psi_{\alpha}$. We say that two formula

$$\begin{array}{c}
 \frac{F \equiv G}{\vdash F^\perp G} \textit{id} \qquad \frac{\vdash \Delta, F \quad \vdash \Gamma, F^\perp}{\vdash \Delta, \Gamma} \textit{cut} \qquad \frac{\vdash \Delta}{\vdash \perp, \Delta} \perp \\
 \\
 \frac{}{\vdash \mathbb{1}} \mathbb{1} \qquad \frac{\vdash F, G, \Delta}{\vdash F \wp G, \Delta} \wp \qquad \frac{\vdash F, \Delta \quad \vdash G, \Gamma}{\vdash F \otimes G, \Delta, \Gamma} \otimes \\
 \\
 \frac{\vdash F, \Delta \quad \vdash G, \Delta}{\vdash F \& G, \Delta} \& \qquad \frac{\vdash F_i, \Delta}{\vdash F_1 \oplus F_2, \Delta} \oplus_R^i \ i \in \{1, 2\} \qquad \frac{\Delta}{\vdash \Delta, \perp} \perp \\
 \\
 \frac{\vdash F[X \leftarrow \mu X.F], \Delta}{\vdash \mu X.F, \Delta} \mu \qquad \frac{\vdash F[X \leftarrow \nu X.F], \Delta}{\vdash \nu X.F, \Delta} \nu
 \end{array}$$

 Figure 3.6.: Rules for μ MALL.

occurrences are structurally equivalent, noted $\psi_\alpha \equiv \phi_\beta$ when the underlying formulas are the same: $\psi = \phi$.

As we will work with formula occurrences, logical connectives need to be lifted on occurrences:

Definition 3.4.3 (Logical Connectives with occurrences).

- For any $\# \in \{\otimes, \oplus, \wp, \&\}$ if $F = \psi_{\alpha l}$ and $G = \phi_{\alpha r}$ then $F\#G = (\psi\#\phi)_\alpha$.
- For any $\# \in \{\mu, \nu\}$ if $F = \psi_{\alpha i}$ then $\#X.F = (\#X.\psi)_\alpha$.

The derivation rules are shown in Figure 3.6. They define a relation $\vdash \Delta$ on a set of formula occurrences defined co-inductively. For each rule the assumptions are above the line while the conclusion is under. In the rules, the comma stands for the disjoint union. Given a rule, we call the formula to which the rule is applied to the *active formula* or *principal formula*. Given a sequent s in a pre-proof π , we denote by $\text{premiss}(s)$ the set of premisses of the rule of conclusion s in π . The cut-elimination rules for μ MALL are the same as the one presented in Figure 3.4 with the additional rule

$$\frac{\frac{\vdash F^\perp[X \leftarrow \nu X.F^\perp], \Gamma}{\vdash \mu X.F^\perp, \Gamma} \mu \quad \frac{\vdash F[X \leftarrow \nu X.F], \Delta}{\vdash \nu X.F, \Delta} \nu}{\vdash \Delta, \Gamma} \textit{cut} \quad \rightsquigarrow \quad \frac{\vdash F^\perp[X \leftarrow \nu X.F^\perp], \Gamma \quad \vdash F[X \leftarrow \nu X.F], \Delta}{\vdash \Delta, \Gamma} \textit{cut}$$

Derivations can be potentially non-well-founded trees: they are not necessarily finite as we can for instance consider the formula $\mu X.X$ and apply the rule μ an infinite number of times.

Example 3.4.4. Taking the natural number $\mathbb{N} = \mu X. \mathbf{1} \oplus X$, one can define the proof of 0 and of n as:

$$\pi_0 = \frac{\frac{\overline{\vdash \mathbf{1}} \quad \mathbf{1}}{\vdash \mathbf{1} \oplus \mathbb{N}} \oplus^1}{\vdash \mu X. \mathbf{1} \oplus X} \mu \qquad \pi_n = \frac{\frac{\pi_{n-1}}{\vdash \mathbb{N}}}{\vdash \mathbf{1} \oplus \mathbb{N}} \oplus^2}{\vdash \mu X. \mathbf{1} \oplus X} \mu$$

As mentioned earlier, not all derivation are indeed proofs. To answer this problem, μ MALL comes with a validity criterion for derivations. This makes uses of the notion of *bouncing-threads*: paths that travel along the infinite derivation and collect some information along the way. In order to formally define bouncing-threads we first need to introduce some notations: given an alphabet Σ , we denote by Σ^∞ the set of infinite words over Σ and $\Sigma^\omega = \Sigma^* \cup \Sigma^\infty$. The letter ϱ will denote ordinals in $\omega + 1$. Finally, we use a special concatenation: given $u = (u_i)_{i \leq n \leq \omega}$ and $v = (v_i)_{i \in \varrho}$ such that $u_n = v_0$, we define $u \odot v$ as the concatenation of u and v without the first element of v , i.e. : $u \cdot (v_i)_{i \in \varrho \setminus \{0\}}$. For instance $aab \odot bac = aabac$.

We begin with the definition of a *pre-threads* : the basic construction of a path along an infinite tree that follows a formula occurrence. Then, we only look at some specials ones called *threads*, and finally define the notion of *valid thread* that validates an infinite derivation as being a proof. A pre-proof is a proof whenever all of its infinite branches are valid.

Definition 3.4.5 (Pre-thread). *A pre-thread is a sequence $(F_i, s_i, d_i)_{i \in \varrho}$ of tuples of a formula, a sequent and a direction $d \in \{\uparrow, \downarrow\}$ such that for all $i \in \varrho$ and $i + 1 \in \varrho$ one of the following holds:*

- $d_i = d_{i+1} = \uparrow$, $s_{i+1} \in \text{premiss}(s_i)$ and $F_{i+1} \sqsubseteq F_i$
- $d_i = d_{i+1} = \downarrow$, $s_i \in \text{premiss}(s_{i+1})$ and $F_i \sqsubseteq F_{i+1}$
- $d_i = \downarrow, d_{i+1} = \uparrow$, s_i and s_{i+1} are the two premisses of the same cut rule and $F_i = F_{i+1}^\perp$
- $d_i = \uparrow, d_{i+1} = \downarrow$ and $s_i = s_{i+1} = \{\vdash F_i, F_{i+1}\}$ is the conclusion of an axiom rule.

As mentioned before, a pre-thread follows a subformula inside a derivation, going up or down and bouncing back on axioms rules and cut rules. We can then define a notion of *weight* on a pre-thread: a potentially infinite word over $\{l, r, i, \bar{l}, \bar{r}, \bar{i}, \mathcal{W}, \mathcal{A}, \mathcal{C}\}$ where l, r, i stands for being on the left, right or inside a subformula while the pre-thread is going up, $\bar{l}, \bar{r}, \bar{i}$ is similar but while the pre-thread is going down and \mathcal{W}, \mathcal{A} and \mathcal{C} stands respectively for *wait*, *axiom* and *cut*: if a pre-thread is going up on a formula that is not the principal formula, then the weight becomes \mathcal{W} , and when the pre-thread bounces back on an axiom (resp. a cut), its weight becomes \mathcal{A} (resp. \mathcal{C}).

Definition 3.4.6 (Weight of a pre-thread). *Let t be a pre-thread, the weight of t , noted $w(t)$ is a word over $(w_i)_{i \in \varrho} \{l, r, i, \bar{l}, \bar{r}, \bar{i}, \mathcal{W}, \mathcal{A}, \mathcal{C}\}^\infty$ such that for every $i \in \varrho$ one of the following holds:*

- $w_i = x$ if $F_i = \psi_\alpha$ and $F_{i+1} = \phi_{\alpha x}$ for $x \in \{l, r, i\}$
- $w_i = \bar{x}$ if $F_i = \psi_{\alpha x}$ and $F_{i+1} = \phi_\alpha$ for $x \in \{l, r, i\}$
- $w_i = \mathcal{A}$ if $d_i = \uparrow$ and $d_{i+1} = \downarrow$ (corresponding to an axiom rule)
- $w_i = \mathcal{C}$ if $d_i = \downarrow$ and $d_{i+1} = \uparrow$ (corresponding to a cut-rule).
- $w_i = \mathcal{W}$ if $F_i = F_{i+1}$ (corresponding to the fact that a rule is not applied to the formula followed by the pre-thread).

As mentioned earlier we are not interested in every single pre-thread, but only those that follow a certain pattern (one that matches with the cut-elimination procedure).

We define two set of words \mathfrak{B} and \mathfrak{H} inductively as follows:

$$\mathfrak{B} ::= \mathcal{C} \mid \mathfrak{B}\mathcal{W}^*\mathcal{A}\mathcal{W}^*\mathfrak{B} \mid \bar{x}\mathcal{W}^*\mathfrak{B}\mathcal{W}^*x$$

$$\mathfrak{H} ::= \epsilon \mid \mathcal{A}\mathcal{W}^*\mathfrak{B}$$

A finite pre-thread t is called a *b-path* if $w(t) \in \mathfrak{B}$, and it is called a *h-path* if $w(t) \in \mathfrak{H}$.

We are now ready to define the notion of thread:

Definition 3.4.7 (Thread). *A pre-thread t is a thread when it can be decomposed as*

$$\odot_{i \in \varrho+1} (H_i \odot V_i) \text{ where for all } i \in \varrho+1:$$

- $w(V_i) \in \{l, r, i, \mathcal{W}\}^\infty$, and it is non-empty if $i \neq \varrho$
- $w(H_i) \in \mathfrak{H}$, and it is non-empty if $i \neq 0$

The decomposition can be read as a thread initially going up, accumulating some debt in the form of the alphabet $\{l, r, i\}$ that will need to be repaid by their opposite $\{\bar{l}, \bar{r}, \bar{i}\}$ when going down after meeting an axiom until it reaches a cut. Those dual alphabets correspond to steps of the cut-elimination: making sure that the correct formulas will at some point interact by a cut.

Such a decomposition is unique, and we call $(V_i)_{i \in \varrho+1}$ the *visible part* of t and $(H_i)_{i \in \varrho+1}$ the *hidden part*. A thread is *stationary* when its visible part is a finite sequence (of finite words) or when there exists $k \in \varrho+1$ such that $w(V_i) \in \{\mathcal{W}\}^\infty$ for all $k \leq i \in \varrho+1$.

We can now define the validity criterion on threads:

Definition 3.4.8 (Valid threads). *If we take the sequence of formulas followed by a non-stationary thread on its visible part and skipping the steps corresponding to \mathcal{W} weights, we obtain an infinite sequence of formulas where each formula is an immediate subformula or an unfolding of the previous formula. The formulas appearing infinitely often admit a minimum with regard to the subformula ordering, such a formula is the minimal formula of the thread.*

A non-stationary thread is valid if its minimal formula is a ν formula.

We then need to import this notion of valid threads to pre-proofs. For this we split it into two cases: the first one being for μMLL : the multiplicative fragment of μMALL , and then for the whole of μMALL .

Definition 3.4.9 (Validity of μMLL). *A pre-proof of μMLL is valid if every infinite branch is valid.*

We say than an infinite branch β is valid if there exists a valid thread starting from one of its sequents, whose visible part is contained in this branch.

Example 3.4.10. *The infinite derivation $\frac{\vdots}{\frac{\vdots}{\vdash \mu X.X} \mu} \mu$ is not valid as the only thread $t = (\mu X.X; \vdash \mu X.X; \uparrow)^\infty$ have for weight i^∞ which is stationary.*

This notion is not sufficient for the whole of μMALL mainly due to the duplication of proofs that arise from the cut-elimination of the $\&$ connective (see Figure 3.4). We need to take into accounts *slices*, we consider two new rules :

$$\frac{\vdash \Delta, F}{\vdash \Delta, F \& G} \&_1 \quad \frac{\vdash \Delta, G}{\vdash \Delta, F \& G} \&_2$$

Now, given a pre-proof π of μMALL we consider the set of its slices $Sl(\pi)$, defined corecursively by:

$$Sl \left(\frac{\frac{\pi_1}{\vdash \Delta, F_1} \quad \frac{\pi_2}{\vdash \Delta, F_2}}{\vdash \Delta, F_1 \& F_2} \& \right) \triangleq \left\{ \frac{\frac{\pi'_i}{\vdash \Delta, F_i}}{\vdash \Delta, F_1 \& F_2} \&_i \mid i \in \{1, 2\}, \pi'_i \in Sl(\pi_i) \right\}$$

Equation Section 3.4.2 means that for each rule $\&$ in π , we consider the two proofs where the rule $\&$ is replaced with $\&_1$ and $\&_2$, similarly to the switching of proof nets.

Definition 3.4.11 (Persisting slice). *Given some slice, a rule $\&_i$ of principal formula $F_1 \& F_2$ is well-sliced if no b -path starting from $F_1 \& F_2$ ends in a formula $F_1^\perp \oplus F_2^\perp$ that is the principal formula of a \oplus_j rule with $i \neq j$. A slice is persistent if all its $\&_i$ are well-sliced.*

Remark 3.4.12. *An additional rule is needed to account for what happens when a π_i and a \oplus interact the cut-elimination. But as we just want to define the validity criterion, we do not detail the additional rule. The interested reader can always refer to [Bae+20].*

Definition 3.4.13 (Persistent slice validity). *A persistent slice is valid if it is valid in the sense of μ MALL.*

Definition 3.4.14 (μ MALL validity). *A pre-proof of μ MALL is valid if all its persistent slices are valid.*

Example 3.4.15. *Remember that $\mathbb{N} = \mu X. \mathbb{1} \oplus X$. We can then define the successor function as:*

$$\pi_{\text{succ}} = \frac{\frac{\frac{\overline{\mathbb{1}}}{\vdash \mathbb{1}} \oplus^1}{\vdash \mathbb{1} \oplus \mathbb{N}} \mu}{\frac{\frac{\vdash \mathbb{N}}{\mathbb{1} \vdash \mathbb{N}} \perp}{\mathbb{1} \oplus \mathbb{N} \vdash \mathbb{N}} \&}} \frac{\frac{\frac{\frac{\pi_{\text{succ}}}{\mathbb{N} \vdash \mathbb{N}} \oplus^2}{\mathbb{N} \vdash \mathbb{1} \oplus \mathbb{N}} \mu}{\mathbb{N} \vdash \mathbb{N}} \&}} \mu$$

In this derivation, all the slices from $\&_1$ are finite derivation. The only infinite derivation becomes the one where all occurrences of $\&$ becomes $\&_2$. In this infinite branch we have two threads: the one on the left and the one on the right of the sequent.

The left-thread t_l have for weight $(irWW)^\infty$ and the set of its formulas encountered infinitely often are $\{\nu X. \perp \& X, \perp \& \nu X. \perp \& X\}$. The smallest formula is a ν formula, validating the branch.

On the other hand, the right-thread has weight $(WWir)^\infty$ and for smallest formula a μ formula and hence is not valid. Since we only require one valid thread per infinite branch, the whole derivation is proof.

3.4.3. Circular Representation of Derivations

Among the infinite derivations that μ MALL offer we can look at the *circular* ones: an infinite derivation is circular if it has finitely many different subtrees. The circular derivation can therefore be represented in a more compact way with the help of *back-edges*: arrows in the derivation that represent a repetition of the derivation. Derivations with back-edges are represented with the addition of sequents marked with a back-edge label, noted \vdash^f , and an additional rule, $\overline{\vdash \Sigma}^{be(f)}$, which represents a back-edge pointing to the sequent \vdash^f . We take the convention that from the root of the derivation from to rule $be(f)$ there must be exactly one sequent annotated by f .

While a circular proof has multiple finite representations (depending on where the back-edge is placed), they can all be mapped back to the same infinite derivation via an infinite unfolding of the back-edge and forgetting the back-edge labels:

Definition 3.4.16 (Unfolding). *We define the unfolding of a circular derivation P with a valuation v from back-edge labels to derivations by:*

- $\mathcal{U} \left(P : \frac{P_1, \dots, P_n}{\vdash \Sigma} r, v \right) = \frac{\mathcal{U}(P_1, v), \dots, \mathcal{U}(P_n, v)}{\vdash \Sigma} r$
- $\mathcal{U}(be(f), v) = v(f)$
- $\mathcal{U} \left(P : \frac{P_1, \dots, P_n}{\vdash^f \Sigma} r, v \right) = \left(\pi = \frac{\mathcal{U}(P_1, v'), \dots, \mathcal{U}(P_n, v')}{\vdash \Sigma} r \right)$ with $v'(g) = \pi$ if $g = f$, else $v(g)$.

Example 3.4.17. *On the left, an infinite derivation and on the right two circular representations of the left-most derivation. Notice that the unfolding of the second and third derivation are equal to the first derivation.*

$$\begin{array}{ccc}
 \begin{array}{c} \vdots \\ \frac{}{\vdash \mu X.X} \mu \\ \frac{}{\vdash \mu X.X} \mu \end{array} & \begin{array}{c} \frac{}{\vdash \mu X.X} be(f) \\ \frac{}{\vdash \mu X.X} \mu \\ \frac{}{\vdash^f \mu X.X} \mu \end{array} & \begin{array}{c} \frac{}{\vdash \mu X.X} be(f) \\ \frac{}{\vdash \mu X.X} \mu \\ \frac{}{\vdash^f \mu X.X} \mu \end{array}
 \end{array}$$

Part II.

Contributions

Chapter 4.

A Curry-Howard Correspondence for Linear, Reversible Computation

Abstract

In this chapter, we present a linear and reversible programming language with inductive types and recursion. The semantics of the language is based on pattern-matching; we show how ensuring syntactical exhaustivity and non-overlapping of clauses is enough to ensure reversibility. The language allows to represent any Primitive Recursive Function. We then give a Curry-Howard correspondence with the logic μ MALL, that is linear logic extended with least fixed points allowing inductive statements. The critical part of our work is to show how primitive recursion yields circular proofs that satisfy μ MALL validity criterion and how the language simulates the cut-elimination procedure of μ MALL.

References: Results of this chapter have been accepted in the paper *Towards a Curry-Howard Correspondence for Linear, Reversible Computation* at CSL'23 and accepted as a Work in Progress at RC'20 [CSV20].

4.1. Introduction

Reversible computation is a paradigm of computation which emerged as an energy-preserving model of computation in which data is never erased [FT82] that makes sure that, given some process f , there always exists an inverse process f^{-1} such that $f \circ f^{-1} = \text{Id} = f^{-1} \circ f$. Many aspects of reversible computation have been considered, such as the development of reversible Turing Machines [AG11a; MY07], reversible programming languages [JS14] and their semantics [CLV21; KAG17; KR21]. However, the formal relationship between a logical system and a computational model has not been developed yet.

While multiple reversible functional programming languages have been developed [YAG12; TA15; JS14; SVV18; JKT18] featuring type systems, case-analysis and pattern-matching, they most often lack a formal connection with logical systems. This chapter aims at proposing a type system featuring inductive types for a purely linear and reversible

language, together with a Curry-Howard correspondence with the logic μMALL . As reversible and linear computation make for a subset of quantum computation, this work is a first step towards understanding purely quantum recursive types. We base our study on the approach presented in [SVV18]. In this model, reversible computation is restricted to two main types: the tensor, written $A \otimes B$, and its neutral element $\mathbb{1}$, and the co-product, written $A \oplus B$. The former corresponds to the type of all pairs of elements of type A and elements of type B , while the latter represents the disjoint union of all elements of type A and elements of type B . For instance, a bit can be typed with $\mathbb{1} \oplus \mathbb{1}$, where $\mathbb{1}$ is a type with only one element. The language in [SVV18] offers the possibility to code isos —reversible maps— with pattern-matching. An iso is for instance the swap operation, typed with $A \otimes B \leftrightarrow B \otimes A$. However, if [SVV18] hints at an extension towards pure quantum computation, the type system is not formally connected to any logical system.

The problem of reversibility between finite types of same cardinality simply requires to check that the function is injective. That is no longer the case when we work with types of infinite cardinality such as natural numbers.

The main contribution of this work is a Curry-Howard correspondence for a purely reversible typed language in the style of [SVV18], with added generalized inductive types and terminating recursion, enforced by the fact that recursive functions must be structurally recursive: each recursive call must be applied to a decreasing argument. We show how requiring exhaustivity and non-overlapping of the clauses of the pattern-matching is enough to ensure reversibility and that the obtained language can encode any Primitive Recursive function [RJ87]. For the Curry-Howard part, we capitalize on the logic μMALL [BDS16; Bae12]: an extension of the additive and multiplicative fragment of linear logic with least and greatest fixed points allowing inductive and coinductive statements. This logic contains both a tensor and a co-product, and its strict linearity makes it a good fit for a reversible type system. In the literature, multiple proofs systems have been considered for μMALL , some finitary proof system with explicit induction inferences à la Park [Bae12] as well as non-well-founded proof systems which allow to build infinite derivations [BDS16; Bae+20]. In the former, the inference rule features an invariant of the operator. While in the latter, no invariant is explicitly featured in the proof system and one can consider infinite derivation, but then the obtained logic is inconsistent and require a validity criterion. The present chapter focuses on the latter. In general, an infinite derivation is called a *pre-proof* and is not necessarily consistent. To solve this problem μMALL comes equipped with a *validity criterion*, telling us when an infinite derivation can be considered as a logical proof. We show how the syntactical constraints of being structurally recursive imply the validation of pre-proofs. In [CLV21], together with Louis Lemonnier and Benoît Valiron we also provided a categorical semantics of a simplified version of the language presented in this chapter. As it was the main work of Louis Lemonnier, we do not present it here, but the interested reader can refer to [CLV21].

Organization of the chapter The chapter is organized as follows: in Section 4.2 we present the language, its syntax, typing rules and semantics and show that any function that can be encoded in our language represents an isomorphism. In Section 4.3 we show that our language can encode any Primitive Recursive Function [RJ87], this is shown by encoding the set of Recursive Primitive Permutations [PPR20] functions. Then in Section 4.4, we develop on the Curry-Howard Correspondence part: we show, given a well-typed term from our language, how to translate it into a circular derivation of the logic μ MALL and show that the given derivation respects the validity condition and how our evaluation strategy simulates the cut-elimination procedure of the logic. Finally, in Section 4.5 we look at the expressiveness of the language if we remove the exhaustivity and termination condition and show that we obtain Turing Completeness.

4.2. First-order Isos

Our language is based on the one introduced by Sabry et al [SVV18] which defines isomorphisms between various types, including the type of lists. We build on the reversible part of the chapter by extending the language to support both a more general rewriting system and more general inductive types. The language is defined by layers. Terms and types are presented in Table 4.1, while typing derivations, inspired from μ MALL, can be found in Tables 4.2 and 4.3. The language consists of the following pieces.

Basic type. They allow us to construct first-order terms. The constructors inj_l and inj_r represent the choice between either the left or right-hand side of a type of the form $A \oplus B$; the constructor \langle, \rangle builds pairs of elements (with the corresponding type constructor \otimes); fold represents inductive structure of the types $\mu X.A$. A value can serve both as a result and as a pattern in the defining clause of an iso. We write x_1, \dots, x_n for $\langle x_1, \dots, x_n \rangle$ or \vec{x} when n is non-ambiguous and $A_1 \otimes \dots \otimes A_n$ for $A_1 \otimes (\dots \otimes A_n)$ and A^n for $\underbrace{A \otimes \dots \otimes A}_{n \text{ times}}$.

First-order isos. An iso of type $A \leftrightarrow B$ acts on terms of base types. An iso is a function of type $A \leftrightarrow B$, defined as a set of clauses of the form $\{v_1 \leftrightarrow e_1 \mid \dots \mid v_n \leftrightarrow e_n\}$. In the clauses, the tokens v_i are open values and e_i are expressions. In order to apply an iso to a term, the iso must be of type $A \leftrightarrow B$ and the term of type A . In the typing rules of isos, the $\text{OD}_A(\{v_1, \dots, v_n\})$ predicate (adapted from [SVV18] and given in Table 4.4) stands for Orthogonal Decomposition and syntactically enforces the exhaustivity and non-overlapping conditions on a set of well-typed values v_1, \dots, v_n of type A . *Exhaustivity* for an iso $\{v_1 \leftrightarrow e_1 \mid \dots \mid v_n \leftrightarrow e_n\}$ of type $A \leftrightarrow B$ means that the expressions on the left (resp. on the right) of the clauses describe all possible values for the type A (resp. the type B). *Non-overlapping* means that two expressions cannot match the same value. For instance, the left and right injections $\text{inj}_l v$ and $\text{inj}_r v'$ are non-overlapping while a variable x is always exhaustive. The typing conditions make sure that both the left-hand-side and right-hand-side of clauses satisfy this condition.

Its formal definition can be found in Table 4.3 where $Val(e)$ is defined as $Val(\mathbf{let} p = \omega p' \mathbf{in} e) = Val(e)$, and $Val(v) = v$ otherwise. These checks are crucial to make sure that our isos are indeed reversible. One can note that the definition is sound, but not complete: not every set of exhaustive and non-overlapping values satisfied the OD predicate, but it captures enough for the expressivity results of Theorem 4.3.7. In the rule $OD_{A \otimes B}$, we define S_v^1 and S_v^2 respectively as $\{w \mid \langle v, w \rangle \in S\}$ and $\{w \mid \langle w, v \rangle \in S\}$ and $\pi_1(S)$ and $\pi_2(S)$ respectively as $\{v \mid \langle v, w \rangle \in S\}$ and $\{w \mid \langle v, w \rangle \in S\}$. The construction $\mathbf{fix} g.\omega$ represents the creation of a recursive function, rewritten as $\omega[g \leftarrow \mathbf{fix} g.\omega]$ by the operational semantics. Each recursive function needs to satisfy a structural recursion criterion, formalized in Definition 4.2.2. It ensures that one of the input arguments strictly decreases on each recursive call. Indeed, since isos can be non-terminating (due to recursion), we need a criterion that implies termination to ensure that we work with total functions. If ω is of type $A \leftrightarrow B$, we can build its inverse $\omega^\perp : B \leftrightarrow A$ and show that their composition is the identity. We recall that 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, even when this remains implicit.

Convention 4.2.1. *We assume that the type constructors are right-associative, hence $\mathbf{fold} \mathbf{inj}_l x$ is $\mathbf{fold} (\mathbf{inj}_l (x))$.*

The type system has two types of judgments: one for terms (noted $\Delta; \Psi \vdash_e t : A$) and one for isos (noted $\Psi \vdash_\omega \omega : A \leftrightarrow B$). In the typing rules, the contexts Δ are sets of pairs that consist of a term-variable and a base type, where each variable can only occur once and Ψ is a singleton set of a pair of an iso-variable and an iso-type association.

Definition 4.2.2 (Structurally Recursive). *Given an iso $\mathbf{fix} f.\{v_1 \leftrightarrow e_1 \mid \dots \mid v_n \leftrightarrow e_n\}$ of type $A_1 \otimes \dots \otimes A_m \leftrightarrow C$, it is structurally recursive if there is $1 \leq j \leq m$ such that $A_j = \mu X.B$ and for all $i \in \{1, \dots, n\}$ we have that v_i is of the form (v_i^1, \dots, v_i^m) such that v_i^j is either:*

- *A closed value, in which case e_i does not contain the subterm $f p$;*
- *Open, in which case for all subterms of the form $f p$ in e_i we have $p = (x_1, \dots, x_m)$ and $x_j : \mu X.B$ is a strict subterm of v_i^j .*

Given a clause $v \leftrightarrow e$, we call the value v_i^j (resp. the variable x_j) the decreasing argument (resp. the focus) of the structurally recursive criterion.

Example 4.2.3. *The iso $\mathbf{fix} f.\{x \leftrightarrow \mathbf{let} y = f x \mathbf{in} y\}$ is not structurally recursive while the one from Example 4.2.14 is.*

Remark 4.2.4. *As we are focused on a very basic notion of structurally recursive function, the typing rules of isos allow to have at most one iso-variable in the context, meaning that we cannot have intertwined recursive calls.*

(Base types)	$A, B ::= \mathbf{1} \mid A \oplus B \mid A \otimes B \mid \mu X.A$
(Isos, first-order)	$\alpha ::= A \leftrightarrow B$
(Values)	$v ::= () \mid x \mid \mathbf{inj}_l v \mid \mathbf{inj}_r v \mid \langle v_1, v_2 \rangle \mid \mathbf{fold} v$
(Pattern)	$p ::= x \mid \langle p_1, p_2 \rangle$
(Expressions)	$e ::= v \mid \mathbf{let} p_1 = \omega p_2 \mathbf{in} e$
(Isos)	$\omega ::= \{v_1 \leftrightarrow e_1 \mid \dots \mid v_n \leftrightarrow e_n\} \mid \mathbf{fix} f.\omega \mid f$
(Terms)	$t ::= () \mid x \mid \mathbf{inj}_l t \mid \mathbf{inj}_r t \mid \langle t_1, t_2 \rangle \mid$ $\mathbf{fold} t \mid \omega t \mid \mathbf{let} p = t_1 \mathbf{in} t_2$

Table 4.1.: Terms and types

$\emptyset; \Psi \vdash_e () : \mathbf{1}$	$x : A; \Psi \vdash_e x : A$	$\frac{\Delta; \Psi \vdash_e t : A}{\Delta; \Psi \vdash_e \mathbf{inj}_l t : A \oplus B}$	$\frac{\Delta; \Psi \vdash_e t : B}{\Delta; \Psi \vdash_e \mathbf{inj}_r t : A \oplus B}$
$\frac{\Delta_1; \Psi \vdash_e t_1 : A \quad \Delta_2; \Psi \vdash_e t_2 : B}{\Delta_1, \Delta_2; \Psi \vdash_e \langle t_1, t_2 \rangle : A \otimes B}$	$\frac{\Delta; \Psi \vdash_e t : A[X \leftarrow \mu X.A]}{\Delta; \Psi \vdash_e \mathbf{fold} t : \mu X.A}$		
$\frac{\Psi \vdash_\omega f : A \leftrightarrow B \quad \Delta; \Psi \vdash_e t : A}{\Delta; \Psi \vdash_e f t : B}$	$\frac{\vdash_\omega \omega : A \leftrightarrow B \quad \Delta; \Psi \vdash_e t : A}{\Delta; \Psi \vdash_e \omega t : B}$		
$\frac{\Delta_1; \Psi \vdash_e t_1 : A_1 \otimes \dots \otimes A_n \quad \Delta_2, x_1 : A_1, \dots, x_n : A_n; \Psi \vdash_e t_2 : B}{\Delta_1, \Delta_2; \Psi \vdash_e \mathbf{let} (x_1, \dots, x_n) = t_1 \mathbf{in} t_2 : B}$			

Table 4.2.: Typing of terms and expressions

Example 4.2.5. We can define the iso of type: $A \oplus (B \oplus C) \leftrightarrow C \oplus (A \oplus B)$ as

$$\left\{ \begin{array}{l} \mathbf{inj}_l a \quad \leftrightarrow \mathbf{inj}_r \mathbf{inj}_l a \\ \mathbf{inj}_r \mathbf{inj}_l b \leftrightarrow \mathbf{inj}_r \mathbf{inj}_r b \\ \mathbf{inj}_r \mathbf{inj}_r c \leftrightarrow \mathbf{inj}_l c \end{array} \right\}$$

Finally, our language is equipped with a rewriting system \rightarrow on terms, defined in Definition 4.2.9, that follows a deterministic call-by-value strategy: each argument of a function is fully evaluated before applying the substitution. This is done through the use of an evaluation context $C[]$, as for the β -reduction defined in Section 3.1.1. Due to the deterministic nature of the strategy, we directly obtain the unicity of the normal form. The evaluation of an iso applied to a value relies on a notion of pattern-matching: the argument is matched against the left-hand-side of each clause until one of them matches (written $\sigma[v] = v'$), in which case the pattern-matching, as defined in Definition 4.5, returns a substitution σ that sends variables to values. Because we ensure exhaustivity and

$$\begin{array}{c}
 \frac{\Delta_1 \vdash_e v_1 : A \quad \dots \quad \Delta_n \vdash_e v_n : A \quad \text{OD}_A(\{v_1, \dots, v_n\})}{\Delta_1; \Psi \vdash_e e_1 : B \quad \dots \quad \Delta_n; \Psi \vdash_e e_n : B \quad \text{OD}_B(\{Val(e_1), \dots, Val(e_n)\})} \\
 \Psi \vdash_\omega \{v_1 \leftrightarrow e_1 \mid \dots \mid v_n \leftrightarrow e_n\} : A \leftrightarrow B. \\
 \\
 \frac{f : \alpha \vdash_\omega f : \alpha}{f : \alpha \vdash_\omega f : \alpha} \quad \frac{f : \alpha \vdash_\omega \omega : \alpha \quad \mathbf{fix} \ f.\omega \text{ is structurally recursive}}{\Psi \vdash_\omega \mathbf{fix} \ f.\omega : \alpha}
 \end{array}$$

Table 4.3.: Typing of isos

$$\begin{array}{c}
 \frac{}{\text{OD}_A(\{x\})} \quad \frac{}{\text{OD}_1(\{()\})} \quad \frac{\text{OD}_A(S) \quad \text{OD}_B(T)}{\text{OD}_{A \oplus B}(\{\mathbf{inj}_l \ v \mid v \in S\} \cup \{\mathbf{inj}_r \ v \mid v \in T\})} \\
 \frac{\text{OD}_{A[X \leftarrow \mu X.A]}(S)}{\text{OD}_{\mu X.A}(\{\mathbf{fold} \ v \mid v \in S\})} \quad \frac{\text{OD}_A(\pi_1(S)) \text{ and } \forall v \in \pi_1(S), \text{OD}_B(S_v^1) \text{ or } \text{OD}_B(\pi_2(S)) \text{ and } \forall v \in \pi_2(S), \text{OD}_A(S_v^2)}{\text{OD}_{A \otimes B}(S = \{\langle v_1, v_1' \rangle, \dots, \langle v_n, v_n' \rangle\})}
 \end{array}$$

Table 4.4.: Exhaustivity and Non-Overlapping

non-overlapping (Lemma 4.2.7), the pattern-matching can always occur on well-typed terms. The *support* of a substitution σ is defined as $\text{supp}(\sigma) = \{x \mid (x \mapsto v) \in \sigma\}$.

Definition 4.2.6 (Substitution). *Applying substitution σ on an expression t , written $\sigma(t)$, is defined, as:*

- $\sigma(()) = ()$,
- $\sigma(x) = v$ if $\{x \mapsto v\} \subseteq \sigma$,
- $\sigma(\mathbf{inj}_r \ t) = \mathbf{inj}_r \ \sigma(t)$,
- $\sigma(\mathbf{inj}_l \ t) = \mathbf{inj}_l \ \sigma(t)$,
- $\sigma(\langle t, t' \rangle) = \langle \sigma(t), \sigma(t') \rangle$,
- $\sigma(\omega \ t) = \omega \ \sigma(t)$,
- $\sigma(\mathbf{let} \ p = t_1 \ \mathbf{in} \ t_2) = (\mathbf{let} \ p = \sigma(t_1) \ \mathbf{in} \ \sigma(t_2))$.

Lemma 4.2.7 ($\text{OD}_A(A)$ ensures exhaustivity and non-overlapping.). *Let $\text{OD}_A(S)$ then for all close value v of type A , there exists a unique $v' \in S$ and a unique σ such that $\sigma[v'] = v$.*

Proof. By induction on $\text{OD}_A(S)$:

- $\text{OD}_A(\{x\})$ and $\text{OD}_1(\{()\})$ are direct.

$$\begin{array}{c}
 \frac{\sigma[e] = e'}{\sigma[\mathbf{inj}_l e] = \mathbf{inj}_l e'} \quad \frac{\sigma[e] = e'}{\sigma[\mathbf{inj}_r e] = \mathbf{inj}_r e'} \quad \frac{\sigma = \{x \mapsto e\}}{\sigma[x] = e} \quad \frac{\sigma[e] = e'}{\sigma[\mathbf{fold} e] = \mathbf{fold} e'} \\
 \frac{\sigma_2[e_1] = e'_1 \quad \sigma_1[e_2] = e'_2 \quad \text{supp}(\sigma_1) \cap \text{supp}(\sigma_2) = \emptyset \quad \sigma = \sigma_1 \cup \sigma_2}{\sigma[\langle e_1, e_2 \rangle] = \langle e'_1, e'_2 \rangle} \quad \frac{}{\sigma[()] = ()}
 \end{array}$$

Table 4.5.: Pattern-matching

- $\text{OD}_{A \oplus B}(\{\mathbf{inj}_l v \mid v \in S_A\} \cup \{\mathbf{inj}_r v \mid v \in S_B\})$ let $v = \mathbf{inj}_l \tilde{v}$ then by induction hypothesis on S_A there exists $v_1 \in S_A$ such that $\sigma_1[v_1] = \tilde{v}$, hence $\sigma_1[\mathbf{inj}_l v_1] = \mathbf{inj}_l \tilde{v}$. Now assume there also exists $v_2 \in S_A$ such that $\sigma_2[v_2] = \tilde{v}$, by induction hypothesis we have that $v_1 = v_2$ hence $\mathbf{inj}_l v_1 = \mathbf{inj}_l v_2$.

Similar case for the \mathbf{inj}_r .

- Similar case for the \mathbf{fold} .
- Assuming we are in the first case of the disjunction in the premise of $\text{OD}_{A \otimes B}$, the other case being similar:

Take some value $\langle \tilde{v}, \tilde{v}' \rangle$, by induction hypothesis we know that there exists some $v_i \in \{v_1, \dots, v_n\}$ such that $\sigma_i[v_i] = \tilde{v}$ and therefore that there also exists some $v'_i \in S_{v_i}^1$ such that $\sigma'_i[v'_i] = \tilde{v}'$.

Now assuming that there exists another pair $\langle v_1, v_2 \rangle \in S$ such that $\sigma[\langle v_1, v_2 \rangle] = \langle \tilde{v}, \tilde{v}' \rangle$, by induction hypothesis we know that $v_1 = v_i$ and that therefore $v_2 \in S_{v_i}^1 = S_{v_1}^1$ so $v_2 = v'_i$. \square

Definition 4.2.8 (Evaluation Contexts). *The evaluations contexts C are defined as:*

$$C ::= [] \mid \mathbf{inj}_l C \mid \mathbf{inj}_r C \mid \omega C \mid \mathbf{let} p = C \mathbf{in} t \mid \langle C, v \rangle \mid \langle v, C \rangle$$

Definition 4.2.9 (Evaluation relation \rightarrow). *The rewriting system \rightarrow is defined as:*

$$\begin{array}{c}
 \frac{t_1 \rightarrow t_2}{C[t_1] \rightarrow C[t_2]} \text{Cong} \quad \frac{\sigma[p] = v}{\mathbf{let} p = v \mathbf{in} t \rightarrow \sigma(t)} \text{LetE} \\
 \frac{}{(\mathbf{fix} f.\omega) \rightarrow \omega[f \leftarrow (\mathbf{fix} f.\omega)]} \text{IsoRec} \\
 \frac{\sigma[v_i] = v'}{\{v_1 \leftrightarrow e_1 \mid \dots \mid v_n \leftrightarrow e_n\} v' \rightarrow \sigma(e_i)} \text{IsoApp}
 \end{array}$$

As usual we note \rightarrow^* for the reflexive transitive closure of \rightarrow .

As mentioned above, from any iso $\omega : A \leftrightarrow B$ we can build its inverse $\omega^\perp : B \leftrightarrow A$, the inverse operation is defined inductively on ω and is given in Definition 4.2.10.

Definition 4.2.10 (Inversion). *Given an iso ω , we define its dual ω^\perp as:*

- $f^\perp = f$
- $(\mathbf{fix} f.\omega)^\perp = \mathbf{fix} f.\omega^\perp$
- $\{(v_i \leftrightarrow e_i)_{i \in I}\}^\perp = \{((v_i \leftrightarrow e_i)^\perp)_{i \in I}\}$

and the inverse of a clause as:

$$\left(\begin{array}{l} v_1 \leftrightarrow \mathbf{let} p_1 = \omega_1 p'_1 \mathbf{in} \\ \dots \\ \mathbf{let} p_n = \omega_n p'_n \mathbf{in} v'_1 \end{array} \right)^\perp := \left(\begin{array}{l} v'_1 \leftrightarrow \mathbf{let} p'_n = \omega_n^\perp p_n \mathbf{in} \\ \dots \\ \mathbf{let} p'_1 = \omega_1^\perp p_1 \mathbf{in} v_1 \end{array} \right).$$

We can show that the inverse is well-typed and behaves as expected:

Lemma 4.2.11 (Inversion is well-typed). *If $\Psi \vdash_\omega \omega : A \leftrightarrow B$, then $\Psi \vdash_\omega \omega^\perp : B \leftrightarrow A$.*

Proof. w.l.o.g. consider $\omega = \{v_1 \leftrightarrow e_1 \mid \dots \mid v_n \leftrightarrow e_n\}$, we look at one clause in particular and its dual:

$$\left(\begin{array}{l} v_1 \leftrightarrow \mathbf{let} p_1 = \omega_1 p'_1 \mathbf{in} \\ \dots \\ \mathbf{let} p_n = \omega_n p'_n \mathbf{in} v'_1 \end{array} \right)^\perp := \left(\begin{array}{l} v'_1 \leftrightarrow \mathbf{let} p'_n = \omega_n^\perp p_n \mathbf{in} \\ \dots \\ \mathbf{let} p'_1 = \omega_1^\perp p_1 \mathbf{in} v_1 \end{array} \right).$$

By typing we know that $\Delta \vdash_e v_1 : A$ and $\Delta; \Psi \vdash_e \mathbf{let} p_1 = \omega_1 p'_1 \mathbf{in} \dots v'_1 : B$

Δ can be split into $\Delta_1, \dots, \Delta_n, \Delta_{n+1}$ and for all $1 \leq i \leq n$ we get that the typing judgment of the expression $\mathbf{let} p_i = \omega_i p'_i \mathbf{in} \dots$ generates the new typing judgment $\Gamma_i^{i+1}, \dots, \Gamma_i^{n+1}$. For all i we get that $\Delta_i \bigcup_{j=1}^{i-1} \Gamma_j^{i-1} \vdash_e \omega p'_i$, finally v'_1 is typed by $\Delta_{n+1} \bigcup_{i=1}^n \Gamma_i^{n+1}$.

When typing the dual clause, we start with contexts $\Delta_{n+1} \bigcup_{i=1}^n \Gamma_i^{n+1}$. We have from hypothesis that:

- Each $\omega_i^\perp p_i$ is typed by $\bigcup_{j=i+1}^n \Gamma_j^n$, which is possible by our typing hypothesis.
- Each p'_i generates the contexts $\Delta_i, \bigcup_{j=1}^i \Gamma_j^i$.

At the end we end up with $\Delta_1, \dots, \Delta_n, \Delta_{n+1} \vdash_e v_1$ which is typable by our hypothesis. \square

In order to show that our isos are indeed isomorphisms, we need the following lemma:

Lemma 4.2.12 (Commutativity of substitution). *Let σ_1, σ_2 and v , such that $\sigma_1 \cup \sigma_2$ closes v and $\text{supp}(\sigma_1) \cap \text{supp}(\sigma_2) = \emptyset$ then $\sigma_1(\sigma_2(v)) = \sigma_2(\sigma_1(v))$*

Proof. Direct by induction on v as σ_1 and σ_2 have disjoint support: In the case where $v = x$ then either $\{x \mapsto v'\} \in \sigma_1$ or $\{x \mapsto v'\} \in \sigma_2$ and hence $\sigma_1(\sigma_2(x)) = v' = \sigma_2(\sigma_1(x))$. All the other case are by direct induction hypothesis as the substitutions enter the subterms. \square

Theorem 4.2.13 (Isos are isomorphisms). *For all well-typed isos $\vdash_\omega \omega : A \leftrightarrow B$, and for all well-typed values $\vdash_e v : A$, if $(\omega (\omega^\perp v)) \rightarrow^* v'$ then $v = v'$.*

Proof. By induction hypothesis on the size of ω :

- Case where $\omega = \{v_1 \leftrightarrow v_1 \mid \dots \mid v_n \leftrightarrow v'_n\}$ then $\omega^\perp(\omega v_0)$, by non-overlapping and exhaustivity there exists a v_i such that $\sigma[v_i] = v_0$ and hence the term reduces to $\omega^\perp \sigma(v'_i)$. It is clear that $\sigma[v'_i] = \sigma(v_i)$ and hence the terms reduces to $\sigma(v_i)$, but by the first pattern-matching we know that $\sigma(v_i) = v_0$, which concludes the case.
- Case where $\omega = \{v_1 \leftrightarrow e_1 \mid \dots \mid v_n \leftrightarrow e_n\}$,
for simplicity of writing we write a single clause:

$$\left(\begin{array}{l} v_1 \leftrightarrow \text{let } p_1 = \omega_1 p'_1 \text{ in} \\ \dots \\ \text{let } p_n = \omega_n p'_n \text{ in } v'_1 \end{array} \right)^\perp := \left(\begin{array}{l} v'_1 \leftrightarrow \text{let } p'_n = \omega_n^\perp p_n \text{ in} \\ \dots \\ \text{let } p'_1 = \omega_1^\perp p_1 \text{ in } v_1 \end{array} \right).$$

Take some closed value $\vdash_e v_0 : A$ such that $\sigma[v_1] = v_0$.

By linearity, we can decompose σ into $\sigma_1, \dots, \sigma_n, \sigma_{n+1}$ such that, after substitution we obtain

$$\text{let } p_1 = \omega_1 \sigma_1(p'_1) \text{ in } \dots \text{let } p_n = \omega_n \sigma_n(p'_n) \text{ in } \sigma_{n+1}(v'_1)$$

Given By Lemma 4.2.19, each **let** construction will reduce, and by the rewriting strategy we get:

$$\begin{array}{l} \text{let } p_1 = \omega_1 \sigma_1(p'_1) \text{ in} \\ \dots \\ \text{let } p_n = \omega_n \sigma_n(p'_n) \text{ in} \\ \sigma_{n+1}(v'_1) \end{array} \rightarrow \begin{array}{l} \text{let } p_1 = \overline{v_1} \text{ in} \\ \dots \\ \text{let } p_n = \omega_n \sigma_n(p'_n) \text{ in} \\ \sigma_{n+1}(v'_1) \end{array} \rightarrow \begin{array}{l} \text{let } p_2 = \omega_2 \gamma_1^2(\sigma_1(p'_1)) \text{ in} \\ \dots \\ \text{let } p_n = \omega_n \gamma_1^n(\sigma_n(p'_n)) \text{ in} \\ \sigma_{n+1}(v'_1) \end{array}$$

The final term reduces to $\gamma_n^n(\dots(\gamma_1^n(\sigma_{n+1}(v'_1)))\dots)$ and creates a new substitution γ_i , the term will hence reduce to $\gamma_n(\dots(\gamma_1(\sigma_{n+1}(v'_1)))\dots)$. Let $\delta = \cup_i \gamma_i \cup \sigma_{n+1}$

We are left to evaluate

$$\left(\begin{array}{l} v'_1 \leftrightarrow \text{let } p'_n = \omega_n^{-1} p_n \text{ in} \\ \dots \\ \text{let } p'_1 = \omega_1^{-1} p_1 \text{ in } v_1 \end{array} \right) \delta(v'_1)$$

We get $\delta[v'_1] = \delta(v'_1)$.

We know that each γ_i closes only p_i , we can therefore substitute the term as:

$$\text{let } p'_n = \omega_n \gamma_n(p_n) \text{ in } \dots \text{let } p'_1 = \omega_1 \gamma_1(p_1) \text{ in } \sigma_{n+1}(v'_1)$$

By induction hypothesis, Each **let** clause will re-create the substitution σ_i , we know this as the fact that the initial **let** construction, **let** $p_i = \omega_i \sigma_i(p'_i)$ **in** \dots reduces to **let** $p_i = v_i$ **in** \dots . While the new one, **let** $p'_i = \omega^\perp \gamma_i(p_i)$ **in** \dots , is, by definition of the substitution the same as **let** $p'_i = \omega^\perp v_i$ **in** \dots .

Then, since we know that v_i is the result of $\omega \sigma_i(p'_i)$, we get by induction hypothesis $\sigma(p'_i)$ as the result of the evaluation.

Therefore, after rewriting we obtain: $\sigma_n(\dots(\sigma_1(\sigma_{n+1}(v'_1)))\dots)$. By Lemma 4.2.12 we get $\sigma_1(\dots(\sigma_n(\sigma_{n+1}(v'_1)))\dots)$ which is equal to v . \square

Example 4.2.14. We give the encoding of the isomorphism $\text{map}(\omega)$ and its inverse: for any given iso $\vdash_\omega \omega : A \leftrightarrow B$ in our language, we can define $\text{map}(\omega) : [A] \leftrightarrow [B]$ where $[A] = \mu X. \mathbf{1} \oplus (A \otimes X)$ is the type of lists of type A and $[]$ is the empty list (**fold** (**inj**_l ())) and $h :: t$ is the list construction (**fold** (**inj**_r ($\langle h, t \rangle$))). We also give its dual $\text{map}(\omega)^\perp$ below, as given by Definition 4.2.10.

$$\text{map}(\omega) = \mathbf{fix} f. \left\{ \begin{array}{l} [] \leftrightarrow [] \\ h :: t \leftrightarrow \text{let } h' = \omega h \text{ in} \\ \quad \text{let } t' = f t \text{ in} \\ \quad h' :: t' \end{array} \right\} : [A] \leftrightarrow [B]$$

$$\text{map}(\omega)^\perp = \mathbf{fix} f. \left\{ \begin{array}{l} [] \leftrightarrow [] \\ h' :: t' \leftrightarrow \text{let } t = f t' \text{ in} \\ \quad \text{let } h = \omega^\perp h' \text{ in} \\ \quad h :: t \end{array} \right\} : [B] \leftrightarrow [A]$$

Remark 4.2.15. In Example 4.2.5 and Example 4.2.14, the left and right-hand side of the \leftrightarrow on each function respect both the criteria of exhaustivity —every-value of each type is being covered by at least one expression— and non-overlapping —no two expressions cover the same value. Both isos are therefore bijections.

The language enjoys the standard properties of typed languages of progress and subject reduction, but requires a substitution lemma on both terms and isos:

Lemma 4.2.16 (Substitution Lemma of variables).

- For all values v_1, \dots, v_n , types A_1, \dots, A_n and context $\Delta_1, \dots, \Delta_n$ such that for $i \in \{1, \dots, n\}$, $\Delta_i \vdash_e v_i : A_i$.
- For all term t , type B and context Γ and variables $x_1, \dots, x_n \in \text{FV}(t)$ such that $\Gamma, x_1 : A_1, \dots, x_n : A_n \vdash_e t : B$

Let $\sigma = \{x_1 \mapsto v_1, \dots, x_n \mapsto v_n\}$ then $\Gamma, \Delta_1, \dots, \Delta_n \vdash_e \sigma(t) : B$

Proof. By induction on t .

- Case x , then $\Gamma = \emptyset$ and we have $\sigma = \{x \mapsto v\}$ for some v of type B under some context Δ , then we get $\Delta \vdash_e \sigma(x) : B$ which leads to $\Delta \vdash_e v : B$ which is typable by our hypothesis.
- Case $()$, nothing to do.
- Case $\text{inj}_l t'$, by substitution we have $\sigma(\text{inj}_l t') = \text{inj}_l \sigma(t')$ and by typing we get
$$\frac{\Gamma, \Delta_1, \dots, \Delta_n \vdash_e \sigma(t')}{\Gamma, \Delta_1, \dots, \Delta_n \vdash_e \text{inj}_l \sigma(t')}$$
 which is typable by induction hypothesis on t' .
- Case $\text{inj}_r t', \text{fold } t', \omega t'$ are similar.
- Case $\langle t_1, t_2 \rangle$, by typing we get that we can split Γ into Γ_1, Γ_2 and the variables x_1, \dots, x_n are split into two parts for typing both t_1 or t_2 depending on whenever or not a variable occurs freely in t_1 or t_2 , w.l.o.g. say that x_1, \dots, x_l are free in t_1 and x_{l+1}, \dots, x_n are free in t_2 then we get:

$$\frac{\Gamma_1, x_1 : A_1, \dots, x_l : A_l \vdash_e t_1 : B_1 \quad \Gamma_2, x_{l+1} : A_{l+1}, \dots, x_n : A_n \vdash_e t_2 : B_2}{\Gamma_1, \Gamma_2, x_1 : A_1, \dots, x_l : A_l, x_{l+1} : A_{l+1}, \dots, x_n : A_n \vdash_e \langle t_1, t_2 \rangle : B_1 \otimes B_2}$$

By substitution we get that $\sigma(\langle t_1, t_2 \rangle) = \langle \sigma(t_1), \sigma(t_2) \rangle$ so we get the following typing derivation which is completed by induction hypothesis on the subterms:

$$\frac{\Gamma_1, \Delta_1, \dots, \Delta_l \vdash_e t_1 : B_1 \quad \Gamma_2, \Delta_{l+1}, \dots, \Delta_n \vdash_e t_2 : B_2}{\Gamma_1, \Gamma_2, \Delta_1, \dots, \Delta_l, \Delta_{l+1}, \dots, \Delta_n \vdash_e \langle t_1, t_2 \rangle : B_1 \otimes B_2}$$

- Case $\text{let } p = t_1 \text{ in } t_2$ Similar to the case of the tensor. □

Lemma 4.2.17 (Substitution Lemma Of Isos). *If :*

- $\Delta; f : \alpha \vdash_e t : A$
- $g : \beta \vdash_\omega \omega : \alpha$

Then $\Delta; g : \beta \vdash_e t[f \leftarrow \omega] : A$.

If :

- $f : \alpha \vdash_\omega \omega_1 : \beta$

- $h : \gamma \vdash_{\omega} \omega_2 : \alpha$

Then $h : \gamma \vdash_{\omega} \omega_1[f \leftarrow \omega_2] : \beta$.

Proof. We prove those two propositions by mutual induction on t and ω_1 .

Terms, by induction on t .

- If $t = x$ or $t = ()$ then there is nothing to do.
- If $t = \text{inj}_l t'$ or $\text{inj}_r t'$ or $\text{fold } t'$ or $\langle t_1, t_2 \rangle$ or $\text{let } p = t_1 \text{ in } t_2$, then similarly to the proof of Lemma 4.2.16 the substitution goes to the subterms and we can apply the induction hypothesis.
- If $t = \omega' t'$. In that case, the substitution goes to both subterms: $t[f \leftarrow \omega] = (\omega'[f \leftarrow \omega]) (t'[f \leftarrow \omega])$ and by induction hypothesis on t' and by the mutually recursive proof.

Isos, by induction on ω_1 .

- If $\omega_1 = f$, then we get $h : \gamma \vdash_{\omega} f[f \leftarrow \omega_2] : \beta$ which is typable by hypothesis.
- If $\omega_1 = g \neq f$ is impossible by our typing hypothesis.
- If $\omega_1 = \mathbf{fix } g.\omega$, then by typing f does not occur in ω_1 so nothing happens.
- If $\omega_1 = \{v_1 \leftrightarrow e_1 \mid \dots \mid v_n \leftrightarrow e_n\}$, then, by definition of the substitution we have that

$$\begin{aligned} & \{v_1 \leftrightarrow e_1 \mid \dots \mid v_n \leftrightarrow e_n\}[f \leftarrow \omega_2] \\ &= \{v_1 \leftrightarrow e_1[f \leftarrow \omega_2] \mid \dots \mid v_n[f \leftarrow \omega_2] \leftrightarrow e_n[f \leftarrow \omega_2]\} \end{aligned}$$

in which case we apply the substitution lemma of isos on terms. \square

From that we can directly deduce subject reduction and progress:

Lemma 4.2.18 (Subject Reduction). *If $\Delta; \Psi \vdash_e t : A$ and $t \rightarrow t'$ then $\Delta; \Psi \vdash_e t' : A$.*

Proof. By induction on $t \rightarrow t'$ and direct by Lemma 4.2.16 and Lemma 4.2.17 \square

Lemma 4.2.19 (Progress). *If $\vdash_e t : A$ then, either t is a value, or $t \rightarrow t'$.*

Proof. Direct by induction on $\vdash_e t : A$. The two possible reduction cases, ωv and $\text{let } p = v \text{ in } t$ always reduce by typing, pattern-matching and by Lemma 4.2.7. \square

4.3. Computational Content

In this section, we study the computational content of our language. We show that we can encode Recursive Primitive Permutations [PPR20] (RPP), which shows us that we can encode at least all Primitive Recursive Functions [RJ87].

4.3.1. From RPP to Isos

We start by defining the type of strictly positive natural numbers, npos , as $\text{npos} = \mu X. \mathbf{1} \oplus X$. We define \underline{n} , the encoding of a positive natural number into a value of type npos as $\underline{1} = \text{fold inj}_l ()$ and $\underline{n+1} = \text{fold inj}_r \underline{n}$. Finally, we define the type of integers as $Z = \mathbf{1} \oplus (\text{npos} \oplus \text{npos})$ along with \bar{z} the encoding of any $z \in \mathbb{Z}$ into a value of type Z defined as: $\bar{0} = \text{inj}_l ()$, $\bar{z} = \text{inj}_r \text{inj}_l \underline{z}$ for z positive, and $\bar{z} = \text{inj}_r \text{inj}_r \underline{-z}$ for z negative. Given some function $f \in \text{RPP}^k$, we will build an iso $\text{isos}(f) : Z^k \leftrightarrow Z^k$ which simulates f . $\text{isos}(f)$ is defined by the size of the proof that f is in RPP^k .

Definition 4.3.1 (Encoding of the primitives).

- The Sign-change is

$$\text{isos}(\text{Sign}) = \left\{ \begin{array}{ll} \text{inj}_r \text{inj}_l x & \leftrightarrow \text{inj}_r \text{inj}_r x \\ \text{inj}_r \text{inj}_r x & \leftrightarrow \text{inj}_r \text{inj}_l x \\ \text{inj}_l () & \leftrightarrow \text{inj}_l () \end{array} \right\} : Z \leftrightarrow Z$$

- The identity is $\text{isos}(\text{Id}) = \{x \leftrightarrow x\} : Z \leftrightarrow Z$
- The Swap is $\text{isos}(\mathcal{X}) = \{(x, y) \leftrightarrow (y, x)\} : Z^2 \leftrightarrow Z^2$
- The Successor is

$$\text{isos}(S) = \left\{ \begin{array}{ll} \text{inj}_l () & \leftrightarrow \text{inj}_r \text{inj}_l \text{fold inj}_l () \\ \text{inj}_r \text{inj}_l x & \leftrightarrow \text{inj}_r \text{inj}_l \text{fold inj}_r x \\ \text{inj}_r \text{inj}_r \text{fold inj}_l () & \leftrightarrow \text{inj}_l () \\ \text{inj}_r \text{inj}_r \text{fold inj}_r x & \leftrightarrow \text{inj}_r \text{inj}_r x \end{array} \right\} : Z \leftrightarrow Z$$

- The Predecessor is the inverse of the Successor: $\text{isos}(P) = \text{isos}(S)^\perp$.

Definition 4.3.2 (Encoding of Composition). Let $f, g \in \text{RPP}^j$, $\omega_f = \text{isos}(f)$ and $\omega_g = \text{isos}(g)$ the isos encoding f and g , we build $\text{isos}(f; g)$ of type $Z^j \leftrightarrow Z^j$ as:

$$\text{isos}(f; g) = \left\{ \begin{array}{ll} (x_1, \dots, x_j) & \leftrightarrow \text{let } (y_1, \dots, y_j) = \omega_f(x_1, \dots, x_j) \text{ in} \\ & \text{let } (z_1, \dots, z_j) = \omega_g(y_1, \dots, y_j) \text{ in} \\ & (z_1, \dots, z_j) \end{array} \right\}$$

Definition 4.3.3 (Encoding of Parallel Composition). *Let $f \in \text{RPP}^j$ and $g \in \text{RPP}^k$, and $\omega_f = \text{isos}(f)$ and $\omega_g = \text{isos}(g)$, we define $\text{isos}(f \parallel g)$ of type $Z^{j+k} \leftrightarrow Z^{j+k}$ as:*

$$\text{isos}(f \parallel g) = \left\{ \begin{array}{l} (x_1, \dots, x_j, y_1, \dots, y_k) \leftrightarrow \text{let } (x'_1, \dots, x'_j) = \omega_f(x_1, \dots, x_j) \text{ in} \\ \text{let } (y'_1, \dots, y'_k) = \omega_g(y_1, \dots, y_k) \text{ in} \\ (x'_1, \dots, x'_j, y'_1, \dots, y'_k) \end{array} \right\}$$

Definition 4.3.4 (Encoding of Finite Iteration). *Let $f \in \text{RPP}^k$, and $\omega_f = \text{isos}(f)$, we encode the finite iteration $\text{It}[f] \in \text{RPP}^{k+1}$ with the help of an auxiliary iso, ω_{aux} , of type $Z^k \otimes \text{npos} \leftrightarrow Z^k \otimes \text{npos}$ doing the finite iteration using npos , defined as:*

$$\omega_{\text{aux}} = \text{fix}g. \left\{ \begin{array}{l} (\vec{x}, \text{fold inj}_l ()) \leftrightarrow \text{let } \vec{y} = \omega_f \vec{x} \text{ in} \\ (\vec{y}, \text{fold inj}_l ()) \\ (\vec{x}, \text{fold inj}_r n) \leftrightarrow \text{let } (\vec{y}) = \omega_f(\vec{x}) \text{ in} \\ \text{let } (\vec{z}, n') = g(\vec{y}, n) \text{ in} \\ (\vec{z}, \text{fold inj}_r n') \end{array} \right\}$$

We can now properly define $\text{isos}(\text{It}[f])$ of type $Z^{k+1} \leftrightarrow Z^{k+1}$ as:

$$\text{isos}(\text{It}[f]) = \left\{ \begin{array}{l} (\vec{x}, \text{inj}_l ()) \leftrightarrow (\vec{x}, \text{inj}_l ()) \\ (\vec{x}, \text{inj}_r \text{inj}_l z) \leftrightarrow \text{let } (\vec{y}, z') = \omega_{\text{aux}}(\vec{x}, z) \text{ in} \\ (\vec{y}, \text{inj}_r \text{inj}_l z') \\ (\vec{x}, \text{inj}_r \text{inj}_r z) \leftrightarrow \text{let } (\vec{y}, z') = \omega_{\text{aux}}(\vec{x}, z) \text{ in} \\ (\vec{y}, \text{inj}_r \text{inj}_r z') \end{array} \right\}$$

Definition 4.3.5 (Encoding of Selection). *Let $f, g, h \in \text{RPP}^k$ and $\omega_f = \text{isos}(f)$, $\omega_g = \text{isos}(g)$, $\omega_h = \text{isos}(h)$. We define $\text{isos}(\text{If}[f, g, h])$ of type $Z^{k+1} \leftrightarrow Z^{k+1}$ as:*

$$\text{isos}(\text{If}[f, g, h]) = \left\{ \begin{array}{l} (\vec{x}, \text{inj}_r \text{inj}_l z) \leftrightarrow \text{let } \vec{x}' = \omega_f(\vec{x}) \text{ in } (\vec{x}', \text{inj}_r \text{inj}_l z) \\ (\vec{x}, \text{inj}_l ()) \leftrightarrow \text{let } \vec{x}' = \omega_g(\vec{x}) \text{ in } (\vec{x}', \text{inj}_l ()) \\ (\vec{x}, \text{inj}_r \text{inj}_r z) \leftrightarrow \text{let } \vec{x}' = \omega_h(\vec{x}) \text{ in } (\vec{x}', \text{inj}_r \text{inj}_r z) \end{array} \right\}$$

Theorem 4.3.6 (The encoding is well-typed). *If $f \in \text{RPP}^k$, then $\vdash_{\omega} \text{isos}(f) : Z^k \leftrightarrow Z^k$.*

Proof. By induction on f , for the two compositions, iteration, and selection the variables \vec{x} are all of type Z , while for ω_{aux} the last argument is of type npos . The predicate OD_Z is always satisfied as the left columns of the n -fold tensor are always variables and the right-most argument on each isos always satisfies OD_Z . \square

Theorem 4.3.7 (Simulation). *Let $f \in \text{RPP}^k$ and n_1, \dots, n_k elements of \mathbb{Z} such that $f(n_1, \dots, n_k) = (m_1, \dots, m_k)$ then $\text{isos}(f)(\bar{n}_1, \dots, \bar{n}_k) \rightarrow^* (\bar{m}_1, \dots, \bar{m}_k)$*

Proof. By induction on f .

- Direct for the identity, swap and sign-change.
- For the Successor:

$$\omega = \left\{ \begin{array}{ll} \text{inj}_l () & \leftrightarrow \text{inj}_r \text{inj}_l \text{fold inj}_l () \\ \text{inj}_r \text{inj}_l x & \leftrightarrow \text{inj}_r \text{inj}_l \text{fold inj}_r x \\ \text{inj}_r \text{inj}_r \text{fold inj}_l () & \leftrightarrow \text{inj}_l () \\ \text{inj}_r \text{inj}_r \text{fold inj}_r x & \leftrightarrow \text{inj}_r \text{inj}_r x \end{array} \right\}$$

we do it by case analysis on the sole input n .

- $n = 0$ then $\bar{0} = \text{inj}_l ()$ and $\omega \text{inj}_l () \rightarrow \text{inj}_r (\text{inj}_l (\text{fold} (\text{inj}_l ()))) = \bar{1}$
- $n = -1$ then $\bar{-1} = \text{inj}_r (\text{inj}_r (\text{fold} (\text{inj}_l ())))$, so the term reduces to $\text{inj}_l () = \bar{0}$
- Case $n < -1$, we have $\bar{n} = \text{inj}_r (\text{inj}_r (\text{fold} (\text{inj}_r \underline{n'})))$ with $\underline{n'} = \underline{n+1}$, by pattern-matching we get $\text{inj}_r \text{inj}_r x$ which is $\bar{n'}$
- Case $n > 1$ is similar.
- The Predecessor is the dual of the Successor.
- **Composition & Parallel composition:** Direct by induction hypothesis on ω_f and ω_g : for the composition, ω_f if first applied on all the input and then ω_g on the result of ω_f . For the parallel composition, ω_f is applied on the first j arguments and ω_g on the argument $j+1$ to k before concatenating the results from both isos.
- **Finite Iteration:** $\text{It}[f]$.

We need the following lemma: $\omega_{\text{aux}}(\bar{x}_1, \dots, \bar{x}_n, \underline{z}) \rightarrow^* (\bar{z}_1, \dots, \bar{z}_n, \underline{z})$ where z is a non-zero integer and $(z_1, \dots, z_n) = f^{|z|}(x_1, \dots, x_n)$ which can be shown by induction on $|z|$: the case $z = 1$ and $\bar{z} = \text{fold inj}_l ()$ is direct by induction hypothesis on ω_f . Then if $z = n+1$ we get it directly by induction hypothesis on both ω_f and our lemma.

Then, for $\text{isosIt}[f]$ we do it by case analysis on the last argument: when it is $\bar{0}$ then we simply return the result, if it is \bar{z} for z (no matter if strictly positive or strictly negative) then we enter ω_{aux} , and apply the previous lemma.

- **Conditional:** $\text{If}[f, g, h]$. Direct by case analysis of the last value and by induction hypothesis on $\omega_f, \omega_g, \omega_h$. \square

Remark 4.3.8. Notice that $\text{isos}(f)^\perp \neq \text{isos}(f^{-1})$, due to the fact that $\text{isos}(f)^\perp$ will inverse the order of the `let` constructions, which will not be the case for $\text{isos}(f^{-1})$. They can nonetheless be considered equivalent up to a permutation of `let` constructions and renaming of variable.

4.4. Proof Theoretical Content

In this section, we want to relate our language of isos to proofs in a suitable logic. As mentioned earlier, an iso $\vdash_\omega \omega : A \leftrightarrow B$ corresponds to both a computation sending a value of type A to a result of type B and a computation sending a value of type B to a result of type A . Therefore, we want to be able to translate an iso into a proof isomorphism: two proofs π and π^\perp of respectively $A \vdash B$ and $B \vdash A$ such that their composition reduces through the cut-elimination to the identity either on A or on B depending on the way we make the cut between those proofs.

Since we are working in a linear system with inductive types we will use the logic μMALL : linear logic with least and greatest fixed points, which allows us to reason about inductive and coinductive statements. μMALL allows us to consider infinite derivation trees, which is required as our isos can contain recursive variables. As not all derivations of μMALL are considered as proofs, we need to be careful in the way we translate isos into derivations. We will also need to show that the obtained derivation are indeed proofs, this is ensured by the structurally recursive constraints.

4.4.1. Translating isos into μMALL

We start by giving the translation from isos to pre-proofs, and then, in Theorem 4.4.20 show that they are actually proofs, therefore obtaining a *static* correspondence between programs and proofs. We then show in Theorem 4.4.29 that our translation entails a *dynamic* correspondence between the evaluation procedure of our language and the cut-elimination procedure of μMALL . This will implies that the proofs we obtain are indeed isomorphisms. meaning that if we cut the aforementioned proofs π and π^\perp , performing the cut-elimination procedure would give either the identity on A or the identity on B .

The derivations we obtain are *circular*, and we therefore translate the isos directly into finite derivations with back-edges, written $\text{circ}(\omega)$ (defined in Definition 4.4.2). As mentioned in Section 3.4, we can define another translation into infinite derivations as the composition of $\text{circ}(-)$ with the unfolding: $\llbracket \omega \rrbracket = \mathcal{U}(\text{circ}(\omega))$. Intuitively, this corresponds to the infinite unfolding of the isos-variable f in an iso `fix` $f.\omega$.

Representing sequents. As mentioned in Section 3.4, there exists multiple validity criterion for the logic μMALL , for instance the one from [NST18] uses *lists* to represent sequents with a *threading function* that tells us how the formulas are distributed over the

inferences rules. Due to the way sequents are represented there is no need for formula occurrence, as in [Bae+20], and one simply works with formulas.

The difference between the two criterias lies in the fact that the one from [NST18] does not enjoy the cut-elimination procedure. Because we need to keep track of which formula is associated to which variable from the typing context, the translation uses a slightly modified version of μ MALL in which contexts are split in two parts, written $\Upsilon; \Theta$, where Υ is a list of formulas occurrence and Θ is a set of formula occurrences associated with a term-variable (written $x : F$). When starting the translation of an iso of type $A \leftrightarrow B$, we start in the context $[A_\alpha]; \emptyset$ (for some address α) and end in some context $[]; \Theta$. The additional information of the variable in Θ is here to make sure we know how to split the contexts accordingly when needed later during the translation, with respect to the way they are split in the typing derivation (for instance, in the case of a tensor). We write $\overline{\Theta} = \{F \mid x : F \in \Theta\}$ and $\underline{\Theta} = \{x : A \mid x : A_\alpha \in \Theta\}$. This modified system also make uses of another rule, called the *exchange rule* which allows us to send the first formula from Υ to Θ and affecting it a variable, defined as:

$$\frac{\Upsilon; x : F, \Theta \vdash G}{F :: \Upsilon; \Theta \vdash G} \text{ex}(x)$$

Given a derivation ι in this system, we write $\llbracket \iota \rrbracket$ for the function that sends ι into a derivation of μ MALL where (i) we remove all occurrences of the exchange rule (ii) the contexts $[]; \Theta$ become $\overline{\Theta}$. In this system the cut-elimination holds: one can simply ignore the lists, remove the exchange rule and the variable in the typing context Θ in order to fall back to the system from [Bae+20].

Translation. Given an iso $\omega : A \leftrightarrow B$ and initial addresses α, β , its translation into a derivation of μ MALL of $A_\alpha \vdash B_\beta$ is described with three separate phases:

Iso Phase. The first phase consists in travelling through the syntactical definition of an iso, keeping as information the last encountered iso-variable bounded by a **fix** and calling the negative phase when encountering an iso of the form $\{v_1 \leftrightarrow e_1 \mid \dots \mid v_n \leftrightarrow e_n\}$ and attaching to the formulas A and B two distinct addresses α and β and labelling the sequent with the name of the last encountered iso-variable. Later on, during the translation, this phase will be recalled when encountering another iso in one of the e_i , and, if that iso corresponds to an iso-variable, we will create a back-edge pointing towards the corresponding sequents.

Negative Phase. Starting from some context $[A_\alpha], \Theta$, the negative phase consists in decomposing the formula A according to the way the values of type A on the left-hand-side of ω are decomposed. The negative phase works as follows: we consider a set of pairs made of a list of values and a typing judgment, written (l, ξ) where each element of the set corresponds to one clause $v \leftrightarrow e$ of the given iso and ξ is the typing judgment of e . The list of values corresponds to what is left to be decomposed in the left-hand-side of the clause (for instance if v is a pair $\langle v_1, v_2 \rangle$ the list will have two elements to decompose).

Each element of the list Υ will correspond to exactly one value in the list l . If the term that needs to be decomposed is a variable x , then we will apply the $ex(x)$ rule, sending the formula to the context Θ . The negative phase ends when the list Υ is empty. When it is the case, we can start decomposing ξ and the *positive phase* starts. The negative phase is defined by case analysis on the first elements of the lists of the set, which are known by typing to have the same type constructor, and is given in Figure 4.1.

Positive phase. The translation of an expression is pretty straightforward: each **let** and iso-application is represented by two cut rules: as usual in Curry-Howard correspondence [SU06]. Keeping the variable-formula pair in the derivation is here to help us know how to split accordingly the context Θ when needed, while Υ is always empty and is therefore omitted. While the positive phase carries over the information of the last-seen iso-variable, it is not noted explicitly as it is only needed when calling the Iso Phase. The positive phase is given in Figure 4.2.

Remark 4.4.1. While μMALL is presented in a one-sided way, we write $\Sigma \vdash \Phi$ for $\vdash \Sigma^\perp, \Phi$ in order to stay closer to the formalism of the type system of isos and label the left rules on Σ as the corresponding right rules on Σ^\perp .

Definition 4.4.2. The translation $\text{circ}(\omega, S, \alpha, \beta) = \pi$ takes a well-typed iso, a singleton set S of an iso-variable corresponding to the last iso-variable seen in the induction definition of ω and two fresh addresses α, β and produces a circular derivation of the variant of μMALL described above with back-edges. $\text{circ}(\omega, S, \alpha, \beta)$ is defined inductively on ω :

- $\text{circ}(\mathbf{fix} f.\omega, S, \alpha, \beta) = \text{circ}(\omega, \{f\}, \alpha, \beta)$
- $\text{circ}(f, \{f\}, \alpha, \beta) = \overline{A_\alpha \vdash B_\beta} be(f)$
- $\text{circ}(\{(v_i \leftrightarrow e'_i)_{i \in I}\} : A \leftrightarrow B, \{f\}, \alpha, \beta) = \left\| \frac{\text{Neg}(\{(v_i, \xi_i)_{i \in I}\})}{A_\alpha \vdash^f B_\beta} \right\|$ where ξ_i is the typing derivation of e_i .

Example 4.4.3. The translation $\pi = \text{circ}(\omega, \emptyset, \alpha, \beta)$ of the iso ω from Example 4.2.5, with $F = A_{\alpha l}, G = B_{\alpha r l}, H = C_{\alpha r r}$ and $F' = A_{\beta r l}, G' = B_{\beta r r}, H' = C_{\beta l}$ is:

$$\frac{\frac{\frac{\overline{[]; a : F \vdash F'} id}{[]; a : F \vdash F' \oplus G'}{\oplus^1} \oplus^2}{[]; a : F \vdash H' \oplus (F' \oplus G')} ex(a)}{[F]; \emptyset \vdash H' \oplus (F' \oplus G')} \frac{\frac{\frac{\overline{[]; b : G \vdash G'} id}{[]; b : G \vdash F' \oplus G'}{\oplus^2} \oplus^2}{[]; b : G \vdash H' \oplus (F' \oplus G')} ex(b)}{[G]; \emptyset \vdash H' \oplus (F' \oplus G')} \frac{\frac{\frac{\overline{[]; c : H \vdash H'} id}{[]; c : H \vdash H' \oplus (F' \oplus G')}{\oplus^1} \oplus^1}{[H]; \emptyset \vdash H' \oplus (F' \oplus G')} ex(c)}{[G \oplus H]; \emptyset \vdash H' \oplus (F' \oplus G')} \wp$$

$$\frac{[F \oplus (G \oplus H)]; \emptyset \vdash H' \oplus (F' \oplus G')}{[F \oplus (G \oplus H)]; \emptyset \vdash H' \oplus (F' \oplus G')}$$

and its corresponding proof $\llbracket \pi \rrbracket$ in μMALL :

$$\begin{aligned}
 & \frac{\text{Neg}(\{(\text{inj}_l v_j :: l_j, \xi_j)_{j \in J}\} \cup \{(\text{inj}_r v_k :: l_k, \xi_k)_{k \in K}\})}{F_1 \oplus F_2 :: \Upsilon; \Theta \vdash G} = \\
 & \frac{\frac{\text{Neg}(\{(v_j :: l_j, \xi_j)_{j \in J}\})}{F_1 :: \Upsilon; \Theta \vdash G} \quad \frac{\text{Neg}(\{(v_k :: l_k, \xi_k)_{k \in K}\})}{F_2 :: \Upsilon; \Theta \vdash G}}{F_1 \oplus F_2 :: \Upsilon; \Theta \vdash G} \ \& \\
 & \frac{\text{Neg}(\{(\llbracket \cdot \rrbracket, \xi)\})}{\llbracket \cdot \rrbracket; \Theta \vdash G} = \frac{\text{Pos}(\xi)}{\llbracket \cdot \rrbracket; \Theta \vdash G} \quad \frac{\text{Neg}(\{(\cdot :: l, \xi)\})}{\mathbb{1} :: \Upsilon; \Theta \vdash G} = \frac{\text{Neg}(\{l, \xi\})}{\Upsilon; \Theta \vdash G} \top \\
 & \frac{\text{Neg}(\{(\langle v_i^1, v_i^2 \rangle :: l_i, \xi_i)_{i \in I}\})}{F_1 \otimes F_2 :: \Upsilon; \Theta \vdash G} = \frac{\text{Neg}(\{(v_i^1 :: v_i^2 :: l_i, \xi_i)_{i \in I}\})}{F_1, F_2 :: \Upsilon; \Theta \vdash G} \ \wp \\
 & \frac{\text{Neg}(\{(\text{fold } v_i :: l_i, \xi_i)_{i \in I}\})}{\mu X.F :: \Upsilon; \Theta \vdash G} = \frac{\text{Neg}(\{(v_i :: l_i, \xi_i)_{i \in I}\})}{F[X \leftarrow \mu X.F] :: \Upsilon; \Theta \vdash G} \ \nu \\
 & \frac{\text{Neg}(\{(x :: l, \xi)\})}{F :: \Upsilon; \Theta \vdash G} = \frac{\text{Neg}(\{l, \xi\})}{\Upsilon; \Theta, x : F \vdash G} \text{ex}(x)
 \end{aligned}$$

Figure 4.1.: Negative Phase

Proof. Since we have at most one iso-variable, we never end up in the case that between an annotated sequent \vdash^f and a back-edge pointing to f we encounter another annotated sequent. \square

Among the terms that we translate, the translation of a value yields what we call a *Purely Positive Proof*: a finite derivation whose only rules have for active formula the sole formula on the right of the sequent. Any such derivation is trivially a valid pre-proof.

Definition 4.4.6 (Purely Positive Proof). *A Purely Positive Proof is a finite, cut-free proof whose rules are only $\oplus^i, \otimes, \mu, \mathbb{1}, \text{id}$ for $i \in \{1, 2\}$.*

$$\begin{aligned}
 \text{Pos} \left(\frac{}{\vdash_e () : \mathbb{1}} \right) &= \overline{\llbracket; \emptyset \vdash \mathbb{1}_\alpha} \mathbb{1} \\
 \text{Pos} \left(\frac{}{x : A \vdash_e x : A} \right) &= \overline{\llbracket; x : A_\alpha \vdash A_\beta} \text{id} \\
 \text{Pos} \left(\frac{\frac{\xi}{\Theta \vdash_e t : A_1}}{\Theta \vdash_e \text{inj}_l t : A_1 \oplus A_2} \right) &= \overline{\llbracket; \Theta \vdash (A_1 \oplus A_2)_\alpha} \oplus^1 \\
 \text{Pos} \left(\frac{\frac{\xi}{\Theta \vdash_e t : A_2}}{\Theta \vdash_e \text{inj}_r t : A_1 \oplus A_2} \right) &= \overline{\llbracket; \Theta \vdash (A_2)_{\alpha r}} \oplus^2 \\
 \text{Pos} \left(\frac{\frac{\xi_1}{\Theta_1 \vdash_e t_1 : A_1} \quad \frac{\xi_2}{\Theta_2 \vdash_e t_2 : A_2}}{\Theta_1, \Theta_2 \vdash_e \langle t_1, t_2 \rangle : A_1 \otimes A_2} \right) &= \overline{\llbracket; \Theta_1 \vdash (A_1)_{\alpha l} \quad \llbracket; \Theta_2 \vdash (A_2)_{\alpha r}} \otimes \\
 \text{Pos} \left(\frac{\frac{\xi}{\Theta \vdash_e t : A[X \leftarrow \mu X.A]}}{\Theta \vdash_e \text{fold } t : \mu X.A} \right) &= \overline{\llbracket; \Theta \vdash (A[X \leftarrow \mu X.A])_{\alpha l}} \mu \\
 \text{Pos} \left(\frac{\frac{\xi_1}{\Theta_1 \vdash_e t_1 : A_1 \otimes \dots \otimes A_n} \quad \frac{\xi_2}{\Theta_2, x_1 : A_1, \dots, x_n : A_n \vdash_e t_2 : B}}{\Theta_1, \Theta_2 \vdash_e \text{let } (x_i)_{i \in I} = t_1 \text{ in } t_2 : B} \right) &= \\
 &= \overline{\llbracket; \Theta_1 \vdash F_1 \otimes \dots \otimes F_n \quad \overline{\text{Neg}(\langle (x_i)_{i \in I}, \xi_2 \rangle)}} \text{cut} \\
 &= \overline{\llbracket; \Theta_1, \Theta_2 \vdash B_\alpha} \\
 \text{Pos} \left(\frac{\frac{\Psi \vdash_\omega \omega : A \leftrightarrow B \quad \frac{\xi}{\Theta \vdash_e t : A}}{\Theta; \Psi \vdash_e \omega t : B}}{\Theta; \Psi \vdash_e \omega t : B} \right) &= \overline{\llbracket; \Theta \vdash A \quad \overline{\text{circ}(\omega, \{f\}, \alpha, \beta)}} \text{cut} \\
 &= \overline{\llbracket; \Theta \vdash B_\beta}
 \end{aligned}$$

Figure 4.2.: Positive Phase

Lemma 4.4.7 (Values are Purely Positive Proofs). *Given $x_1 : A^1, \dots, x_n : A^n \vdash_e v : A$ then $\overline{\llbracket; x_1 : A_{\alpha_1}^1, \dots, x_n : A_{\alpha_n}^n \vdash A_\alpha}$ is a purely positive proof.*

Proof. By induction on $\Delta \vdash_e v : A$

- $x : A \vdash_e x : A$ then the derivation is $\overline{\llbracket; F \vdash F} \text{id}$, which is a purely positive proof;
- $\vdash () : \mathbb{1}$ then the derivation is $\overline{\llbracket; \emptyset \vdash \mathbb{1}} \mathbb{1}$, which is a purely positive proof;
- $\frac{\pi_1}{\Delta_1 \vdash A} \quad \frac{\pi_2}{\Delta_2 \vdash B} \otimes$ and then by induction hypothesis on π_1 and π_2 ;

- $\underline{\Delta} \vdash \text{inj}_l v : A \oplus B$ then the derivation is $\frac{\frac{\pi}{\underline{\Delta} \vdash A}}{[]; \underline{\Delta} \vdash A \oplus B} \oplus^1$ then by induction hypothesis on π ;
- Similar for $\text{inj}_r v$ and $\text{fold } v$. □

We can then define the notion of *bouncing-cut* and their origin:

Definition 4.4.8 (Bouncing-Cut). *A Bouncing-cut is a cut of the form:*

$$\frac{\frac{\pi}{\Sigma \vdash G} \quad \frac{}{G \vdash F} \text{be}(f)}{\Sigma \vdash F} \text{cut}$$

Due to the syntactical restrictions of the language we get the following:

Property 4.4.9 (Origin of Bouncing-Cut). *Given a well-typed iso, every occurrence of a rule $\text{be}(f)$ in $\llbracket \text{circ}(\omega) \rrbracket$ is a premise of a bouncing-cut.*

In particular, when following a thread going up into a *bouncing-cut*, it will always start from the left-hand-side of the sequent, before going back down on the right-hand-side of the sequent. It will also always bounce back up on the bouncing-cut to reach the back-edge. Also note that when translating an iso, the derivation π in the bouncing-cut is a purely positive proof.

To make sure we follow the correct formula we define a notion of term occurrence and show that it matches the addresses obtained from the negative phases:

Definition 4.4.10 (Term Occurrence). *We note by $\text{Occ}(v)$ the set of Occurrence in the value v defined inductively on v by:*

- $\text{Occ}(v) = \{\epsilon\}$ if $v = x$ or $v = ()$
- $\text{Occ}(\langle v_1, v_2 \rangle) = \{\epsilon\} \cup l \cdot \text{Occ}(v_1) \cup r \cdot \text{Occ}(v_2)$
- $\text{Occ}(\text{inj}_l v) = \{\epsilon\} \cup l \cdot \text{Occ}(v)$
- $\text{Occ}(\text{inj}_r v) = \{\epsilon\} \cup r \cdot \text{Occ}(v)$
- $\text{Occ}(\text{fold } v) = \{\epsilon\} \cup i \cdot \text{Occ}(v)$

Where $x \cdot S = \{x\alpha \mid \alpha \in S\}$ for $x \in \{l, r, i\}$

Given $\alpha \in \text{Occ}(v)$ we write $v@_\alpha$ for the subterm of v at position α

We write $\xi(v) = \{\alpha \in \text{Occ}(v) \mid v@_\alpha = x\}$ for the set of position of variables in v and $\xi(x, v)$ for the position of x in v .

Theorem 4.4.11. *Given a sequence of sequents S_0, \dots, S_n , with $S_0 = A_\alpha \vdash B_\beta$ and $S_n = \Sigma \vdash B_\beta$ and the only rules applied are $\top, \&, \wp, \nu$.*

There exists a unique value v and context Δ such that $\Delta \vdash_e v : A$ and such that for all expressions e such that $\Delta \vdash_e e : B$, for all isos $\vdash_\omega \omega : A \leftrightarrow B$ such that $v \leftrightarrow e$ is a clause of ω , consider $\pi = \llbracket \text{circ}(\omega, \emptyset, \alpha, \beta) \rrbracket$ then S_0, \dots, S_n is a branch of π and for all formulas $A_\alpha \in \Sigma$, there exists a unique variable x such that $\xi(x, v)$ is a suffix of α .

Proof. By induction on n .

- Case 0, then take $\Delta = x : A$ and $v = x$, obviously $\Delta \vdash_e x : A$. We also get that $\omega = \{x \leftrightarrow e\}$ and $\llbracket \text{circ}(\omega) \rrbracket = \frac{\llbracket \text{Pos}(e) \rrbracket}{A_\alpha \vdash B_\beta}$ so the empty sequence is a branch and $\xi(v, v) = \epsilon$ which is a suffix of α .
- Case $n + 1$. By induction hypothesis, the sequence S_0, \dots, S_n with S_n sequent of $\Sigma^n \vdash B_\beta$ gives us $\Delta^n \vdash_e v^n : A$. Define the *values contexts* as $\mathcal{V} = [] \mid \langle \mathcal{V}, v \rangle \mid \langle v, \mathcal{V} \rangle \mid \text{inj}_l \mathcal{V} \mid \text{inj}_r \mathcal{V} \mid \text{fold } \mathcal{V}$.

Then, by case analysis on the rule of S_{n+1} .

- \wp : we can write Σ^n as $A_{\alpha_1}^1, \dots, (C_1 \otimes C_2)_{\alpha_k}^k, \dots, A_{\alpha_n}^n \vdash B_\beta$ and we know that $\Delta^n = x_1 : A^1, \dots, x_k : C_1 \otimes C_2, \dots, x_n : A^n$ then v can be written as $\mathcal{V}[x_k]$.

Build $v^{n+1} = \mathcal{V}[\langle y, z \rangle]$ and $\Delta^{n+1} = \Delta \setminus \{x_k : C_1 \otimes C_2\} \cup \{y : C_1, z : C_2\}$.

We get that $\Delta^{n+1} \vdash_e v^{n+1}$, then for any iso ω such that $\mathcal{V}[x_k] \leftrightarrow e$ is a clause, we can replace the clause by $\mathcal{V}[\langle y, z \rangle] \leftrightarrow e[x \leftarrow \langle y, z \rangle]$ in order to build ω' , and if S_0, \dots, S_n was a branch in $\llbracket \text{circ}(\omega) \rrbracket$ then so is S_0, \dots, S_n, S_{n+1} in ω' .

We know that $\xi(x, v) = \gamma$ is a suffix of α_k , then after applying the \wp rule we have that C_1 has address $\alpha_k l$ and C_2 has address $\alpha_k r$. Therefore, $\xi(y, v^{n+1}) = \gamma l$ and $\xi(z, v^{n+1}) = \gamma r$ which are respectively suffixes of $\alpha_k l$ and $\alpha_k r$.

- $\&$. Assuming that S_{n+1} goes to the left branch of the $\&$ rule.

We then have $\Sigma^n = A_{\alpha_1}^1, \dots, (C_1 \oplus C_2)_{\alpha_k}^k, \dots, A_{\alpha_n}^n \vdash B_\beta$ and $\Delta^n = x_1 : A^1, \dots, x_k : C_1 \oplus C_2, \dots, x_n : A^n$ with $v = \mathcal{V}[x_k]$.

Consider $v^{n+1} = \mathcal{V}[\text{inj}_l y]$ and Δ^{n+1} .

For any iso ω where $v^n \leftrightarrow e$ was a clause, we can consider the isos ω' where the clause $v^n \leftrightarrow e$ has been replaced by two clauses $\mathcal{V}[\text{inj}_l y]$ and $\mathcal{V}[\text{inj}_r r]$ with $e[x \leftarrow y]$ and $e[x \leftarrow z]$. S_0, \dots, S_n, S_{n+1} is obviously a branch in $\llbracket \text{circ}(\omega) \rrbracket$ by definition of the negative phase.

Also since $\xi(x, v) = \gamma$ is a suffix of α_k , after applying the $\&$ rule, on the left branch we get C_1 with address $\alpha_k l$. And $\xi(y, v^{n+1}) = \gamma l$ is a suffix of $\alpha_k l$.

- The other side of the $\&$ is similar.

- The ν rule is similar.
- \top . In which case we have $\Sigma^n = \mathbb{1}_\alpha, A_{\alpha_1}^1 \dots, A_{\alpha_n}^n$ with $\Delta^n = x : \mathbb{1}, x_1 : A^1, \dots, x_n : A^n$ with $v^n = \mathcal{V}[x]$, build v^{n+1} as $\mathcal{V}[\langle \rangle]$ and $\Delta^{n+1} = x_1 : A^1, \dots, x_n : A^n$. Then after the \top rule we get $\Gamma^{n+1} = A_{\alpha_1}^1, \dots, A_{\alpha_n}^n$ so the property holds by our induction hypothesis. \square

We can now show that our translation is well-defined:

Lemma 4.4.12. *Given a closed iso $\vdash_\omega \omega$ then $\text{circ}(\omega, \emptyset, \alpha, \beta)$ for α, β fresh addresses, is well-defined.*

Proof. By induction on \vdash_ω . The iso is of the form $\mathbf{fix} f_1 \dots, \mathbf{fix} f_n. \{v_1 \leftrightarrow e_1 \mid \dots \mid v_n \leftrightarrow e_n\}$ and as only the last iso-variable is kept, we end up in the case $\text{circ}(\{(v_i \leftrightarrow e'_i)_{i \in I}\} : A \leftrightarrow B, \{f_n\}, \alpha, \beta)$, where the root of the derivation take place, annotated with f_n .

The negative phase $\mathbf{Neg}(\{(v_i, \xi'_i)_{i \in I}\})$ is well-defined due to the predicate \mathbf{OD}_A : by definition of \mathbf{OD}_A all left-hand-side values have the same type constructors, they also have the same type.

The positive phase is well-defined as it consists in recreating the typing judgment of each expression e_i . By Theorem 4.4.11 we know that the typing context of e_i and the one obtained from the negative phase contain the same variable associated to the same formula / formula occurrence. \square

Given an iso $\omega = \mathbf{fix} f. \{v_1 \leftrightarrow e_1 \mid \dots \mid v_n \leftrightarrow e_n\}$, we want to show that for any infinite branch there exists a valid thread that inhabits it. As given by Lemma 4.4.5, an infinite branch is uniquely defined by a single iso-variable.

Given the value v_i^j that is the decreasing argument for the structurally recursive criterion, we want to build a pre-thread that follows the variable $x_j : \mu X.B$ in $v_i^j : \mu X.B$ that is the focus of the criterion.

Since our sequents are different from the one of μMALL during the negative phase, we are forced to define properly the way the pre-thread is built. For the positive phase, our sequents and the one of μMALL are the same, aside from the fact that the context also contains variables, so for this part we can just use Definition 3.4.5.

Definition 4.4.13 (Pre-Thread of the negative phase). *Given a well-typed iso $\omega = \mathbf{fix} f. \{v_1 \leftrightarrow e_1 \mid \dots \mid v_n \leftrightarrow e_n\} : A \leftrightarrow B$ and a clause $v_i \leftrightarrow e_i$ such that $f p$ is a subterm of e_i and the variable x^p is the focus of the primitive recursive criterion, and considering $\pi = \text{circ}(\omega)$, we define $PT_n(x^p, \pi)$ as the pre-thread that follow the formula $\mu X.A'$ corresponding to x^p up to the positive phase by induction on $\mathbf{Neg}(\{(v_i, e_i)_{i \in I}\})$. For simplicity, we simply omit the first argument of PT_n .*

- $PT_n(\mathbf{Neg}(\{(\langle \rangle, e)\}))$ is impossible as we follow a variable;

- $PT_n(\text{Neg}(\{(((() :: l, e))\})) = PT_n(\text{Neg}(\{(l, e)\}));$
- $PT_n(\text{Neg}(\{((y :: l, e))\})) = \begin{cases} \epsilon & \text{if } y = x^p \\ PT_n(\text{Neg}(\{(l, e)\})) & \text{otherwise;} \end{cases}$
- $PT_n(\text{Neg}(\{(\text{inj}_l v_i :: l_i, e_i)_{i \in I}\} \cup \{(\text{inj}_l v_j :: l_j, e_j)_{j \in J}\}))$

$$= \begin{cases} (A \oplus C; A \oplus C, \Delta \vdash B) \cdot (A; A, \Delta \vdash B) \cdot PT_n(\text{Neg}(\{(v_i :: l_i, e_i)_{i \in I}\})) & \text{if } x^p \subseteq v_i \\ (C \oplus A; C \oplus A, \Delta \vdash B) \cdot (A; A, \Delta \vdash B) \cdot PT_n(\text{Neg}(\{(v_j :: l_j, e_j)_{j \in J}\})) & \text{if } x^p \subseteq v_j \\ (\mu X.D; A_1 \oplus A_2, \Delta \vdash B) \cdot (\mu X.D, A_k, \Delta \vdash B) \cdot PT_n(\text{Neg}(\{(v_l :: l_l, e_l)_{l \in L}\})) & \text{for } L \in \{I, J\}, k \in \{1, 2\} \text{ and if } x^p \subseteq l_L \end{cases}$$
- Case $PT_n(\text{Neg}(\{(v_i^1, v_i^2 :: l_i, e_i)_{i \in I}\}))$

$$= \begin{cases} (A_1 \otimes A_2; A_1 \otimes A_2, \Delta \vdash B) \cdot (A_k; A_1, A_2, \Delta \vdash B) \cdot PT_n(\text{Neg}(\{(v_i^1, v_i^2 :: l_i, e_i)_{i \in I}\})) & \text{for } k \in \{1, 2\} \text{ if } x^p \subseteq v_k \\ (\mu X.D; A_1 \otimes A_2, \Delta \vdash B) \cdot (\mu X.D; A_1, A_2, \Delta \vdash B) \cdot PT_n(\text{Neg}(\{(v_i^1, v_i^2 :: l_i, e_i)_{i \in I}\})) & \text{if } x^p \subseteq l \end{cases}$$
- $PT_n(\text{Neg}(\{(\text{fold } v_i :: l_i, e_i)_{i \in I}\}))$

$$= \begin{cases} (\mu X.A; \mu X.A, \Delta \vdash B) \cdot (A[X \leftarrow \mu X.A]; A[X \leftarrow \mu X.A], \Delta \vdash B) & \text{if } x^p \subseteq v_i \\ (\mu X.D; \mu X.A, \Delta \vdash B) \cdot (\mu X.D; A[X \leftarrow \mu X.A], \Delta \vdash B) & \text{if } x^p \subseteq l_i \end{cases}$$

Lemma 4.4.14 (Weight of the pre-thread for the negative phase). *Given a well-typed iso $\omega = \mathbf{fix} f.\omega$ with a clause $v \leftrightarrow e$, then $w(PT_n(\text{Neg}(\{([v], e)\})))$ is a word over $\{l, r, i, W\}$.*

Proof. By case analysis of Definition 4.4.13:

- If the variable x^p is not a subterm of the first value from the list l then the thread has the form: $(A; C, \Delta \vdash B, \uparrow) \cdot (A; C', \Delta \vdash B, \uparrow)$ and the weight is W .
- If the variable x^p is a subterm of the first value of the list l then by direct case analysis on the first value: if the value is of form $\langle v_1, v_2 \rangle$ then depending on whether x^p is in v_1 or v_2 the weight will be l or r , similarly for the inj_l and inj_r , while the fold will create weight i . \square

A similar analysis can be done for the positive phase:

Definition 4.4.15 (Pre-thread of the positive phase). *Given a well-typed iso $\omega = \mathbf{fix} f.\{v_1 \leftrightarrow e_1 \mid \dots \mid v_n \leftrightarrow e_n\} : A \leftrightarrow B$ and a clause $v_i \leftrightarrow e_i$ such that $f p$ is a subterm of e_i and the variable x^p is the focus of the primitive recursive criterion, and considering $\pi = \text{circ}(\omega, S, A_\alpha, B_\beta)$, then we define $PT_p(x^p, \text{Pos}(e_i))$ as the pre-thread that follow the formula $\mu X.A'$ corresponding to x^p until the sequent $A_{\alpha'} \vdash^f B_{\beta'}$ is reached. For simplicity we also just write it as $PT_p(x^p)$.*

Lemma 4.4.16. *Shape of the pre-thread of the positive phase* Given a well-typed iso ω such that the variable x^p is the focus of the primitive recursive criterion, and two successive element a, b from the pre-thread $PT(x^p)$ from the translation of ω , then $a \cdot b$ are of either:

- $(\nu X.F; s; \uparrow) \cdot (\nu X.F; s'; \uparrow);$
- $((\nu X.A)_\alpha; (\nu X.A)_\alpha \vdash (\nu X.A)_\beta; \uparrow) \cdot ((\nu X.A)_\beta; (\nu X.A)_\alpha \vdash (\nu X.A)_\beta; \downarrow);$
- $(F; s; \downarrow) \cdot (F'; s'; \downarrow);$
- $(F; s; \downarrow) \cdot (F; s'; \uparrow)$

Proof. By a straightforward case analysis on the positive phase:

- If a right-rule is applied while the pre-thread is on the left side of the sequent, then we are in the first case as the thread just goes up while following the formula.
- The second case occurs when encountering an axiom rule on the variable x^p that we follow.
- The third case is when going down on a purely positive proof. As we follow the sole formula on the right, the formula necessarily changes and s, s' are of the shape $\Delta \vdash s, \Delta' \vdash s'$.
- The last one is when encountering a cut, at the root of the purely positive proof that comes from the translation of a `let`.

□

We can now look at the weight of the positive phase:

Lemma 4.4.17 (Weight of the pre-Thread for the positive phase). *Given a well-typed iso such that the variable x^p is the focus of the primitive recursive criterion, the weight of the pre-thread on the positive phase $PT_p(x^p)$ is of the shape $\mathcal{W}^* \mathcal{A} \{ \bar{l}, \bar{r}, \mathcal{W} \}^* \mathcal{C}$*

Proof. By case analysis on $\text{Pos}(e)$. As the thread only goes up by encountering cut-rules or right-rules, we get \mathcal{W}^* , and the thread goes up all the way to an axiom rule, corresponding to the formula $x^p : \nu X.F$, which adds the \mathcal{A} . Finally, the thread goes down on the purely positive proof, generating $\{ \bar{l}, \bar{r}, \mathcal{W} \}^*$ until reaching the cut-rule from the bouncing cut. □

We can then consider the infinite pre-thread as the concatenation of both the pre-thread of the negative phase, and the one of the positive phase.

Lemma 4.4.18 (Form of the Pre-Thread). *Given the pre-thread t following x^p we have that $w(t) = p_0(\sum_{i \leq n} p_i \mathcal{W}_i^* \mathcal{A} q_i \mathcal{C})^\omega$ with*

- p_0 is any prefix.
- $p_i \in \{l, r, i, \mathcal{W}\}^*$
- $q_i \in \{\bar{l}, \bar{r}, \mathcal{W}\}^*$

With, $\forall i \leq n, \bar{q}_i \sqsubset \bar{p}_i$ and $|p_i| > |q_i|$ without counting the \mathcal{W} , where $p \sqsubset q$ is q is a prefix of p and with $\bar{x}\bar{p} = \bar{x}\bar{p}$ if $x \in \{l, r, i\}$, $\bar{x}\bar{p} = \bar{x}\bar{p}$ if $x \in \{l, r, i\}$ and $\overline{\mathcal{W}p} = \bar{p}$

Proof. p_i is generated from Definition 4.4.13 while the rest up to the C (included) is generated from Definition 4.4.15.

First, we show that $|p_i| > |q_i|$ modulo the \mathcal{W} .

Since p_i is generated by the negative phase, we have that, modulo \mathcal{W} , $p_i = \{r\}^* l^+ \{l, r, i\}^*$, this is due by definition of being primitive recursive and because we are looking for the right variable. By definition of being primitive recursive the input type of the iso is $A_1 \otimes \cdots \otimes A_n$, hence $\{r\}^* l^+$ correspond to the search for the correct A_i for the input type $A_1 \otimes \cdots \otimes A_n$ while $\{l, r, i\}^*$ is the decomposition of the primitive recursive value, as described in Theorem 4.4.11.

As q_i corresponds to the Purely Positive Proof, we know that the Purely Positive Proof is the encoding of a pattern $p = \langle x_1, \langle \dots, x_n \rangle \rangle$. Hence, q_i can be decomposed as $\{\bar{l}^+ \bar{r}^*\}$

By the fact that the iso is primitive recursive we know that the variable in p is a strict subterm of the primitive recursive value, hence $|p_i| > |q_i|$.

The fact that $\bar{q}_i \sqsubset \bar{p}_i$ is direct as the Purely Positive Proof reconstructs the type $A_1 \otimes \cdots \otimes A_n$ without modifying the A_i while \bar{p}_i start by searching for the corresponding type A_i , so it is only composed of $\{l, r\}^*$, which will be the same as \bar{q}_i . \square

Theorem 4.4.19 (The Pre-thread generated is a thread). *We want to find a decomposition of the pre-thread such that it can uniquely be decomposed into $\odot(H_i \odot V_i)$ with*

- $w(V_i) \in \{l, r, i, \mathcal{W}\}^\infty$ and non-empty if $i \neq \lambda$
- $w(H_i) \in \mathcal{H}$

Proof. We set H_0 as the empty pre-thread. (so $w(H_0) = \epsilon$) We set V_0 as the maximal possible sequence such that $w(V_0) \in \{l, r, i, \mathcal{W}\}^*$, i.e the sequence that ends with $(A; A \vdash A; \uparrow)$. Then, for all $i \geq 1$ we set

- H_i starts at $(A; A \vdash A; \uparrow)$ just before the axiom rule so that the first element of $w(H_i)$ is A . Then H_i is composed of
 - All of the pre-thread going down on the Purely Positive Proof after the axiom, accumulating a word over $\{\bar{l}, \bar{r}, \bar{i}, \mathcal{W}\}^*$;
 - Going back up into the cut-rule of the bouncing cut, making a \mathcal{C} ;

- Going up to compensate every \bar{x} seen in the Purely Positive Proof while going down. This is possible as shown in Lemma 4.4.18.
- V_i is the maximal possible sequence such that $w(V_i) \in \{l, r, i, \mathcal{W}\}^*$, i.e the sequence that ends with $(A; A \vdash A; \uparrow)$. \square

Theorem 4.4.20 (*Validity of proofs*). *If $\vdash_\omega \omega : A \leftrightarrow B$ and $\pi = \llbracket \text{circ}(\omega, \emptyset, \alpha, \beta) \rrbracket$ then π satisfies μMALL validity criterion from Chapter 3.*

Proof. By Theorem 4.4.19 we know that we have a thread of which we also know by Theorem 4.4.19 that the visible part is not stationary.

Finally, by Lemma 4.4.18 and Theorem 4.4.19 we know that the visible part will see infinitely often the subformulas of the formula $\mu X.B$ that is the focus of the primitive recursive criterion. This is due to the difference in size in the part of the thread from the negative and from the positive phase and the fact that the positive phase does not encounter a μ formula when going down on a purely positive proof.

By the constraints on the syntax of our isos, all the possible slices are necessarily persistent.

Therefore, the smallest formula we will encounter is nu formula, validating the thread. \square

4.4.3. Proof Simulation

In order to show the relationship between the cut-elimination procedure of μMALL and the rewriting system of our language we introduce a slightly modified version of the rewriting system, that we call $\rightarrow_{e\beta}$, using explicit substitution.

Convention 4.4.21. *Given a substitution $\sigma = \{x_1 \mapsto v_1, \dots, x_n \mapsto v_n\}$ we will use the shorthand $\text{let } \sigma \text{ in } t$ for $\text{let } x_1 = v_1 \text{ in } \dots \text{let } x_n = v_n \text{ in } t$.*

Definition 4.4.22 (*Explicit Substitution Rewriting System*). $\rightarrow_{e\beta}$ is defined by the following rules:

$$\begin{array}{l}
 \text{let } x = v \text{ in } x \rightarrow_{e\text{let}} v \\
 \text{let } \langle x_1, p \rangle = \langle t_1, t_2 \rangle \text{ in } t \rightarrow_{e\text{let}} \text{let } x_1 = t_1 \text{ in let } p = t_2 \text{ in } t \\
 \text{let } x = v \text{ in } \langle t_1, t_2 \rangle \rightarrow_{e\text{let}} \langle \text{let } x = v \text{ in } t_1, t_2 \rangle \quad \text{when } x \in FV(t_1) \\
 \text{let } x = v \text{ in } \langle t_1, t_2 \rangle \rightarrow_{e\text{let}} \langle t_1, \text{let } x = v \text{ in } t_2 \rangle \quad \text{when } x \in FV(t_2) \\
 \text{let } x = v \text{ in } \text{inj}_l t \rightarrow_{e\text{let}} \text{inj}_l \text{let } x = v \text{ in } t \\
 \text{let } x = v \text{ in } \text{inj}_r t \rightarrow_{e\text{let}} \text{inj}_r \text{let } x = v \text{ in } t \\
 \text{let } x = v \text{ in } \text{fold } t \rightarrow_{e\text{let}} \text{fold let } x = v \text{ in } t
 \end{array}$$

$$\text{let } x = v \text{ in } \omega \ t \rightarrow_{\text{elet}} \omega \ \text{let } x = v \text{ in } t$$

and the following rules:

$$\frac{t \rightarrow_{e\beta} \cup \rightarrow_{\text{elet}} t'}{C[t] \rightarrow_{e\beta} C[t']} \beta - \text{Cong} \quad \frac{\sigma[p] = v}{\text{let } p = v \text{ in } t \rightarrow_{e\beta} \text{let } \sigma \text{ in } t} \beta - \text{LetE}$$

$$\frac{}{(\mathbf{fix} \ f.\omega) \rightarrow_{e\beta} \omega[f \leftarrow (\mathbf{fix} \ f.\omega)]} \beta - \text{IsoRec}$$

$$\frac{\sigma[v_i] = v'}{\{v_1 \leftrightarrow e_1 \mid \dots \mid v_n \leftrightarrow e_n\} v' \rightarrow_{e\beta} \text{let } \sigma \text{ in } e_i} \beta - \text{IsoApp}$$

Remark 4.4.23. The rule $\beta - \text{LetE}$ is superfluous as it can be inferred from the decomposition of the rule of the decomposition of a `let`.

Each step of this rewriting system will correspond to exactly one step of cut-elimination, while in the previous system, the rewriting that uses a substitution σ correspond in fact to multiple steps of cut-elimination. $\rightarrow_{e\beta}$ makes this explicit. But first, we need to make sure that both systems do the same thing:

Lemma 4.4.24 (Specialisation of the substitution on pairs). *Let σ be a substitution that closes $\Delta \vdash_e \langle t_1, t_2 \rangle$, then there exists σ_1, σ_2 , such that $\sigma(\langle t_1, t_2 \rangle) = \langle \sigma_1(t_1), \sigma_2(t_2) \rangle$ Where $\sigma = \sigma_1 \cup \sigma_2$.*

Proof. By the linearity of the typing system we know that $FV(t_1) \cup FV(t_2) = \emptyset$, so there always exists a decomposition of σ into σ_1, σ_2 defined as $\sigma_i = \{(x_i \mapsto v_i) \mid x_i \in FV(t_i)\}$ for $i \in \{1, 2\}$. \square

Lemma 4.4.25 (Explicit substitution and substitution coincide). *Let $\sigma = \{x_i \mapsto v_i\}$ be a substitution that closes t , then $\text{let } \sigma \text{ in } t \rightarrow_{\text{elet}}^* \sigma(t)$.*

Proof. By induction on t .

- x , then $\sigma(x) = v$ and $\text{let } x = v \text{ in } x \rightarrow_{\text{elet}} v = \sigma(x)$.
- $()$ then σ is empty and no substitution applies.
- $\langle t_1, t_2 \rangle$, then by Lemma 4.4.24 $\sigma(\langle t_1, t_2 \rangle) = \langle \sigma_1(t_1), \sigma_2(t_2) \rangle$. By $\rightarrow_{\text{elet}}$, each `let` construction will enter either t_1 or t_2 , then by induction hypothesis.
- $\text{let } p = t_1 \text{ in } t_2$ is similar to the product case.
- $\text{inj}_l \ t, \text{inj}_r \ t, \text{fold } t, \omega \ t$. All cases are treated in the same way: by definition of $\rightarrow_{\text{elet}}$, each `let` will enter into the subterm t , as with the substitution σ , then by induction hypothesis. \square

Corollary 4.4.26. *If $t \rightarrow t'$ then $t \rightarrow_{e\beta}^* t'$.*

Proof. By induction on \rightarrow , the case of IsoRec is the same, while for the other rules, just by applying either β – IsoApp or β – LetE and then by Lemma 4.4.25, let σ in $t \rightarrow_{e\text{let}}^* \sigma(t)$ for any substitution σ that closes t . But $\sigma(t) = t'$, so $t \rightarrow_{e\beta}^* t'$. \square

It is then possible to show a first step of the simulation procedure: that $\rightarrow_{e\beta}$ corresponds to one step of cut-elimination:

Lemma 4.4.27 (Simulation of the let-rules of $\rightarrow_{e\beta}$). *Let $\Theta \vdash_e t : G$ be a well-typed closed term: if $t \rightarrow_{e\text{let}} t'$ then, given some fresh address β we get: $\frac{\llbracket \text{Pos}(t) \rrbracket}{\Theta \vdash G_\beta} \rightsquigarrow \frac{\llbracket \text{Pos}(t') \rrbracket}{\Theta \vdash G_\beta}$.*

Proof. By case analysis on $\rightarrow_{e\text{let}}$, for simplicity of reading we omit the address β .

- let $x = v$ in $x \rightarrow_{e\text{let}} v$.

$$\frac{\frac{\llbracket \text{Pos}(v) \rrbracket}{\Theta \vdash G} \quad \overline{G \vdash G} \text{ id}}{\Theta \vdash G} \text{ cut} \rightsquigarrow \frac{\llbracket \text{Pos}(v) \rrbracket}{\Theta \vdash G}$$

- let $\langle x_1, p \rangle = \langle t_1, t_2 \rangle$ in $t \rightarrow_{e\text{let}} \text{let } x_1 = t_1 \text{ in let } p = t_2 \text{ in } t$

$$\begin{aligned} & \frac{\frac{\frac{\llbracket \text{Pos}(t_1) \rrbracket}{\Theta_1 \vdash G} \quad \frac{\llbracket \text{Pos}(t_2) \rrbracket}{\Theta_2 \vdash F}}{\Theta_1, \Theta_2 \vdash G \otimes F} \otimes \quad \frac{\llbracket \text{Neg}(\{[p], t\}) \rrbracket}{\Theta_3, G, F \vdash H}}{\Theta_3, G \otimes F \vdash H} \wp}{\Theta_1, \Theta_2, \Theta_3 \vdash H} \text{ cut} \\ \text{Then:} & \\ & \rightsquigarrow \frac{\frac{\llbracket \text{Pos}(t_1) \rrbracket}{\Theta_1 \vdash G} \quad \frac{\frac{\llbracket \text{Pos}(t_2) \rrbracket}{\Theta_2 \vdash F} \quad \frac{\llbracket \text{Neg}(\{[p], t\}) \rrbracket}{\Theta_3, G, F \vdash H}}{\Theta_2, \Theta_3, G \vdash H} \text{ cut}}{\Theta_1, \Theta_2, \Theta_3 \vdash H} \text{ cut} \end{aligned}$$

- let $x = v$ in $\text{inj}_l t \rightarrow_{e\text{let}} \text{inj}_l \text{let } x = v \text{ in } t$.

$$\begin{aligned} & \frac{\frac{\frac{\llbracket \text{Pos}(v) \rrbracket}{\Theta_1 \vdash F} \quad \frac{\frac{\llbracket \text{Pos}(t) \rrbracket}{F, \Theta \vdash H}}{F, \Theta_2 \vdash H \oplus G} \oplus_R^1}}{\Theta_1, \Theta_2 \vdash H \oplus G} \text{ cut}}{\Theta_1, \Theta_2 \vdash H \oplus G} \oplus_R^1 \\ \text{Then:} & \\ & \rightsquigarrow \frac{\frac{\llbracket \text{Pos}(v) \rrbracket}{\Theta_1 \vdash F} \quad \frac{\llbracket \text{Pos}(t) \rrbracket}{F, \Theta_2 \vdash H}}{\Theta_1, \Theta_2 \vdash H} \text{ cut}}{\Theta_1, \Theta_2 \vdash H \oplus G} \oplus_R^1 \end{aligned}$$

- The same goes for $\text{let } x = v \text{ in inj}_r t \rightarrow_{\text{e1et}} \text{inj}_r \text{let } x = v \text{ in } t$ and $\text{let } x = v \text{ in fold } t \rightarrow_{\text{e1et}} \text{fold let } x = v \text{ in } t$

- $\text{let } x = v \text{ in } \langle t_1, t_2 \rangle \rightarrow_{\text{e1et}} \langle \text{let } x = v \text{ in } t_1, t_2 \rangle$ when $x \in FV(t_1)$

Then:

$$\begin{aligned} & \frac{\frac{\frac{\llbracket \text{Pos}(v) \rrbracket}{\Theta_1 \vdash F}}{\Theta_1, \Theta_2, \Theta_3 \vdash H \otimes G} \quad \frac{\frac{\frac{\llbracket \text{Pos}(t_1) \rrbracket}{F, \Theta_2 \vdash H} \quad \frac{\llbracket \text{Pos}(t_2) \rrbracket}{\Theta_3 \vdash G}}{F, \Theta_2, \Theta_3 \vdash H \otimes G} \otimes_R}{\Theta_1, \Theta_2, \Theta_3 \vdash H \otimes G} \text{cut}}{\Theta_1, \Theta_2, \Theta_3 \vdash H \otimes G} \text{cut}} \\ & \rightsquigarrow \frac{\frac{\frac{\llbracket \text{Pos}(v) \rrbracket}{\Theta_1 \vdash F} \quad \frac{\llbracket \text{Pos}(t_1) \rrbracket}{F, \Theta_2 \vdash H}}{\Theta_1, \Theta_2 \vdash H} \text{cut} \quad \frac{\llbracket \text{Pos}(t_2) \rrbracket}{\Theta_3 \vdash G}}{\Theta_1, \Theta_2, \Theta_3 \vdash H \otimes G} \otimes_R \end{aligned}$$

- Similar for the second rule on the pair.
- $\text{let } x = v \text{ in } \omega t \rightarrow_{\text{e1et}} \omega (\text{let } x = v \text{ in } t)$

$$\begin{aligned} \text{Then: } & \frac{\frac{\frac{\llbracket \text{Pos}(v) \rrbracket}{\Theta_1 \vdash F} \quad \frac{\frac{\llbracket \text{Pos}(t) \rrbracket}{F, \Theta_2 \vdash G} \quad \frac{\llbracket \text{circ}(\omega) \rrbracket}{G \vdash H}}{F, \Theta_2 \vdash H} \text{cut}}{\Theta_1, \Theta_2 \vdash H} \text{cut}}{\Theta_1, \Theta_2 \vdash H} \text{cut}} \\ & \rightsquigarrow \frac{\frac{\frac{\llbracket \text{Pos}(v) \rrbracket}{\Theta_1 \vdash F} \quad \frac{\llbracket \text{Pos}(t) \rrbracket}{F, \Theta_2 \vdash G}}{\Theta_1, \Theta_2 \vdash G} \text{cut} \quad \frac{\llbracket \text{circ}(\omega) \rrbracket}{G \vdash H}}{\Theta_1, \Theta_2 \vdash H} \text{cut} \end{aligned} \quad \square$$

We then show that the pattern-matching is captured by the cut-elimination:

Lemma 4.4.28. *Let $\Gamma \vdash_e v : A$ and $\Delta \vdash_e v' : A$ such that $\sigma[v] = v'$ and $\sigma = \{\vec{x}_j \mapsto \vec{w}_j\}$ then for any list $l = [v_1, \dots, v_n]$ where for all $i \in \{0, \dots, n\}$ there exists Γ_i such that $(\Gamma_i \vdash_e v_i)$ and such that $\text{OD}_A(\{v, v_1, \dots, v_n\})$ and for all $i \in \{0, \dots, n\}$ and e_1, \dots, e_n such that $(\Gamma_i \vdash_e e_i : B)$ and such that $\text{OD}_B(\{\text{Val}(e_1), \dots, \text{Val}(e_n)\})$ and given $\Theta = \{x : A_\alpha \mid x : A \in \Delta\}$ and $G = B_\beta$ we have:*

$$\pi = \frac{\frac{\frac{\llbracket \text{Pos}(v') \rrbracket}{\Theta \vdash H} \quad \frac{\llbracket \text{Neg}(\{(v :: l)_i, e_i\}_{i \in I}) \rrbracket}{H \vdash G}}{\Theta \vdash G} \text{cut}}{\Theta \vdash G} \rightsquigarrow^* \frac{\llbracket \text{Neg}(\{l, \text{let } \vec{x}_i = \vec{w}_i \text{ in } e\}) \rrbracket}{\Theta \vdash G} = \pi'$$

Proof. By induction on $\text{OD}_A(\{v, v_1, \dots, v_n\})$:

- Case $\text{OD}_A(\{x\})$ we get $\sigma[x] = v$ then $\pi = \pi'$.

- Case $\text{OD}_{\mathbb{1}}(\{\mathbb{1}\})$ then $\sigma[()] = ()$

$$\pi = \frac{\frac{\overline{\vdash \mathbb{1}} \mathbb{1}_R \quad \frac{\frac{\llbracket \text{Pos}(e) \rrbracket}{\vdash G} \mathbb{1}_L}{\mathbb{1} \vdash G} \text{cut}}{\vdash G}}{\vdash G} \text{cut} \text{ which reduces to } \frac{\llbracket \text{Pos}(e) \rrbracket}{\vdash G} = \pi' \text{ as } \sigma \text{ is empty.}$$

- Case $\text{OD}_{\mu X.A}(\{\text{fold } v_i\})$ such that $\sigma[\text{fold } v_j] = \text{fold } v'$

$$\text{Then } \pi = \frac{\frac{\frac{\llbracket \text{Pos}(v') \rrbracket}{\Theta \vdash H[X \leftarrow \mu X.H]} \mu_R \quad \frac{\frac{\llbracket \text{Neg}(\{[v_i], e_i\}) \rrbracket}{H[X \leftarrow \mu X.H] \vdash G} \mu_L}{\mu X.H \vdash G} \text{cut}}{\Theta \vdash \mu X.H}}{\Theta \vdash G} \text{cut}$$

will reduce to

$$\frac{\frac{\llbracket \text{Pos}(v') \rrbracket}{\Theta \vdash H[X \leftarrow \mu X.H]} \quad \frac{\llbracket \text{Neg}(\{[v_i], e_i\}) \rrbracket}{H[X \leftarrow \mu X.H] \vdash G}}{\Theta \vdash G} \text{cut}$$

then we can apply our induction hypothesis.

- Case $\text{OD}_{A \oplus B}(\{\text{inj}_l v_i\} \cup \{\text{inj}_r v_k\})$ with $\sigma[\text{inj}_l v_j] = \text{inj}_l v'$

$$\text{Then the proof } \pi = \frac{\frac{\frac{\llbracket \text{Pos}(v') \rrbracket}{\Theta \vdash H}}{\Theta \vdash H \oplus F} \oplus_R \quad \frac{\frac{\llbracket \text{Neg}(\{[v_i], e_i\}) \rrbracket}{H \vdash G} \quad \frac{\llbracket \text{Neg}(\{[v_k], e_k\}) \rrbracket}{F \vdash G}}{H \oplus F \vdash G} \text{cut}}{\Theta \vdash G} \oplus_L$$

$$\text{reduces to } \frac{\frac{\llbracket \text{Pos}(v') \rrbracket}{\Theta \vdash H} \quad \frac{\llbracket \text{Neg}(\{[v_i], e\}) \rrbracket}{H \vdash G}}{\Theta \vdash G} \text{cut}$$

then we can apply our induction hypothesis.

- The case $\sigma[\text{inj}_r v_j] = \text{inj}_r v'$ is similar to the previous case.

- Case $\text{OD}_{A \otimes B}(\{\langle v_i^1, v_i^2 \rangle\})$ with $\sigma[\langle v_j^1, v_j^2 \rangle] = \langle v'_1, v'_2 \rangle$

$$\text{Then } \pi = \frac{\frac{\frac{\llbracket \text{Pos}(v'_1) \rrbracket}{\Theta_1 \vdash H} \quad \frac{\llbracket \text{Pos}(v'_2) \rrbracket}{\Theta_2 \vdash F}}{\Theta_1, \Theta_2 \vdash H \otimes F} \otimes_R \quad \frac{\llbracket \text{Neg}(\{(\langle v_1, v_2 \rangle i), e_i\}_{i \in I}) \rrbracket}{H, F \vdash G}}{H \otimes F \vdash G} \otimes_L}{\Theta_1, \Theta_2 \vdash G} \text{cut}$$

Which reduces to

$$\frac{\frac{\frac{\llbracket \text{Pos}(v'_1) \rrbracket}{\Theta_1 \vdash H} \quad \frac{\frac{\llbracket \text{Pos}(v'_2) \rrbracket}{\Theta_2 \vdash F} \quad \frac{\llbracket \text{Neg}(\{(v_1 :: v_2 :: l)_i, e_i\}_{i \in I}) \rrbracket}{H, F \vdash G}}{\text{cut}}}{\Theta_2, H \vdash G}}{\text{cut}}}{\Theta_1, \Theta_2 \vdash G}$$

Because the negative phase on $[v_1, v_2]$ only produces $\&$, \wp , \top , ν rules, we get that $\text{Neg}(\{(v_1 :: v_2 :: l, e)\}) = \text{Neg}(\{(v_2 :: v_1 :: l, e)\})$ by the commutation of rules of Linear Logic. Therefore, we can get

$$\frac{\frac{\frac{\llbracket \text{Pos}(v'_1) \rrbracket}{\Theta_1 \vdash H} \quad \frac{\frac{\llbracket \text{Pos}(v'_2) \rrbracket}{\Theta_2 \vdash F} \quad \frac{\llbracket \text{Neg}(\{(v_2 :: v_1 :: l)_i, e_i\}_{i \in I}) \rrbracket}{H, F \vdash G}}{\text{cut}}}{\Theta_2, H \vdash G}}{\text{cut}}}{\Theta_1, \Theta_2 \vdash G}$$

which, by induction hypothesis on v_2 , reduces to

$$\frac{\frac{\llbracket \text{Pos}(v'_1) \rrbracket}{\Theta_1 \vdash H} \quad \frac{\llbracket \text{Neg}(\{(v_1 :: l)_i, \text{let } x_j = w_j \text{ in } e_i\}_{i \in I}) \rrbracket}{H \vdash G}}{\text{cut}}}{\Theta_1, \Theta_2 \vdash G}$$

And then we can apply our second induction hypothesis on v_1 . \square

We can then conclude with the global simulation theorem as a direct implication of the two previous lemmas:

Theorem 4.4.29 (Iso-substitution cut-elim). *Let $\{v_1 \leftrightarrow e_1 \mid \dots \mid v_n \leftrightarrow e_n\} v \rightarrow \sigma(e_i)$ when $\sigma[v_i] = v$ then $\llbracket \text{Pos}(\{v_1 \leftrightarrow e_1 \mid \dots \mid v_n \leftrightarrow e_n\} v) \rrbracket \rightsquigarrow^* \llbracket \text{Pos}(\text{let } x_j = v_j \text{ in } e_i) \rrbracket \rightsquigarrow^* \llbracket \text{Pos}(\sigma(e_i)) \rrbracket$ when $\sigma = \{x_j \mapsto v_j\}$*

Proof. By Lemma 4.4.25, we know that the explicit substitution and the substitution coincide. Lemma 4.4.27 tells us that one step of $\rightarrow_{\text{elet}}$ is simulated by exactly one step of cut-elimination and finally Lemma 4.4.28 tells us that we simulate properly the pattern-matching. \square

Corollary 4.4.30 (Simulation). *Provided an iso $\vdash_\omega \omega : A \leftrightarrow B$ and values $\vdash_e v : A$ and $\vdash_e v' : B$, let $\pi = \llbracket \text{Pos}(\omega v) \rrbracket$ and $\pi' = \llbracket \text{Pos}(v') \rrbracket$, if $\omega v \rightarrow^* v'$ then $\pi \rightsquigarrow^* \pi'$.*

Proof. Direct application of Theorem 4.4.29. \square

This leads to the following corollary:

Corollary 4.4.31 (Isomorphism of proofs.). *Given a well-typed iso $\vdash_\omega \omega : A \leftrightarrow B$ and two well-typed close value v_1 of type A and v_2 of type B and the proofs $\pi : F_1 \vdash G_1$, $\pi^\perp : G_2 \vdash F_1$, $\phi : F_3$, $\psi : G_2$ corresponding respectively to the translation of $\omega, \omega^\perp, v_1, v_2$ then:*

$$\frac{\frac{\frac{\phi}{\vdash F_3} \rightsquigarrow}{\vdash F_3} \quad \frac{\frac{\pi}{F_1 \vdash G_1}}{\vdash G_1} \text{ cut}}{\vdash F_2} \quad \frac{\frac{\pi^\perp}{G_2 \vdash F_2}}{\vdash G_1} \text{ cut}}{\vdash F_2} \text{ cut} \quad \frac{\frac{\frac{\psi}{\vdash G_3} \rightsquigarrow}{\vdash G_3} \quad \frac{\frac{\pi^\perp}{F_2 \vdash G_2}}{\vdash F_2} \text{ cut}}{\vdash G_1} \text{ cut}}{\vdash G_1} \text{ cut} \rightsquigarrow \frac{\psi}{\vdash G_3}$$

Proof. As a direct implication of Theorem 4.2.13 and Corollary 4.4.30. \square

4.5. Removing Exhaustivity

While the language presented thus far works with total functions, partial functions are enough to consider reversible computation. In particular, a partial injective function f can be made reversible by considering its inverse f^{-1} as being only defined on the codomain of f . In this section we work on *partial isos* by removing the constraints of exhaustivity imposed by OD_A . We preserve the orthogonality between values, noted $v \perp v'$ and defined in Table 4.6. Most of the result from Section 4.2 still holds except for progress, as the term ωv can be stuck and not reduce, if no pattern of ω matches the value v . What we obtain though is a proof of Turing Completeness for the language, for that we show how any Reversible Turing Machine [MY07] can be encoded as a well-typed, partial iso of the language.

4.5.1. Encoding of Reversible Turing Machines

We want to be able to encode any RTM $(Q, \Sigma, \delta, b, q_s, q_f)$ into our language. As mentioned in Definition 1.1.5, δ is a partial relation: hence the encoded isos will also need to be partial, something not possible with the current typing rules because of the predicate OD_A . We modify the predicate OD_A in order to ensure only non-overlapping through a notion of *orthogonality* between values, noted $v \perp v'$. Obtained isos will then represent *partial injective* functions. Orthogonality is defined similarly as in [SVV18] and its definition is given in Table 4.6. Another constraint we need to lift is termination: a Turing Machine may not terminate, while our isos always do.

We can show that this new definition of orthogonality satisfies the usual condition:

Lemma 4.5.1. *Given a finite set of well-typed value S of type A such that for all $v_1 \neq v_2 \in S, v_1 \perp v_2$, and a value v of type A , if there exists $v_1, v_2 \in S$ such that $\sigma_1[v_1] = v$ and $\sigma_2[v_2] = v$ then $v_1 = v_2$.*

$$\begin{array}{c}
 \frac{}{\text{inj}_l v \perp \text{inj}_r v'} \quad \frac{}{\text{inj}_r v \perp \text{inj}_l v'} \quad \frac{v \perp v'}{\text{fold } v \perp \text{fold } v'} \\
 \frac{v \perp v'}{\text{inj}_l v \perp \text{inj}_l v'} \quad \frac{v \perp v'}{\text{inj}_r v \perp \text{inj}_r v'} \quad \frac{v_1 \perp v_2}{\langle v, v_1 \rangle \perp \langle v', v_2 \rangle} \quad \frac{v_1 \perp v_2}{\langle v_1, v \rangle \perp \langle v_2, v' \rangle}
 \end{array}$$

Table 4.6.: Orthogonality Condition on Values

Proof. By induction on $\sigma_1[v_1] = v$

- Case $\sigma_1[x] = v$: There is no value v_2 such that $v_2 \perp x$, hence $S = \{x\}$ so $v_2 = x$.
- Case $\sigma_1[()] = ()$: Similarly.
- Case $\sigma_1[\text{inj}_l v'_1] = \text{inj}_l v'$: By definition of the pattern-matching we have $\sigma_1[v'_1] = v'$. The only possible value for v_2 is then $\text{inj}_l v'_2$. By IH we get $v'_1 = v'_2$ hence $v_1 = v_2$.
- The case for $\text{inj}_r v_1$ and $\text{fold } v_1$ are similar.
- Case for $\sigma[\langle v_1^1, v_1^2 \rangle] = \langle v^1, v^2 \rangle$: By definition of the pattern-matching we have $\sigma_1[v_1^1] = v^1$ and $\sigma_2[v_1^2] = v^2$. By orthogonality, we have that $v_2 = \langle v_2^1, v_2^2 \rangle$ and therefore by IH $v_1^1 = v_2^1$ and $v_1^2 = v_2^2$ so $v_1 = v_2$. \square

We can also related the OD predicate with this orthogonality:

Lemma 4.5.2. *Given a set of well-typed value $S = \{v_1, \dots, v_n\}$ of type A , if $\text{OD}_A(S)$ then, for all $i \neq j, v_i \perp v_j$.*

Proof. By induction on $\text{OD}_A(S)$.

- Direct for the case where $S = \{x\}$ or $\{()\}$.
- Case where $S = \{\text{inj}_l v \mid v \in S_1\} \cup \{\text{inj}_r v \mid v \in S_2\}$, then we want to show that for all $i \neq j, v_i \perp v_j$ for $v_i, v_j \in S$. We have three cases:
 - If $v_i, v_j \in S_1$ then by direction induction hypothesis on S_1 .
 - Similar is $v_i, v_j \in S_2$
 - If $v_i \in S_1$ and $v_j \in S_2$ (or conversely) then by construction we have $v_i = \text{inj}_l v'_i$ and $v_j = \text{inj}_r v'_j$ and are therefore orthogonal.
- Case $S = \text{inj}_l \text{fold } v \mid v \in S'$ is direct by induction hypothesis on S' .

- Case where $S = \{\langle v_1, v'_1 \rangle, \dots, \langle v_n, v'_n \rangle\}$, assuming the left side of the premise is taken:

We want to show that for all $i \neq j, \langle v_i, v'_i \rangle \perp \langle v_j, v'_j \rangle$.

By induction hypothesis we know $\text{OD}_A(\pi_1(S))$ and therefore $v_i \perp v'_j$. The other case being similar. \square

Then the typing rules for isos become

$$\frac{\begin{array}{c} \Delta_1 \vdash_e v_1 : A \quad \dots \quad \Delta_n \vdash_e v_n : A \quad \forall i \neq j, v_i \perp v_j \\ \Delta_1; \Psi \vdash_e e_1 : B \quad \dots \quad \Delta_n; \Psi \vdash_e e_n : B \quad \forall i \neq j, \text{Val}(e_i) \perp \text{Val}(e_j) \end{array}}{\Psi \vdash_\omega \{v_1 \leftrightarrow e_1 \mid \dots \mid v_n \leftrightarrow e_n\} : A \leftrightarrow B}$$

$$\frac{f : A \leftrightarrow B \vdash_\omega \omega : A \leftrightarrow B}{\Psi \vdash_\omega \mathbf{fix} \ f.\omega : A \leftrightarrow B}$$

In order to encode a Turing Machine $T = (Q, \Sigma, \delta, b, q_s, q_f)$ into an iso $\text{isos}(T)$ we first need to define suitable types and a representation for each component of T .

Definition 4.5.3 (Encoding of States and Tape Symbols). *Given a finite set of states $Q = \{q_1, \dots, q_n\}$ and finite set of tape symbols $\Sigma = \{s_1, \dots, s_m\}$ with $b \in \Sigma, q_s, q_f \in Q$ define the types Q^T and Σ^T as respectively $\overbrace{\mathbb{1} \oplus \dots \oplus \mathbb{1}}^{n+1 \text{ times}}$ and $\overbrace{\mathbb{1} \oplus \dots \oplus \mathbb{1}}^{m+1 \text{ times}}$.*

Then b, q_s and q_f are values of type Q^T and Σ^T taken by convention as: $b = \text{inj}_l()$, $q_s = \text{inj}_l()$, $q_f = \text{inj}_r \text{inj}_l()$.

We write q^T (resp. s^T) for a value of type Q^T (resp. Σ^T) representing the state q (resp. the letter s) of the RTM.

Since the String Semantics defined on Definition 1.1.9 is only defined on terminating run that use a finite amount of tape, we can represent the tapes as a Zipper: a pair of lists of type Σ^T .

Definition 4.5.4 (Encoding of Configurations). *Given a Turing Machine T let $\text{Zipper} = [\Sigma^T] \otimes [\Sigma^T]$*

Define the type of configuration as $C^T = (Q^T \otimes \text{Zipper})$ We represent the configuration $C = (q, (l, a, r))$ as $\text{isos}(C) = (\text{isos}(q), (\text{isos}(a) :: \text{isos}(l), \text{isos}(r)))$ where $\text{isos}(l), \text{isos}(r)$ is the list of element of the encoding of the element on tape where

- *If the tape is empty (infinite tape of empty symbol), $\text{isos}(l) = []$.*
- *If the tape is non-empty but has an infinite suffix of empty symbol, i.e it can be written as $l = [s_1, \dots, s_n, \epsilon, \epsilon, \dots]$ then $\text{isos}(l) = [\text{isos}(s_1), \dots, \text{isos}(s_n)]$.*

We can finally encode the transition relation δ where each element of δ give rise to a clause made of values:

Definition 4.5.5 (Encoding of δ). *We encode each rule of δ as a clause of an iso.*

- *Transition (q, \rightarrow, q') is encoded as the clause*
 $(\text{isos}(q), (l_1, x :: l_2)) \leftrightarrow (\text{isos}(q'), (x :: l_1, l_2)).$
- *Transition (q, \leftarrow, q') is encoded as the clause*
 $(\text{isos}(q), (x :: l_1, l_2)) \leftrightarrow (\text{isos}(q'), (l_1, x :: l_2))$
- *Transition (q, \downarrow, q') is encoded as the clause*
 $(\text{isos}(q), (l_1, l_2)) \leftrightarrow (\text{isos}(q'), (l_1, l_2))$
- *Transition $(q, (s, s'), q')$ is encoded as the clause*
 $(\text{isos}(q), (s^T :: l_1, l_2)) \leftrightarrow (\text{isos}(q'), (s'^T :: l_1, l_2))$

Lemma 4.5.6. *Given two different states (resp. two letter) a_1, a_2 we get $\text{isos}(a_1) \perp \text{isos}(a_2)$*

Proof. Direct as each state (resp. letter) is represented by a distinct value of its type. \square

This lemma tells us that local / backward determinism will imply the orthogonality of the clauses of the iso.

Corollary 4.5.7 ($\text{isos}(T)$ is well-typed). *Given a RTM T , $\text{isos}(T)$ is well-typed iso of type $C^T \leftrightarrow C^T$.*

Lemma 4.5.8 (One Step Simulation). *Given a RTM T , if $T \vdash C \rightsquigarrow C'$ then*

$$\text{isos}(T) \text{ isos}(C) \rightarrow \text{isos}(C')$$

Proof. By analysis of the transition relation.

- If $(q, (l, s, r))$ and $(q, \downarrow, q') \in \delta$ then the configuration becomes $(q', (l, s, r))$. In this case, by definition we have a clause $(\text{isos}(q'), (l, r)) \leftrightarrow (\text{isos}(q'), (l, r))$. The configuration C will be in state $\text{isos}(q')$ and so will enter this clause, the second clause is indeed the configuration $\text{isos}(C)$.
- If $(q, (l, s, a :: r))$ and $(q, \rightarrow, q') \in \delta$ then the configuration becomes $(q', (l \cdot s, a, r))$.
- Similar if $(q, \leftarrow, q') \in \delta$

- If $(q, (l, s, r))$ and $(q, (s, s'), q') \in \delta$ then the configuration becomes $(q', (l, s', r))$
 By translation, we have a rule of the form $(\text{isos}(q'), (s :: l, r)) \leftrightarrow (\text{isos}(q'), (s' :: l, r))$
 which directly send $\text{isos}(C)$ to $\text{isos}(C')$. \square

So far, $\text{isos}(T)$ simulates exactly one evaluation step of the Turing Machine, in order to simulate a full run we need to define an iterator iso that will apply $\text{isos}(T)$ until the configuration reached is in the final state.

Definition 4.5.9 (Iterator Iso). *Let $\text{It}(\omega) : A \leftrightarrow A \otimes \mathbb{N}$ be an iso parametrized by another iso $\omega : A \leftrightarrow A \otimes (1 \oplus 1)$. Remember that $0 = \text{fold inj}_l ()$ and $S n = \text{fold inj}_r n$ of type $\mathbb{N} = \mu X. \mathbb{1} \oplus X$:*

First define $\omega_{\text{aux}} = \left\{ \begin{array}{l} (y, \top) \leftrightarrow \text{let } (z, n) = g y \text{ in } (z, S n) \\ (y, \perp) \leftrightarrow (y, S 0) \end{array} \right\}$

Let $\text{It}(\omega) : A \leftrightarrow A \otimes \mathbb{N}$ be an iso defined as:

$$\text{fix } g. \left\{ \begin{array}{l} x \leftrightarrow \text{let } y = \omega x \text{ in} \\ \quad \text{let } z = \omega_{\text{aux}} y \text{ in} \\ \quad z \end{array} \right\}$$

Assuming that ω is an iso of type $A \leftrightarrow A \otimes (1 \oplus 1)$, then $\text{It}(\omega)$ will iterate ω until it returns some value a, \perp , while counting the number of time it called ω .

Lemma 4.5.10 (Semantics of $\text{It}(\omega)$). *Given $\omega : A \leftrightarrow A \otimes (1 \oplus 1)$ and some value $\vdash_e v : A$ if $\text{It}(\omega) v \rightarrow^* (v', \bar{n})$ then $\underbrace{\omega \dots \omega}_{n+1 \text{ times}} v \rightarrow^* (v', \perp)$*

Proof. By induction on n .

- 0, then $\text{It}(\omega) v \rightarrow \text{let } y = \omega v \text{ in let } z = \omega_{\text{aux}} y \text{ in } z \rightarrow^* \text{let } z = \omega_{\text{aux}} (v', \perp) \text{ in } z \rightarrow (v', 0)$, and hence by hypothesis $\omega v \rightarrow (v', \perp)$.
- $n+1$: $\text{It}(\omega) v \rightarrow \text{let } y = \omega v \text{ in let } z = \omega_{\text{aux}} y \text{ in } z \rightarrow \text{let } z = \omega_{\text{aux}} (v_1, \top) \text{ in } z = \text{let } z = \left\{ \begin{array}{l} (y, \top) \leftrightarrow \text{let } (z, n) = \text{It}(\omega) y \text{ in } (z, S n) \\ (y, \perp) \leftrightarrow (y, S 0) \end{array} \right\} (v_1, \top) \text{ in } z \rightarrow \text{let } y = \text{let } (z, n) = \text{It } v_1 \text{ in } z, S n \text{ in } y$

Then by we know that $\text{It } v_1$ will reduce to v', n and so by IH we get that $\underbrace{\omega \dots \omega}_{n+1 \text{ times}} v_1 \rightarrow^* (v', \perp)$, so we get $\underbrace{\omega \dots \omega}_{n+2 \text{ times}} v \rightarrow^* (v', \perp)$. \square

With this, we just need to adapt the encoding δ to the type $C^T \leftrightarrow C^T \otimes (1 \oplus 1)$:

Definition 4.5.11 (Encoding of δ). *We modify the encoding of δ to be of type $C^T \leftrightarrow C^T \otimes (1 \oplus 1)$ to be the same but if the translation leads into the final state, the second argument is sent to \perp otherwise it is sent to \top . By abuse of notation, we also call this encoding $\text{isos}(T)$.*

The new encoding does not change the orthogonality and hence the new iso is well-typed:

Lemma 4.5.12 ($\text{isos}(\delta)$ is well-typed). *Given a RTMT, then $\vdash_{\omega} \text{isos}(\delta) : C^T \otimes (1 \oplus 1) \leftrightarrow C^T \otimes (1 \oplus 1)$.*

Proof. Same as Corollary 4.5.7. □

Finally, we get that if a Turing Machine, from an initial configuration C evaluates into the final configuration C' in $n + 1$ steps, then the Iterator iso on $\text{isos}(\delta)$ with input the encoding of C reduces to the encoding of C' with the encoding of the number n .

Theorem 4.5.13 (Simulation of String Semantics). *Let T be a RTM, if*

$$T \vdash (q_s, (\epsilon, b, s)) \rightsquigarrow^{n+1} (q_f, (\epsilon, b, s'))$$

then

$$It(\text{isos}(\delta)) \text{ isos}(q_s, ([], b, s)) \rightarrow^* (\text{isos}(q_f, (\epsilon, b, s')), \bar{n})$$

.

Proof. Direct by Lemma 4.5.8 and Lemma 4.5.10. □

4.6. Conclusion

Summary of the contribution. We presented a linear, reversible language with inductive types, extending the language from [SVV18]. We showed how ensuring non-overlapping and exhaustivity is enough to ensure the reversibility of the isos. The language comes with both an expressivity result that shows that any Primitive Recursive Functions can be encoded in this language as well as an interpretation of programs into μMALL proofs. The latter result rests on the fact that our isos are *structurally recursive*. We then removed the constraints of exhaustivity: while still preserving a notion of reversibility on partial injective functions, we showed how this version of the languages allows us to encode any Reversible Turing Machine, hence attaining Turing Completeness.

Future works. A first extension to our work would be to relax the structurally recursive condition to allow for more functions to be encoded. For instance, the Cantor Pairing $\mathbb{N} \leftrightarrow \mathbb{N} \otimes \mathbb{N}$ can be encoded as:

Example 4.6.1 (Cantor Pairing).

$$\omega_1 = \left\{ \begin{array}{l} \langle S \ i, j \rangle \leftrightarrow \text{inj}_l (\langle i, S \ j \rangle) \\ \langle 0, S \ S \ j \rangle \leftrightarrow \text{inj}_l (\langle S \ j, 0 \rangle) \\ \langle 0, S \ 0 \rangle \leftrightarrow \text{inj}_l (\langle 0, 0 \rangle) \\ \langle 0, 0 \rangle \leftrightarrow \text{inj}_r () \end{array} \right\} : \mathbb{N} \otimes \mathbb{N} \leftrightarrow (\mathbb{N} \otimes \mathbb{N}) \oplus \mathbf{1}$$

$$\omega_2 = \left\{ \begin{array}{l} \text{tinj}_l (x) \leftrightarrow \text{let } y = g \ x \ \text{in } S \ y \\ \text{inj}_r (x) \leftrightarrow 0 \end{array} \right\} : (\mathbb{N} \otimes \mathbb{N}) \oplus \mathbf{1} \leftrightarrow \mathbb{N}$$

$$\text{CantorPairing} = \mathbf{fix} \ g. \left\{ \begin{array}{l} x \leftrightarrow \text{let } y = \omega_1 \ x \ \text{in} \\ \quad \text{let } z = \omega_2 \ y \ \text{in } z \end{array} \right\} : (\mathbb{N} \otimes \mathbb{N}) \leftrightarrow \mathbb{N}$$

While the iso has the expected operational semantics, it is not well-typed as it is not structurally recursive. One would require accepting lexicographical order (or more generally, any well-founded order) on recursive isos that act as a termination proof. And then, see how such a criterion would be captured in terms of pre-proof validity. Along with this, allowing for coinductive statements and terms would allow for a truly general reversible language. This is a focus of our forthcoming research.

On a more denotational point of view, the works in collaboration with Louis Lemonnier [CLV21] is currently being extended to the case of the language presented in this chapter, along to a categorical semantics, with Robin Kaarsgaard, for the quantum version of the language from [SVV18] extended with inductive types, as done in this chapter.

Finally, we want to consider quantum computation, by extending our language with linear combinations of terms. We plan to study purely quantum recursive types and generalized quantum loops: in [SVV18], lists are the only recursive type which is captured, and recursion is terminating. The logic μMALL would help in providing a finer understanding of termination and non-termination.

Chapter 5.

Geometry of Interaction for ZX Calculus

Abstract

Getting inspiration from the Geometry of Interaction as a Token Machine, we introduce a Token Machine for the ZX-Calculus and its extension to mixed-processes. We show how the Token Machine captures the denotational semantics of the ZX-Calculus. We then discuss variants of the token machine, in particular in the context of Sum-Over-Paths semantics.

References: Results of this chapter have been published in the paper *Geometry of Interaction for ZX-Calculus* at MFCS 2021 [CVV21].

5.1. Introduction

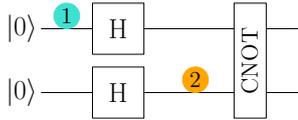
As seen previously, the standard models of both quantum circuits and the ZX-Calculus is based on linear operators in some Hilbert space, most often described with a matrix interpretation. An alternative operational interpretation of quantum circuits following a *particle-style* semantics has recently been investigated in the literature [Dal17]. In this model, quantum bits are intuitively seen as *tokens* flowing inside the wires of the circuit. Formally, a quantum circuit is interpreted as a token-based automata, based on Geometry of Interaction (GoI) [Gir89b; Gir89a; Gir88; Gir95; Gir06; Gir11; Gir13]. This framework is used in [Dal17] to formalize the notion of qubits-as-tokens flowing inside a higher-order term representing a quantum computation—that is, computing a quantum circuit. However, in this work, quantum gates are still regarded as black-boxes, and tokens are purely classical objects *requiring synchronicity*: to fire, a two-qubit gate needs its two arguments to be ready.

As a summary, despite their ad-hoc construction, quantum circuits can be seen from two perspectives: computation as a flow of particles (i.e., tokens), and as a wave passing through the gates, i.e. the standard vectorial state representation. On the other hand, although ZX-Calculus is a well-established language, it still misses such a particle-style perspective.

In this chapter, we aim at giving a novel insight on the computational content of a ZX term in an asynchronous way, emphasizing the non-locality of the behaviour of a ZX-computation.

Following the idea of using a token machine to exhibit the computational content of a proof-net or a quantum circuit, we present in this chapter a token machine for the ZX-Calculus. To exemplify the versatility of the approach, we show how to extend it to mixed processes [CP12; Car+19] and to the Sum-Over-Path semantics, the development for both perspectives being very similar. Those two perspectives are related to the notion of particle-style and wave-style semantics, similarly to what has been done for quantum circuits [Dal17]. To assess the validity of the semantics, we show how it links to the standard interpretation of ZX-diagrams. While the standard interpretation of ZX-diagrams proceeds with conventional graph rewriting, the tokens flowing inside the diagram do not modify it, and the computation emerges from the ability of tokens to be in superposition.

This ability illustrates one fundamental difference between our approach and the one in [Dal17]. The latter follows a *classical control* approach: if qubits can be in superposition, each qubit inhabits a token sitting in *one single* position in the circuit. For instance, on the circuit below, the state of the two tokens $|\bullet\circ\rangle$ is $\frac{\sqrt{2}}{2}(|00\rangle + |10\rangle)$. Although the two tokens can be regarded as being in superposition, their *position* is not. In our system, tokens and positions can be superposed.



The second fundamental difference lies in the *asynchronicity* of our token-machine. Unlike [Dal17], we rely on the canonical generators of ZX-diagrams: tokens can travel through these nodes in an asynchronous manner. For instance, in the above circuit the orange token must wait for the blue token before crossing the CNOT gate. As illustrated Table 5.1, in our system one token can interact with multi-wire nodes. Finally, as formalized in Theorem 5.3.25, a third difference is that compared to [Dal17], the token-machine we present is *non-oriented*: in the circuit above, tokens have to start on the left and flow towards the right of the circuit whereas our system is agnostic on where tokens initially “start”.

Organization of the chapter The chapter is organized as follows: in Section 5.3 we present the first token machine, with the rewriting invariant needed to obtain confluence, termination and the relation with the standard interpretation of the ZX-Calculus, then in Section 5.4 we consider the extension of mixed-processes and relate the new token machine with the previous one through the use of the map CPM. Then we briefly discuss two variations of the token machine in Section 5.5. Finally, in Section 5.6 we present the Sum-Over-Paths semantics token machine.

5.2. Notions of Graph Theory in ZX

In order to work with our token machine, we need to attribute to each wire of the ZX-Diagram a name, given as such:

$$\left\{ \begin{array}{l} e_0, \text{ } \begin{array}{c} e_0 \\ \diagdown \quad \diagup \\ e_1 \end{array}, e_0 \cup e_1, e_0 \cap e_1, \begin{array}{c} e_1 \quad e_n \\ \dots \\ \alpha \\ \dots \\ e'_1 \quad e'_m \end{array}, \begin{array}{c} e_0 \\ \square \\ e_1 \end{array} \end{array} \right\} \begin{array}{l} n, m \in \mathbb{N} \\ \alpha \in \mathbb{R} \\ e_i, e'_i \in \mathcal{E} \end{array}$$

We shall be using the following labelling convention: wires (edges) are labelled with e_i , taken from an infinite set of labels \mathcal{E} . We take for granted that distinct wires have distinct labels. We write $\mathcal{E}(D)$ for the set of edge labels in the diagram D , and $\mathcal{I}(D)$ (resp. $\mathcal{O}(D)$) for the list of input edges (resp. output edges) of D .

We need to have a special treatment of edge labels when composing two diagrams D_2 and D_1 :

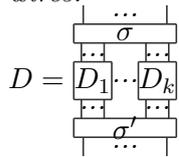
For the sequential composition $D_2 \circ D_1$ we need to do a relabelling of the input edges of the bottom diagram by the output labels of the top diagram, we also require that $\mathcal{E}(D_2 \circ D_1) = \mathcal{E}(D_1) \cup \mathcal{E}(D_2) \setminus \mathcal{I}(D_2)$, $\mathcal{I}(D_2 \circ D_1) = \mathcal{I}(D_1)$ and $\mathcal{O}(D_2 \circ D_1) = \mathcal{O}(D_2)$.

While for the parallel composition we require that $\mathcal{E}(D_2 \otimes D_1) = \mathcal{E}(D_1) \cup \mathcal{E}(D_2)$, $\mathcal{I}(D_2 \otimes D_1) = \mathcal{I}(D_2) \cup \mathcal{I}(D_1)$ and $\mathcal{O}(D_2 \otimes D_1) = \mathcal{O}(D_2) \cup \mathcal{O}(D_1)$.

Remind that we assume that $\mathcal{E}(D_1) \cap \mathcal{E}(D_2) = \emptyset$

Theorem 2.1.5 is essential: it allows us to transpose notions of graphs into ZX-Calculus. It is for instance possible to define a notion of connectivity.

Definition 5.2.1 (Connected Components). *Let D be a non-empty ZX-diagram. Consider all the possible decompositions with $D_1, \dots, D_n \in \mathbf{ZX}$ and σ, σ' permutations of wires:*



The largest such k is called the number of connected components of D . It induces a decomposition. The induced D_1, \dots, D_n are called the connected components of D . If D has only one connected component, we say that D is connected.

We can also consider the notions of paths, distance and cycles of usual multi-graphs. We denote $\text{Paths}(e, e')$ the set of paths from edge e to e' . The set of paths (resp. cycles) of a diagram D is denoted by $\text{Paths}(D)$ (resp. $\text{Cycles}(D)$). For a path p , we denote $|p|$ its length. We denote $d(e, e')$ the distance i.e. the length of the shortest path between e and e' .

In the remainder of the thesis, we omit the edge labels when not necessary.

5.3. A Token Machine for ZX-diagrams

Inspired by the Geometry of Interaction [Gir89b; Gir89a; Gir88; Gir95; Gir06; Gir11; Gir13] and the associated notion of token machine [DR99; AL95] for proof nets [Gir96], we define here a first token machine on pure ZX-diagrams. A token consists of an edge of the diagram, a direction (either going up, noted \uparrow , or down, noted \downarrow) and a bit (state). The idea is that, starting from an input edge the token will traverse the graph and duplicate itself when encountering an n-ary node (such as the green and red node) into each of the input / output edges of the node. Notice that it is not the case for token machines for proof-nets where the token never duplicates itself. This duplication is necessary to make sure we capture the whole linear map encoded by the ZX-diagram. Due to this duplication, two tokens might collide together when they are on the same edge and going in different directions. The result of such a collision will depend on the states held by both tokens. For a cup, cap or identity diagram, the token will simply traverse it. As for the Hadamard node the token will traverse it and become a superposition of two tokens with opposite states. Therefore, as tokens move through a diagram, some may be added, multiplied together, or annihilated.

Definition 5.3.1 (Tokens and Token States). *Let D be a ZX-diagram. A token in D is a triplet $(e, d, b) \in \mathcal{E}(D) \times \{\downarrow, \uparrow\} \times \{0, 1\}$. We shall omit the commas and simply write $(e \ d \ b)$. The set of tokens on D is written $\mathbf{tk}(D)$. A token state s is then a multivariate polynomial over \mathbb{C} , evaluated in $\mathbf{tk}(D)$. We define $\mathbf{tkS}(D) := \mathbb{C}[\mathbf{tk}(D)]$ the algebra of multivariate polynomials over $\mathbf{tk}(D)$.*

In the token state $t = \sum_i \alpha_i t_{1,i} \cdots t_{n,i}$, where the $t_{k,i}$'s are tokens, the components $\alpha_i t_{1,i} \cdots t_{n,i}$ are called the terms of t .

A monomial $(e_1 \ d_1 \ b_1) \cdots (e_n \ d_n \ b_n)$ encodes the state of n tokens in the process of flowing in the diagram D . A token state is understood as a *superposition* —a linear combination— of multi-tokens flowing in the diagram.

Convention 5.3.2. *In token states, the sum $(+)$ stands for the superposition while the product stands for additional tokens within a given diagram. We follow the usual convention of algebras of polynomials: for instance, if t_i stands for some token $(e_i \ d_i \ b_i)$, then $(t_1 + t_2)t_3 = (t_1t_2) + (t_1t_3)$, that is, the superposition of t_1, t_2 flowing in D and t_1, t_3 flowing in D . Similarly, we consider token states modulo commutativity of sum and product, so that for instance the monomial t_1t_2 is the same as t_2t_1 . Notice that 0 is an absorbing element for the product ($0 \times t = 0$) and that 1 is a neutral element for the same operation ($1 \times t = t$).*

Example 5.3.3. Given $D = \begin{array}{c} e_1 \ e_n \\ \cdots \\ \uparrow \\ \bullet \\ \downarrow \\ \cdots \\ e'_1 \ e'_m \end{array}$ then $(e_1 \uparrow 0)(e_2 \uparrow 1) + (e'_3 \downarrow 1)$ is a token state on D .

$e_0 \downarrow$	$(e_0 \downarrow x)(e_0 \uparrow x) \rightsquigarrow_c 1$	$(e_0 \downarrow x)(e_0 \uparrow \neg x) \rightsquigarrow_c 0$	(Positive/Negative Collision)
$e_0 \cup e_1$	$(e_b \downarrow x) \rightsquigarrow_d (e_{\neg b} \uparrow x)$		(\cup -diffusion)
$e_0 \cap e_1$	$(e_b \uparrow x) \rightsquigarrow_d (e_{\neg b} \downarrow x)$		(\cap -diffusion)
$e_1 \dots e_n$ 	$(e_k \downarrow x) \rightsquigarrow_d e^{i\alpha x} \prod_{i \neq k} (e_i \uparrow x) \prod_j (e'_j \downarrow x)$		(Green Spider -Diffusion)
$e'_1 \dots e'_m$	$(e'_k \uparrow x) \rightsquigarrow_d e^{i\alpha x} \prod_{j \neq k} (e'_j \downarrow x) \prod_i (e_i \uparrow x)$		
e_0 	$(e_0 \downarrow x) \rightsquigarrow_d (-1)^x \frac{1}{\sqrt{2}} (e_1 \downarrow x) + \frac{1}{\sqrt{2}} (e_1 \downarrow \neg x)$		(Yellow Square -Diffusion)
e_1	$(e_1 \uparrow x) \rightsquigarrow_d (-1)^x \frac{1}{\sqrt{2}} (e_0 \uparrow x) + \frac{1}{\sqrt{2}} (e_0 \uparrow \neg x)$		

 Table 5.1.: Asynchronous token-state evolution, for all $x, b \in \{0, 1\}$

5.3.1. Diffusion and Collision Rules

The tokens in a ZX-diagram D are meant to move inside D . The set of rules presented in this section describes an *asynchronous* evolution, meaning that given a token state, we will rewrite only one token at a time. The synchronous setting is discussed in Section 5.5.2.

Definition 5.3.4 (Asynchronous Evolution). *Token states on a diagram D are equipped with two transition systems:*

- a collision system (\rightsquigarrow_c), whose effect is to annihilate tokens;
- a diffusion system (\rightsquigarrow_d), defining the flow of tokens within D .

The two systems are defined as follows. With $X \in \{d, c\}$ and $1 \leq j \leq n_i$, if $t_{i,j}$ are tokens in $\mathbf{tk}(D)$, then using Convention 5.3.2,

$$\sum_i \alpha_i t_{i,1} \cdots t_{i,j} \cdots t_{i,n_i} \rightsquigarrow_X \sum_i \alpha_i t_{i,1} \cdots \left(\sum_k \beta_k t'_k \right) \cdots t_{i,n_i}$$

provided that $t_{i,j} \rightsquigarrow_X \sum_k \beta_k t'_k$ according to the rules of Table 5.1. In the table, each rule corresponds to the interaction with the primitive diagram constructor on the left-hand-side. Variables x and b span $\{0, 1\}$, and \neg stands for the negation. In the green-spider rules, $e^{i\alpha x}$ stands for the complex number $\cos(\alpha x) + i \sin(\alpha x)$ and not an edge label.

Finally, as it is customary for rewrite systems, if (\rightarrow) is a step in a transition system, (\rightarrow^*) stands for the reflexive, transitive closure of (\rightarrow) .

We do not give the rewriting rule for the red-spider since it can be recovered by Convention 2.1.1, for reference one can check that the rules in Table 5.2 are correct.

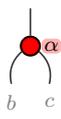
	$(a \downarrow x) \rightsquigarrow_d \frac{1}{\sqrt{2}} (1 + (-1)^x e^{i\alpha})$
	$(a \downarrow x) \rightsquigarrow_d \frac{1}{2\sqrt{2}} \sum_{y,z} (1 + (-1)^{x+y+z} e^{i\alpha}) (b \downarrow y)(c \downarrow z)$

Table 5.2.: Samples of asynchronous token-state evolution for red spiders

We aim at a transition system marrying both collision and diffusion steps. However, for consistency of the system, the order in which we apply them is important as illustrated by the following example.

Example 5.3.5. Consider the equality given by the ZX equational theories:  = .

If we drop a token with bit 0 at the top, we hence expect to get a single token with bit 0 at the bottom. We underline the token that is being rewritten at each step. This is what we get when giving the priority to collisions:

	$(a \downarrow 0) \rightsquigarrow_d \underline{(b \downarrow 0)}(c \downarrow 0) \rightsquigarrow_d (d \downarrow 0)\underline{(c \uparrow 0)}(c \downarrow 0) \rightsquigarrow_c (d \downarrow 0)$
--	--

Notice that the collision $(c \uparrow 0)(c \downarrow 0)$ rewrites to 1, and therefore the product $(d \downarrow 0) \times 1 = (d \downarrow 0)$. If however we decide to ignore the priority of collisions, we may end up with a non-terminating run, unable to converge to $(d \downarrow 0)$:

$$(a \downarrow 0) \rightsquigarrow_d \underline{(b \downarrow 0)}(c \downarrow 0) \rightsquigarrow_d (d \downarrow 0)\underline{(c \uparrow 0)}(c \downarrow 0) \rightsquigarrow_d (d \downarrow 0)(a \uparrow 0)(b \downarrow 0)(c \downarrow 0) \rightsquigarrow_d \dots$$

We therefore set a rewriting strategy as follows.

Definition 5.3.6 (Collision-Free). A token state s of $\mathbf{tkS}(D)$ is called collision-free if for all $s' \in \mathbf{tkS}(D)$, we have $s \not\rightsquigarrow_c s'$.

Definition 5.3.7 (Token Machine Rewriting System). We define a transition system \rightsquigarrow as exactly one \rightsquigarrow_d rule followed by all possible \rightsquigarrow_c rules. In other words, $t \rightsquigarrow u$ if and only if there exists t' such that $t \rightsquigarrow_d t' \rightsquigarrow_c^* u$ and u is collision-free.

In [Dal17], a token arriving at an input of a gate is blocked until all the inputs of the gates are populated by a token, at which point all the tokens go through at once (while obviously changing the state). The control is purely classical: it is causal. In our approach, the state of the system is global and there is no explicit notion of qubit. Instead, tokens collect the operations that are to be applied to the input qubits.

5.3.2. Strong Normalization and Confluence

The token machine Rewrite System of Definition 5.3.7 ensures that the collisions that can happen always happen. The system does not a priori forbid two tokens on the same edge, provided that they have the same direction. However, this is something we want to avoid as there is no good intuition behind it: we want to link the token machine to the standard interpretation, which is not possible if two tokens can appear on the same edge.

In this section we show that, under a notion of well-formedness characterizing token's uniqueness on each edge, the Token State Rewrite System (\rightsquigarrow) is strongly normalizing and confluent.

Definition 5.3.8 (Polarity of a Term in a Path). *Let D be a ZX-diagram, and $p \in \text{Paths}(D)$ be a path in D . Let $t = (e, d, x) \in \mathbf{tk}(D)$. Then:*

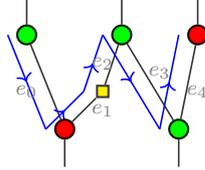
$$P(p, t) = \begin{cases} 1 & \text{if } e \in p \text{ and } e \text{ is } d\text{-oriented} \\ -1 & \text{if } e \in p \text{ and } e \text{ is } \neg d\text{-oriented} \\ 0 & \text{if } e \notin p \end{cases}$$

We extend the definition to subterms $\alpha t_1 \dots t_m$ of a token-state s :

$$P(p, 0) = P(p, 1) = 0, \quad P(p, \alpha t_1 \dots t_m) = P(p, t_1) + \dots + P(p, t_m).$$

In the following, we shall simply refer to such subterms as “terms of s ”.

Example 5.3.9. *In the (piece of) diagram presented below, the blue directed line $p = (e_0, e_1, e_2, e_3, e_4)$ is a path. The orientation of the edges in the path is represented by the arrow heads, and e_3 for instance is \downarrow -oriented in p which implies that we have $P(p, (e_3 \uparrow x)) = -1$.*



Definition 5.3.10 (Well-formedness). *Let D be a ZX-diagram, and $s \in \mathbf{tkS}(D)$ a token state on D . We say that s is well-formed if for every term t in s and every path $p \in \text{Paths}(D)$ we have $P(p, t) \in \{-1, 0, 1\}$.*

Intuitively, this definition tells us that when we have multiple tokens on the same path, they will all collide with each other until there is at most one left, going in either direction.

Proposition 5.3.11 (Invariance of Well-Formedness). *Well-formedness is preserved by (\rightsquigarrow): if $s \rightsquigarrow^* s'$ and s is well-formed, then s' is well-formed.*

Proof. Let D be a ZX-diagram, and s be a well-formed token state on D . Consider a rewrite $s \rightsquigarrow s'$. We want to show that for all paths p in D , if $P(p, t) \in \{-1, 0, 1\}$ for all terms t of s , then $P(p, t') \in \{-1, 0, 1\}$ for all terms t' in s' .

Let t be a term of s , and e_0 be the edge where a rewriting occurs. If the rewriting does not affect t , then the well-formedness of t obviously holds. If it does, and $t \rightsquigarrow_{c,d} \sum_q t_q$, we have to check two cases:

- Collision: let $p \in \text{Paths}(D)$. If no token remains in the term t_q , then $P(p, t_q) = 0$. Otherwise:
 - if $e_0 \notin p$, then $P(p, t_q) = P(p, t) = 0$;
 - if $e_0 \in p$, then $P(p, t_q) = P(p, t) + 1 - 1$ because the two tokens have alternating polarity.
- Diffusion: let $p \in \text{Paths}(D)$, and $(e_0, d, x) \rightsquigarrow_d \sum_q \lambda_q \prod_{i \in S} (e_i, d_i, x_{i,q})$ (this captures all possible diffusion rules).
 - if $e_0 \notin p$ and $\forall i \in S, e_i \notin p$, then $P(p, t_q) = P(p, t)$;
 - if $e_0 \in p$ and $\exists k_1, \dots, k_n \in S$ such that $\forall i \in \{1, \dots, n\}, e_{k_i} \in p$ then we want to show that $P(p, (e_0, d, x)) = \sum_{i \in \{1, \dots, n\}} P(p, (e_{k_i}, d_i, x_i))$. For that, consider a subpath forming a cycle c between k_i and k_j , both k_i and k_j will have a token on it as the result of the diffusion rule of e_0 , then we can reason by case analysis on the orientation of the path on k_i and k_j , and it can be shown that in everything case we have $P(c, (k_i, d_i, x_i)) + P(c, (k_j, d_j, x_j)) = 0$. Hence the polarity on the whole path p is preserved;
 - if $e_0 \in p$ and $\forall i, e_i \notin p$, then, either (i) p ends with e_0 and e_0 is d -oriented in p , or (ii) p starts with e_0 and e_0 is $\neg d$ -oriented in p . In both cases, since that $p \setminus \{e_0\}$ is still a path, we have $P(p \setminus \{e_0\}, t) \in \{-1, 0, 1\}$ and since $P(p, t_q) = P(p \setminus \{e_0\}, t)$, we deduce that t_q is still well-formed;
 - if $e_0 \notin p$ but $\exists k \in S, e_k \in p$, either e_k is an endpoint of p , or $\exists k', e_{k'} \in p$. In the latter case, the tokens in e_k and $e_{k'}$ will have alternating polarity in p , so $\forall q, P(p, t_q) = P(p, t) + 1 - 1$. In the first case, we can show in a way similar to the previous point, that $P(p, t_q) = P(p \setminus \{e_k\}, t) \in \{-1, 0, 1\}$. \square

Well-formedness prevents the unwanted scenario of having two tokens on the same wire, and oriented in the same direction (e.g. $(e_0 \downarrow x)(e_0 \downarrow y)$). As shown in the Proposition 5.3.12, this property is in fact stronger.

Proposition 5.3.12 (Full Characterization of Well-Formed Terms). *Let D be a ZX-diagram, and $s \in \mathbf{tkS}(D)$ be ill-formed, i.e. there exists a term t in s , and $p \in \text{Paths}(D)$ such that $|P(p, t)| \geq 2$. Then we can rewrite $s \rightsquigarrow s'$ such that a term in s' has a product of at least two tokens of the form $(e_0, d, -)$.*

Proof. Let t be a term in s , and $p = (e_0, \dots, e_n)$ such that $P(p, t) \geq 2$. We can show that we can rewrite t into a token state with term $t' = (e_i, d, _)(e_i, d, _)t''$. We do so by induction on $n = |p| - 1$.

If $n = 0$, we have a path constituted of one edge, such that $|P(p, t)| \geq 2$. Even after doing all possible collisions, we are left with $|P(p, t)|$ tokens on e_0 , and oriented accordingly.

For $n + 1$, we look at e_0 , build $p' := (e_1, \dots, e_n)$, and distinguish four cases.

- If there is no token on e_0 , we have $P(p', t) = P(p, t)$, so the result is true by induction hypothesis on p' .
- If we have a product of at least two tokens going in the same direction, the result is directly true.
- If we have exactly one token going in each direction, we apply the collision rules, and therefore we have $P(p', t) = P(p, t)$, so the result is true by induction hypothesis on p' .
- Finally, if we have exactly one token $(e_0, d, _)$ on e_0 , either e_0 is not d -oriented, in which case $P(p', t) = P(p, t) + 1$, or e_0 is d -oriented, in which case the adequate diffusion rule on $(e_0, d, _)$ will rewrite $t \rightsquigarrow \sum_q t_q$ with $P(p', t_q) = P(p, t)$.

□

Although well-formedness prevents products of tokens on the same wire, it does not guarantee termination: for this we need to consider polarities along cycles.

Proposition 5.3.13 (Invariant on Cycles). *Let D be a ZX-diagram, and $c \in \text{Cycles}(D)$ a cycle. Let t_1, \dots, t_n be tokens, and s be a token state such that $t_1 \dots t_n \rightsquigarrow^* s$. Then for every non-null term t in s we have $P(c, t_1 \dots t_n) = P(c, t)$.*

Proof. The proof can be adapted from the previous one, by forgetting the cases related to the endpoint of the paths, as well as the null terms (which can arise from collisions). It can then be observed that the quantity P in this simplified setting is more than bounded to $\{-1, 0, 1\}$, but preserved. □

This proposition tells us that the polarity is preserved inside a cycle. By requiring the polarity to be 0, we can show that the token machine terminates. This property is defined formally in the following.

Definition 5.3.14 (Cycle-Balanced Token State). *Let D be a ZX-diagram, and t a term in a token state on D . We say that t is cycle-balanced if for all cycles $c \in \text{Cycles}(D)$ we have $P(c, t) = 0$. We say that a token state is cycle-balanced if all its terms are cycle-balanced.*

To show that being cycle-balanced implies termination, we need the following intermediate lemma. This essentially captures the fact that a token in the diagram comes from some other token that “travelled” in the diagram earlier on.

Lemma 5.3.15 (Rewinding). *Let D be a ZX-diagram, and t be a term in a well-formed token state on D , and such that $t \rightsquigarrow^* \sum_i \lambda_i t_i$, with $(e_n, d, x) \in t_1$. If t is cycle-balanced, then there exists a path $p = (e_0, \dots, e_n) \in \text{Paths}(D)$ such that e_n is d -oriented in p , and $P(p, t) = 1$.*

Proof. We reason by induction on the length k of the rewrite that leads from t to $\sum_i \lambda_i t_i$. If $k = 0$, we have $(e_n, d, x) \in t$, so the path $p := (e_n)$ is sufficient.

For the induction case, $k = n + 1$, suppose $t \rightsquigarrow \sum_i \lambda_i t_i$, and $t_1 \rightsquigarrow^n \sum_j \lambda'_j t'_j$ (hence $t \rightsquigarrow^{n+1} \sum_{i \neq 1} \lambda_i t_i + \sum_j \lambda'_j t'_j$), with $(e_n, d, x) \in t'_1$. By induction hypothesis, there is $p = (e_0, \dots, e_n)$ such that $P(p, t_1) = 1$. We now need to look at the first rewrite from t .

- if the rewrite concerns a generator not in p , then $P(p, t) = P(p, t_1) = 1$;
- if the rewrite is a collision, then $P(p, t) = P(p, t_1) = 1$;
- if the rewrite is $(e, d_e, x_e) \rightsquigarrow \sum_q \lambda_q \prod_i (e'_i, d_i, x_{i,q})$
 - If $e \in p$ and $e'_1 \in p$, then $P(p, t) = P(p, t_1) = 1$.
 - If $e'_1 \in p$ and $e'_2 \in p$, then $P(p, t) = P(p, t_1) - 1 + 1 = 1$.
 - The case $e \in p$ and $\forall i, e'_i \notin p$ is impossible:
 - * if e is not d_e -oriented in p , it means $e = e_0$, hence $P((e_1, \dots, e_n), t) = P(p, t) + 1 = 2$ which is forbidden by well-formedness;
 - * if e is d_e -oriented in p , it means $e = e_n$, which would imply that $P(p, t_1) = 0$.
 - If $e \notin p$ and $e'_1 \in p$ and $\forall i \neq 1, e'_i \notin p$, then $P(e :: p, t) = P(p, t_1) = 1$, since well-formedness prevents the otherwise possible situation $P(e :: p, t) = P(p, t_1) + 1 = 2$. However, $e :: p$ may not be a path any more. If $c = (e, e_0, \dots, e_\ell)$ forms a cycle, then, since $P(c, t) = 0$, we can simply keep the path $p' := (e_{\ell+1}, \dots, e_n)$ with $P(p', t) = 1$. \square

We can now prove strong-normalization.

Theorem 5.3.16 (Termination of well-formed, cycle-balanced token state). *Let D be a ZX-diagram, and $s \in \mathbf{tkS}(D)$ be well-formed. The token state s is strongly normalizing if and only if it is cycle-balanced.*

Proof. $[\Rightarrow]$: Suppose $\exists c \in \text{Cycles}(D)$ and t a term of s such that $P(c, t) \neq 0$. By well-formedness, $P(c, t) \in \{-1, 1\}$. Any terminal term t' has $P(c, t') = 0$, so by preservation of the quantity $P(c, _)$, t (and henceforth s) cannot terminate.

$[\Leftarrow]$: We are going to show for the reciprocal that, if t is well-formed, and if the constraint $P(c, t) = 0$ is verified for every cycle c , then any generator in the diagram can be visited at most once. More precisely, we show that if a generator is visited in a term t , then it cannot be visited any more in all the terms derived from t . However, the same generator can be visited once for each superposed term (e.g. once in t_1 and once in t_2 for the token state $t_1 + t_2$).

Consider an edge e with token exiting generator g in the term t . Suppose, by contradiction, that a token will visit g again in t' (obtained from t), by edge e_n with orientation d . By Lemma 5.3.15, there exists a path $p = (e_0, \dots, e_n)$ such that $P(p, t) = 1$ and e_n is d -oriented. Since $e \notin p$ (we would not have a path then), then $p' := (e_0, \dots, e_n, e)$ is a path (or possibly a cycle) such that $P(p', t) = 2$. This is forbidden by well-formedness. Hence, every generator can be visited at most once. As a consequence, the lexicographic order $(\#g, \#tk)$ (where $\#g$ is the number of non-visited generators in the diagram, and $\#tk$ the number of tokens in the diagram) strictly reduces with each rewrite. This finishes the proof of termination. \square

Intuitively, this means that tokens inside a cycle will cancel themselves out if the token state is cycle-balanced. Since cycles are the only way to have a non-terminating token machine, we are sure that our machine will always terminate.

Example 5.3.17. *Going back to the diagram from Example 5.3.5:*  *consider the token state $(b \downarrow 0)(c \uparrow 0)$ we get that the polarity in the cycle (b, c, b) is 2 and hence the token state will not terminate, which is indeed the case as:*

$$(b \downarrow 0)(c \uparrow 0) \rightsquigarrow_d (b \downarrow 0)(b \downarrow 0)(a \uparrow 0) \rightsquigarrow_d (b \downarrow 0)(c \uparrow 0)(d \downarrow 0)(a \uparrow 0)$$

The tokens $(b \downarrow 0)$ and $(c \uparrow 0)$ will never collide, hence termination cannot be ensured.

Proposition 5.3.18 (Local Confluence). *Let D be a ZX-diagram, and $s \in \mathbf{tkS}(D)$ be well-formed and collision-free. Then, for all $s_1, s_2 \in \mathbf{tkS}(D)$ such that $s_1 \leftarrow s \rightsquigarrow s_2$, there exists $s' \in \mathbf{tkS}(D)$ such that $s_1 \rightsquigarrow^* s' \leftarrow^* s_2$.*

Proof. We are going to reason on every possible pair of rewrite rules that can be applied from a single token state s . Notice first, that if the two rules are applied on two different terms of s , such that the rewriting of a term creates a copy of the other, they obviously

$$\begin{array}{ccc} s & \rightsquigarrow & s_2 \\ \text{commute, so} & \begin{array}{c} \downarrow & \downarrow \\ s_1 & \rightsquigarrow & s' \end{array} & \end{array} .$$

In the case where $s = \alpha t + \beta t_1 + s_0$ such that $t_1 \rightsquigarrow s'$ and $t \rightsquigarrow \sum_i \lambda_i t_i$, we have:

$$\begin{array}{ccc} \nearrow & \alpha t + \beta s' + s_0 & \rightsquigarrow \sum_i \alpha \lambda_i t_i + \beta s' + s_0 \\ s & & \downarrow \\ \searrow & (\alpha \lambda_1 + \beta) t_1 + \sum_{i \neq 1} \alpha \lambda_i t_i + s_0 & \rightsquigarrow (\alpha \lambda_1 + \beta) s' + \sum_{i \neq 1} \alpha \lambda_i t_i + s_0 \end{array}$$

Then, we can, in the following, focus on pairs of rules applied on the same term. The term we focus on is obviously collision-free, by hypothesis and by preservation of collision-freeness by \rightsquigarrow .

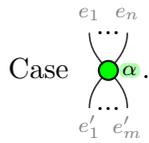
Suppose the two rewrites are applied on tokens at positions e and e' . We may reason using the distance between the two edges.

- The case $d(e, e') = 0$ would imply a collision, which is impossible by collision-freeness;
- if $d(e, e') \geq 3$, the two rules still do not interfere, they commute (up to collisions which do not change the result);
- if $d(e, e') = 2$, there will be common collisions (i.e. collisions between tokens created by each of the diffusions), however, the order of application of the rules will not change the bits in the tokens we will apply a collision on, so the result holds;
- if $d(e, e') = 1$, then the two tokens have to point to the same generator. If they didn't, (e, e') would form a path such that $|P((e, e'), t)| = 2$ which is forbidden by well-formedness. We can then show the property for all generators:

Case $e_0 \cup e_1$.

$$\begin{array}{ccc} (e_0 \downarrow x)(e_1 \downarrow x') & \rightsquigarrow_d & (e_1 \uparrow x)(e_1 \downarrow x') \\ & \downarrow_d & \downarrow_c \\ (e_0 \downarrow x)(e_0 \uparrow x') & \rightsquigarrow_c & \langle x \mid x' \rangle \end{array}$$

Case $e_0 \cap e_1$: similar.



$$\begin{array}{ccc} e^{i\alpha x} \prod_{i \neq 1} (e_i \uparrow x) \prod_i (e'_i \downarrow x) (e'_1 \uparrow x') & \rightsquigarrow_c & \langle x \mid x' \rangle e^{i\alpha x} \prod_{i \neq 1} (e_i \uparrow x) \prod_{i \neq 1} (e'_i \downarrow x) \\ & \downarrow_d & \parallel \\ (e_1 \downarrow x)(e'_1 \uparrow x') & & \langle x \mid x' \rangle e^{i\alpha x'} \prod_{i \neq 1} (e_i \uparrow x') \prod_{i \neq 1} (e'_i \downarrow x') \\ e^{i\alpha x'} \prod_i (e_i \uparrow x') \prod_{i \neq 1} (e'_i \downarrow x) (e_1 \downarrow x) & \nearrow_c & \end{array}$$

Case $\begin{array}{c} e_0 \\ \downarrow \\ \square \\ \uparrow \\ e_1 \end{array}$.

$$\begin{aligned} \frac{1}{\sqrt{2}} \left((-1)^x (e_1 \downarrow x)(e_1 \uparrow x') + (e_1 \downarrow \neg x)(e_1 \uparrow x') \right) &\stackrel{\text{c}}{\rightsquigarrow} \frac{1}{\sqrt{2}} \left((-1)^x \langle x | x' \rangle + \langle \neg x | x' \rangle \right) \\ &\quad \begin{array}{c} \{d \\ (e_0 \downarrow x)(e_1 \uparrow x') \\ \{d \end{array} \quad \begin{array}{c} \\ \\ \parallel \end{array} \\ \frac{1}{\sqrt{2}} \left((-1)^{x'} (e_0 \downarrow x)(e_0 \uparrow x') + (e_0 \downarrow x)(e_0 \uparrow \neg x') \right) &\stackrel{\text{c}}{\rightsquigarrow} \frac{1}{\sqrt{2}} \left((-1)^{x'} \langle x | x' \rangle + \langle x | \neg x' \rangle \right) \end{aligned}$$

□

Using Newmann's Lemma [New42] that states that any terminating and locally confluent rewriting system is confluent, we obtain the confluence of our rewriting system:

Corollary 5.3.19 (Confluence). *Let D be a ZX-diagram. The rewrite system \rightsquigarrow is confluent for well-formed and cycle-balanced token states.*

Corollary 5.3.20 (Uniqueness of Normal Forms). *Let us consider a ZX-diagram D . A well-formed and cycle-balanced token state admits a unique normal form under the rewrite system \rightsquigarrow .*

5.3.3. Semantics and Structure of Normal Forms

In this section, we discuss the structure of normal forms, and relate the system to the standard interpretation presented in Section 2.1.

Proposition 5.3.21 (Single-Token Input). *Let $D : n \rightarrow m$ be a connected ZX-diagram with $\mathcal{I}(D) = [a_i]_{0 < i \leq n}$ and $\mathcal{O}(D) = [b_i]_{0 < i \leq m}$, $0 < k \leq n$ and $x \in \{0, 1\}$, such that:*

$$\llbracket D \rrbracket \circ (id_{k-1} \otimes |x\rangle \otimes id_{n-k}) = \sum_{q=1}^{2^{m+n-1}} \lambda_q |y_{1,q}, \dots, y_{m,q}\rangle \langle x_{1,q}, \dots, x_{k-1,q}, x_{k+1,q}, \dots, x_{n,q}|$$

Then:

$$(a_k \downarrow x) \rightsquigarrow^* \sum_{q=1}^{2^{m+n-1}} \lambda_q \prod_i (b_i \downarrow y_{i,q}) \prod_{i \neq k} (a_i \uparrow x_{i,q})$$

Proof. Let us first notice that, using the map/state duality, we have

$$(a_k \downarrow x) \rightsquigarrow^* \sum_{q=1}^{2^{m+n-1}} \lambda_q \prod_i (b_i \downarrow y_{i,q}) \prod_{i \neq k} (a_i \uparrow x_{i,q})$$

whereby induction hypothesis

$$\llbracket D' \rrbracket |x\rangle = \sum_{q=1}^{2^m} \lambda_q |y_{1,q}, \dots, y_{m,q}\rangle$$

and

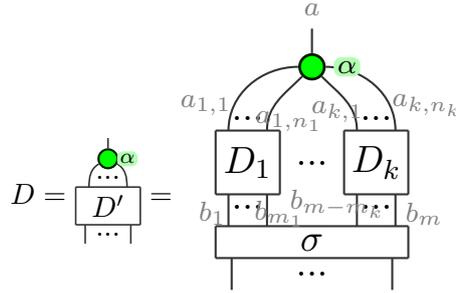
$$\llbracket D' \rrbracket |\neg x\rangle = \sum_{q=1}^{2^m} \lambda'_q |y_{1,q}, \dots, y_{m,q}\rangle$$

so:

$$\begin{aligned} \llbracket D \rrbracket |x\rangle &= \llbracket D' \circ H \rrbracket |x\rangle = \llbracket D' \rrbracket \circ \llbracket H \rrbracket |x\rangle = \llbracket D' \rrbracket \circ \left(\frac{(-1)^x}{\sqrt{2}} |x\rangle + \frac{1}{\sqrt{2}} |\neg x\rangle \right) \\ &= \frac{(-1)^x}{\sqrt{2}} \llbracket D' \rrbracket |x\rangle + \frac{1}{\sqrt{2}} \llbracket D' \rrbracket |\neg x\rangle = \sum_{q=1}^{2^m} \frac{\lambda'_q + (-1)^x \lambda_q}{\sqrt{2}} |y_{1,q}, \dots, y_{m,q}\rangle \end{aligned}$$

which is the expected result.

Now, if $D = \begin{array}{c} \bullet \\ \vdots \\ \boxed{D'} \\ \vdots \\ | \end{array}$, we can decompose D' in its connected components:



with D_i connected. Then:

$$\begin{aligned} (a \downarrow x) &\rightsquigarrow e^{i\alpha x} \prod_{\ell} \prod_i (a_{\ell,i} \downarrow x) \\ &\rightsquigarrow^* e^{i\alpha x} \prod_{\ell} \left(\sum_{q=1}^{2^{m_{\ell}+n_{\ell}-1}} \lambda_{q,\ell} \prod_{i \neq 1} (a_{\ell,i} \downarrow x) (a_{\ell,i} \uparrow x_{\ell,i,q}) \prod_i (b_{\ell,i} \downarrow y_{\ell,i,q}) \right) \\ &\rightsquigarrow^* e^{i\alpha x} \prod_{\ell} \left(\sum_{q=1}^{2^{m_{\ell}+n_{\ell}-1}} \lambda_{q,\ell} \delta_{x, x_{\ell,i,q}} \prod_i (b_{\ell,i} \downarrow y_{\ell,i,q}) \right) \\ &= e^{i\alpha x} \prod_{\ell} \left(\sum_{q=1}^{2^{m_{\ell}}} \lambda'_{q,\ell} \prod_i (b_{\ell,i} \downarrow y_{\ell,i,q}) \right) \end{aligned}$$

$$\begin{aligned}
 &= e^{i\alpha x} \sum_{q_1=1}^{2^{m_1}} \dots \sum_{q_k=1}^{2^{m_k}} \lambda'_{q_1,1} \dots \lambda'_{q_k,k} \prod_i (b_{1,i} \downarrow y_{1,i,q_1}) \dots \prod_i (b_{k,i} \downarrow y_{k,i,q_k}) \\
 &= \sum_{q=1}^{2^m} \lambda'_q \prod_i (b_i \downarrow y_{i,q})
 \end{aligned}$$

where the first is the diffusion through a Z-spider, and the second set of rewrites is the induction hypothesis applied to each connected component.

$$\begin{aligned}
 \llbracket D \rrbracket |x\rangle &= \llbracket (D_1 \otimes \dots \otimes D_k) \circ Z_k^1(\alpha) \rrbracket |x\rangle = (\llbracket D_1 \rrbracket \otimes \dots \otimes \llbracket D_k \rrbracket) \circ \llbracket Z_k^1(\alpha) \rrbracket |x\rangle \\
 &= e^{i\alpha x} (\llbracket D_1 \rrbracket \otimes \dots \otimes \llbracket D_k \rrbracket) \circ |x, \dots, x\rangle = e^{i\alpha x} \llbracket D_1 \rrbracket |x, \dots, x\rangle \otimes \dots \otimes \llbracket D_k \rrbracket |x, \dots, x\rangle \\
 &= e^{i\alpha x} \left(\sum_{q_1}^{2^{m_1+n_1-1}} \lambda_{q_1,1} |y_{1,1,q_1}, \dots, y_{1,m_1,q_1}\rangle \langle x_{1,2,q_1}, \dots, x_{1,n_1,q_1} | x, \dots, x \rangle \right) \otimes \\
 &\quad \dots \otimes \left(\sum_{q_k}^{2^{m_k+n_k-1}} \lambda_{q_k,k} |y_{k,1,q_k}, \dots, y_{k,m_k,q_k}\rangle \langle x_{k,2,q_k}, \dots, x_{k,n_k,q_k} | x, \dots, x \rangle \right) \\
 &= e^{i\alpha x} \left(\sum_{q_1}^{2^{m_1+n_1-1}} \lambda_{q_1,1} \prod_i \delta_{x,x_{1,i,q_1}} |y_{1,1,q_1}, \dots, y_{1,m_1,q_1}\rangle \right) \otimes \\
 &\quad \dots \otimes \left(\sum_{q_k}^{2^{m_k+n_k-1}} \lambda_{q_k,k} \prod_i \delta_{x,x_{k,i,q_k}} |y_{k,1,q_k}, \dots, y_{k,m_k,q_k}\rangle \right) \\
 &= e^{i\alpha x} \left(\sum_{q_1}^{2^{m_1}} \lambda'_{q_1,1} |y_{1,1,q_1}, \dots, y_{1,m_1,q_1}\rangle \right) \otimes \dots \otimes \left(\sum_{q_k}^{2^{m_k}} \lambda'_{q_k,k} |y_{k,1,q_k}, \dots, y_{k,m_k,q_k}\rangle \right) \\
 &= \sum_{q=1}^{2^m} \lambda'_q |y_{1,q}, \dots, y_{m,q}\rangle
 \end{aligned}$$

where the third line is obtained by induction hypothesis, and all λ' match the ones obtained from the rewrite of token states. \square

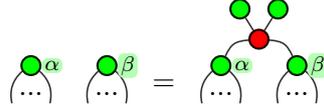
This proposition conveys the fact that dropping a single token in state x on wire a_k gives the same semantics as the one obtained from the standard interpretation on the ZX-diagram, with wire a_k connected to the state $|x\rangle$.

Proposition 5.3.21 can be made more general. However, we first need the following result on ZX-diagrams:

Lemma 5.3.22 (Universality of Connected ZX-Diagrams). *Let $f : \mathbb{C}^{2^n} \rightarrow \mathbb{C}^{2^m}$. There exists a connected ZX-diagram $D_f : n \rightarrow m$ such that $\llbracket D_f \rrbracket = f$.*

Proof. There exist several methods to build a diagram D_f such that $\llbracket D_f \rrbracket = f$, using the universality of quantum circuits together with the map/state duality [CD11], or using normal forms [JPV19]. The novelty here is that the diagram should be connected. This problem can be fairly simply dealt with:

Suppose that we have such a D_f that has several connected components. We can turn it into an equivalent diagram that is connected. Let us consider two disconnected components of D_f . Each of these disconnected components either has at least one wire, or is one of $\{\bullet\alpha, \bullet\beta\}$. In either case, we can use the rules of ZX ((I_g) or (H)) to force the existence of a green node. These green nodes in each of the connected components can be “joined” together like this:



It is hence possible to connect every different connected components of a diagram in a way that preserves the semantics. \square

Proposition 5.3.23 (Multi-Token Input). *Let D be a connected ZX-diagram with $\mathcal{I}(D) = [a_i]_{1 \leq i \leq n}$ and $\mathcal{O}(D) = [b_i]_{1 \leq i \leq m}$; with $n \geq 1$.*

$$\text{If: } \llbracket D \rrbracket \circ \left(\sum_{q=1}^{2^n} \lambda_q |x_{1,q}, \dots, x_{n,q}\rangle \right) = \sum_{q=1}^{2^m} \lambda'_q |y_{1,q}, \dots, y_{m,q}\rangle$$

$$\text{then: } \sum_{q=1}^{2^n} \lambda_q \prod_{i=1}^n (a_i \downarrow x_{i,q}) \rightsquigarrow^* \sum_{q=1}^{2^m} \lambda'_q \prod_{i=1}^m (b_i \downarrow y_{i,q})$$

Proof. Using Lemma 5.3.22, there exists a connected ZX-diagram D' with $\mathcal{I}(D') = [a']$ and such that $\llbracket D' \rrbracket |0\rangle = \sum_{q=1}^{2^n} \lambda_q |x_{1,q}, \dots, x_{n,q}\rangle$. Consider now a derivation from the token state $(a' \downarrow 0)$ in $D \circ D'$:

$$\begin{array}{c} a' \\ \downarrow \\ \boxed{D'} \\ \begin{array}{c} \dots \\ a_1 \dots a_n \end{array} \\ \downarrow \\ \boxed{D} \\ \begin{array}{c} \dots \\ b_1 \dots b_m \end{array} \end{array} \parallel \begin{array}{l} (a' \downarrow 0) \rightsquigarrow^* \sum_{q=1}^{2^n} \lambda_q \prod_{i=1}^n (a_i \downarrow x_{i,q}) \\ \text{and} \\ (a' \downarrow 0) \rightsquigarrow^* \sum_{q=1}^{2^m} \lambda'_q \prod_{i=1}^m (b_i \downarrow y_{i,q}) \end{array}$$

The first run comes from Proposition 5.3.21 on D' which is connected. The second run results from Proposition 5.3.21 on $D \circ D'$ which is also connected. The proposition also gives us that:

$$\llbracket D \rrbracket \circ \left(\sum_{q=1}^{2^n} \lambda_q |x_{1,q}, \dots, x_{n,q}\rangle \right) = \llbracket D \rrbracket \circ \llbracket D' \rrbracket |0\rangle = \llbracket D \circ D' \rrbracket |0\rangle = \sum_{q=1}^{2^m} \lambda'_q |y_{1,q}, \dots, y_{m,q}\rangle$$

Finally, by confluence in $D \circ D'$, we get $\sum_{q=1}^{2^n} \lambda_q \prod_{i=1}^n (a_i \downarrow x_{i,q}) \rightsquigarrow^* \sum_{q=1}^{2^m} \lambda'_q \prod_{i=1}^m (b_i \downarrow y_{i,q})$ in D . \square

Example 5.3.24 (CNOT). *In the ZX-Calculus, the CNOT-gate (up to some scalar)*

can be constructed as follows:

$$\left[\begin{array}{c} a_1 \\ \text{---} \text{---} \text{---} \\ \text{---} \text{---} \text{---} \\ \text{---} \text{---} \text{---} \\ b_1 \end{array} \right] = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

On classical inputs, this gate applies the NOT-gate on the second bit if and only if the first bit is at 1. Therefore, if we apply the state $|10\rangle$ to it, we get $\frac{1}{\sqrt{2}}|11\rangle$.

We demonstrate how the token machine can be used to get this result. Following Proposition 5.3.23, we start by initializing the Token Machine in the token state $(a_1 \downarrow 1)(a_2 \downarrow 0)$, matching the input state $|10\rangle$.

We underline each step that is being rewritten, and take the liberty to sometimes do several rewrites in parallel at the same time.

$$\begin{aligned} \underline{(a_1 \downarrow 1)}(a_2 \downarrow 0) &\rightsquigarrow_d \underline{(b_1 \downarrow 1)}(e_1 \downarrow 1)\underline{(a_2 \downarrow 0)} \rightsquigarrow_d (b_1 \downarrow 1)(e_1 \downarrow 1)\frac{1}{\sqrt{2}}\left(\underline{(e_3 \downarrow 0)} + \underline{(e_3 \downarrow 1)}\right) \\ &\rightsquigarrow_d \frac{1}{\sqrt{2}}(b_1 \downarrow 1)\underline{(e_1 \downarrow 1)}\left(\underline{(e_2 \uparrow 0)}(e_4 \downarrow 0) + \underline{(e_2 \uparrow 1)}(e_4 \downarrow 1)\right) \\ &\rightsquigarrow_d \frac{1}{2}(b_1 \downarrow 1)\left(\underline{(e_2 \downarrow 0)} - \underline{(e_2 \downarrow 1)}\right)\left(\underline{(e_2 \uparrow 0)}(e_4 \downarrow 0) + \underline{(e_2 \uparrow 1)}(e_4 \downarrow 1)\right) \\ &\rightsquigarrow_c^2 \frac{1}{2}(b_1 \downarrow 1)\left(\underline{(e_4 \downarrow 0)} + \left(\underline{(e_2 \downarrow 0)} - \underline{(e_2 \downarrow 1)}\right)\underline{(e_2 \uparrow 1)}(e_4 \downarrow 1)\right) \\ &\rightsquigarrow_c^2 \frac{1}{2}(b_1 \downarrow 1)\left(\underline{(e_4 \downarrow 0)} - \underline{(e_4 \downarrow 1)}\right) \\ &\rightsquigarrow_d \frac{1}{2\sqrt{2}}(b_1 \downarrow 1)\left(\underline{(b_2 \downarrow 0)} + \underline{(b_2 \downarrow 1)} - \underline{(b_2 \downarrow 0)} + \underline{(b_2 \downarrow 1)}\right) \\ &= \frac{1}{\sqrt{2}}(b_1 \downarrow 1)(b_2 \downarrow 1) \end{aligned}$$

The final token state corresponds to $\frac{1}{\sqrt{2}}|11\rangle$, as described by Proposition 5.3.23. Notice that during the run, all invariants presented before holds and that due to confluence we could have rewritten the tokens in any order and still obtain the same result.

Proposition 5.3.23 is a direct generalization of Proposition 5.3.21. It shows we can compute the output of a diagram provided a particular input state. We can also recover the semantics of the whole operator by initializing the starting token state in a particular configuration.

Theorem 5.3.25 (Arbitrary Wire Initialisation). *Let D be a connected ZX-diagram, with $\mathcal{I}(D) = [a_i]_{1 \leq i \leq n}$, $\mathcal{O}(D) = [b_i]_{1 \leq i \leq m}$, and $e \in \mathcal{E}(D) \neq \emptyset$ such that $(e \downarrow x)(e \uparrow x) \rightsquigarrow^* t_x$ for $x \in \{0, 1\}$ with t_x terminal (the rewriting terminates by Corollary 5.3.20). Then:*

$$\llbracket D \rrbracket = \sum_{q=1}^{2^{m+n}} \lambda_q |y_{1,q} \dots y_{m,q}\rangle \langle x_{1,q} \dots x_{n,q}| \implies t_0 + t_1 = \sum_{q=1}^{2^{m+n}} \lambda_q \prod_i (b_i \downarrow y_{i,q}) \prod_i (a_i \uparrow x_{i,q})$$

Proof. First, let us single out e in the diagram $D = \begin{array}{c} | \dots | \\ \boxed{D_1} \\ | \dots | \\ \boxed{D_2} \\ | \dots | \\ b_i \end{array} e$. We can build a second

diagram by cutting e in half and seeing each piece of wire as an input and an output:

$\begin{array}{c} | \dots | \\ \boxed{D'} \\ | \dots | \\ b_i \end{array} := \begin{array}{c} | \dots | \\ \boxed{D_1} \\ | \dots | \\ \boxed{D_2} \\ | \dots | \\ b_i \end{array} \begin{array}{l} e_0 \\ e_1 \end{array}$. We can easily see that a rewriting of the token states $(e \downarrow 0)(e \uparrow 0)$

and $(e \downarrow 1)(e \uparrow 1)$ in D corresponds step by step to a rewriting of the token states $(e_0 \downarrow 0)(e_1 \uparrow 0)$ and $(e_0 \downarrow 1)(e_1 \uparrow 1)$ in D' . We can then focus on D' , whose interpretation is taken to be

$$\llbracket D' \rrbracket = \sum_{q=1}^{2^{m+n+2}} \lambda'_q |y'_{1,q}, \dots, y'_{m+1,q}\rangle \langle x'_{1,q}, \dots, x'_{n+1,q}|$$

such that

$$(id^{\otimes m} \otimes \langle 0|) \circ \llbracket D' \rrbracket \circ (id^{\otimes n} \otimes |0\rangle) + (id^{\otimes m} \otimes \langle 1|) \circ \llbracket D' \rrbracket \circ (id^{\otimes n} \otimes |1\rangle) = \llbracket D \rrbracket$$

from which we get:

$$\begin{aligned} \llbracket D \rrbracket &= \sum_{q=1}^{2^{m+n+2}} \lambda'_q \delta_{0,y'_{m+1,q}} \delta_{0,x'_{n+1,q}} |y'_{1,q}, \dots, y'_{m,q}\rangle \langle x'_{1,q}, \dots, x'_{n,q}| \\ &\quad + \sum_{q=1}^{2^{m+n+2}} \lambda'_q \delta_{1,y'_{m+1,q}} \delta_{1,x'_{n+1,q}} |y'_{1,q}, \dots, y'_{m,q}\rangle \langle x'_{1,q}, \dots, x'_{n,q}| \end{aligned}$$

We now have to consider two cases:

- D' is still connected: By Proposition 5.3.21, for $x \in \{0, 1\}$:

$$\begin{aligned} (e_0 \downarrow x)(e_1 \uparrow x) &\rightsquigarrow^* \sum_{q=1}^{2^{m+n+2}} \lambda'_q \delta_{x,x'_{n+1,q}} \prod_i (a_i \uparrow x'_{i,q}) \prod_i (b_i \downarrow y'_{i,q}) (e_1 \downarrow y'_{m+1,q}) (e_1 \uparrow x) \\ &\rightsquigarrow \sum_{q=1}^{2^{m+n+2}} \lambda'_q \delta_{x,y'_{m+1,q}} \delta_{x,x'_{n+1,q}} \prod_i (a_i \uparrow x'_{i,q}) \prod_i (b_i \downarrow y'_{i,q}) \end{aligned}$$

We hence have

$$(e_0 \downarrow 0)(e_1 \uparrow 0) \rightsquigarrow^* t_0 = \sum_{q=1}^{2^{m+n+2}} \lambda'_q \delta_{0,y'_{m+1,q}} \delta_{0,x'_{n+1,q}} \prod_i (a_i \uparrow x'_{i,q}) \prod_i (b_i \downarrow y'_{i,q})$$

$$(e_0 \downarrow 1)(e_1 \uparrow 1) \rightsquigarrow^* t_1 = \sum_{q=1}^{2^{m+n+2}} \lambda'_q \delta_{1, y'_{m+1, q}} \delta_{1, x'_{n+1, q}} \prod_i (a_i \uparrow x'_{i, q}) \prod_i (b_i \downarrow y'_{i, q})$$

so $t_0 + t_1$ corresponds to the interpretation of D .

- D' is now disconnected: Since D was connected, the two connected components of D were connected through e . Hence, D' only has two connected components, one connected to e_0 and the other to e_1 . By applying Proposition 5.3.21 to both connected components, we get the desired result. \square

Example 5.3.26. Consider again the diagram from Example 5.3.24 and initialize any wire e of the diagram in the state $(e \downarrow 0)(e \uparrow 0) + (e \downarrow 1)(e \uparrow 1)$ and apply the rewriting as in Theorem 5.3.25 we end up with the final token state $\frac{1}{\sqrt{2}} \left((a_1 \uparrow 0)(a_2 \uparrow 0)(b_1 \downarrow 0)(b_2 \downarrow 0) + (a_1 \uparrow 0)(a_2 \uparrow 1)(b_1 \downarrow 0)(b_2 \downarrow 1) + (a_1 \uparrow 1)(a_2 \uparrow 0)(b_1 \downarrow 1)(b_2 \downarrow 1) + (a_1 \uparrow 1)(a_2 \uparrow 1)(b_1 \downarrow 1)(b_2 \downarrow 0) \right)$ which corresponds to the matrix $\frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$, that is the one obtained from the standard interpretation.

Notice that while technically there is a collision happening on the initial token state given by Theorem 5.3.25, we do not apply it, intuitively this is saying that while the tokens are on the same wire they have already crossed each others, so they cannot collide.

5.3.4. Discussions

At this point, it is legitimate to wonder about the benefits of the token machine over the standard interpretation for computing the semantics of a diagram. Let us first notice that when computing the semantics of a diagram à la Theorem 5.3.25, we get in the token state one term per non-null entry in the associated matrix (the one obtained by the standard interpretation).

We can already see that the token-based interpretation can be interesting if the matrix is sparse, the textbook case being Z_n^n whose standard interpretation requires a $2^n \times 2^n$ matrix, while the token-based interpretation only requires two terms (each with $2n$ tokens).

Secondly, we can notice that we can "mimic" the standard interpretation with the token machine. Consider a diagram decomposed as a product of slices (tensor product of generators) for the standard interpretation. Then, for the token machine, without going into technical details, we can follow the strategy that consists in moving tokens through the diagram one slice at a time. This essentially computes the matrix associated with each slice and its composition.

The point of the token machine, however, is that it is versatile enough to allow for more original strategies, some of which may have a worst complexity, but also some of which may have a better one.

5.4. Extension to Mixed Processes

The token machine presented in Section 5.3 works for so-called *pure* quantum processes i.e. with no interaction with the environment. To demonstrate how generic our approach is, we show how to adapt it to the natural extension of *mixed* processes, presented in Section 2.1.4. In particular, this allows us to represent quantum measurements.

With respect to what happens to edge labels, notice that every edge in D can be mapped to 2 edges in $\text{CPM}(D)$. We propose that label e induces label e in the first copy, and \bar{e} in the second, e.g., for the identity diagram: $|_{e_0} \mapsto |_{e_0} |_{\bar{e}_0}$

5.4.1. Token Machine for Mixed Processes

We now aim at adapting the token machine to \mathbf{ZX}^{\pm} , the formalism for completely positive maps. To do so we give an additional state to each token to mimic the evolution of two tokens on $\text{CPM}(D)$.

Definition 5.4.1. *Let D be a ZX-diagram. A \pm -token is a quadruplet $(p, d, x, y) \in \mathcal{E}(D) \times \{\downarrow, \uparrow\} \times \{0, 1\} \times \{0, 1\}$. We denote the set of \pm -tokens on D by $\mathbf{tk}^{\pm}(D)$. A \pm -token-state is then a multivariate polynomial over \mathbb{C} , evaluated in $\mathbf{tk}^{\pm}(D)$. We denote the set of \pm -token-states on D by $\mathbf{tkS}^{\pm}(D)$*

In other words, the difference with the previous machine is that tokens here have an additional state (e.g. y in $(e \downarrow x, y)$). The rewrite rules are given in Table 5.3.

It is possible to link this formalism back to the pure token-states, using the existing CPM construction for ZX-diagrams.

We extend this map by $\text{CPM} : \mathbf{tkS}^{\pm}(D) \rightarrow \mathbf{tkS}(\text{CPM}(D))$, defined as:

$$\sum_{q=1}^{2^{m+n}} \lambda_q \prod_j (p_j, d_j, x_{j,q}, y_{j,q}) \mapsto \sum_{q=1}^{2^{m+n}} \lambda_q \prod_j (p_j, d_j, x_{j,q})(\bar{p}_j, d_j, y_{j,q})$$

Since $\text{CPM}(D)$ can be seen as two copies of D where \pm is replaced by \cup , each token in D corresponds to two tokens in $\text{CPM}(D)$, at the same spot but in the two copies of D . The two states x and y represent the states of the two corresponding tokens.

We can then show that this rewriting system is consistent:

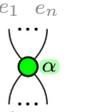
	$(e_0 \downarrow x, y)(e_0 \uparrow x', y') \rightsquigarrow_c \delta_{x,x'} \delta_{y,y'}$	(Collision)
	$(e_b \downarrow x, y) \rightsquigarrow_d (e_{-b} \uparrow x, y)$	(\cup -diffusion)
	$(e_b \uparrow x, y) \rightsquigarrow_d (e_{-b} \downarrow x, y)$	(\cap -diffusion)
	$(e_k \downarrow x, y) \rightsquigarrow_d e^{i\alpha(x-y)} \prod_{j \neq k} (e_j \uparrow x, y) \prod_j (e'_j \downarrow x, y)$	
	$(e'_k \uparrow x, y) \rightsquigarrow_d e^{i\alpha(x-y)} \prod_j (e_j \uparrow x, y) \prod_{j \neq k} (e'_j \downarrow x, y)$	(\odot -Diffusion)
	$(e_0 \downarrow x, y) \rightsquigarrow_d \frac{1}{2} \sum_{z, z' \in \{0,1\}} (-1)^{xz+yz'} (e_1 \downarrow z, z')$	(\square -Diffusion)
	$(e_1 \uparrow x, y) \rightsquigarrow_d \frac{1}{2} \sum_{z, z' \in \{0,1\}} (-1)^{xz+yz'} (e_0 \uparrow z, z')$	
	$(e_0 \downarrow x, y) \rightsquigarrow_d \delta_{x,y}$	(Trace-Out)

 Table 5.3.: The rewrite rules for \rightsquigarrow_{\pm} , where δ is the Kronecker delta.

Theorem 5.4.2. *Let D be a \mathbf{ZX}^{\pm} -diagram, and $t_1, t_2 \in \mathbf{tkS}^{\pm}(D)$. Then whenever $t_1 \rightsquigarrow_{\pm} t_2$ we have $\text{CPM}(t_1) \rightsquigarrow^{\{1,2\}} \text{CPM}(t_2)$.*

Proof. The proof is done by induction on \rightsquigarrow_{\pm} :

- Collision: $t = (e_0 \downarrow x, y)(e_0 \uparrow x', y') \rightsquigarrow_{\pm} \delta_{x,x'} \delta_{y,y'}$.
We get $\text{CPM}(t) = (e_0 \downarrow x)(\bar{e}_0 \downarrow y)(e_0 \uparrow x')(\bar{e}_0 \uparrow y') \rightsquigarrow \delta_{x,x'} \delta_{y,y'}$
- Cup (Cap being similar): $t = (e_b \downarrow x, y) \rightsquigarrow_{\pm} (e_{-b} \uparrow x, y)$.
We get $\text{CPM}(t) = (e_b \downarrow x)(\bar{e}_b \downarrow y) \rightsquigarrow (e_{-b} \uparrow x)(\bar{e}_{-b} \uparrow y)$.
- $Z_m^n(\alpha)$: $t = (e_k \downarrow x, y) \rightsquigarrow_{\pm} e^{i\alpha(x-y)} \prod_{j \neq k} (e_j \uparrow x, y) \prod_j (e'_j \downarrow x, y) = t'$
then we get

$$\begin{aligned} \text{CPM}(t) &= (e_k \downarrow x)(\bar{e}_k \downarrow y) \\ &\rightsquigarrow^2 e^{i\alpha x} \prod_{j \neq k} (e_j \uparrow x) \prod_j (e'_j \downarrow x) e^{i(-\alpha)y} \prod_{j \neq k} (\bar{e}_j \uparrow y) \prod_j (\bar{e}'_j \downarrow y) \\ &= \text{CPM}(t') \end{aligned}$$

- Hadamard: $t = (e_0 \downarrow x, y) \rightsquigarrow_{\pm} \frac{1}{2} \sum_{z, z' \in \{0,1\}} (-1)^{xz+yz'} (e_1 \downarrow z, z') = t'$, then

$$\text{CPM}(t) = (e_0 \downarrow x)(\bar{e}_0 \downarrow y)$$

$$\begin{aligned}
 & \rightsquigarrow_d^2 [(-1)^x \frac{1}{\sqrt{2}}(e_1 \downarrow x) + \frac{1}{\sqrt{2}}(e_1 \downarrow \neg x)][(-1)^y \frac{1}{\sqrt{2}}(\bar{e}_1 \downarrow y) + \frac{1}{\sqrt{2}}(\bar{e}_1 \downarrow \neg y)] \\
 &= \frac{1}{2}((-1)^{x+y}(e_1 \downarrow x)(\bar{e}_1 \downarrow y) + (-1)^x(e_1 \downarrow x)(\bar{e}_1 \downarrow \neg y) \\
 &+ (-1)^y(e_1 \downarrow \neg x)(\bar{e}_1 \downarrow y) + (e_1 \downarrow \neg x)(\bar{e}_1 \downarrow \neg y)) \\
 &= \text{CPM}(t')
 \end{aligned}$$

- Ground: $t = (e_0 \downarrow x, y) \rightsquigarrow_{\pm} \delta_{x,y}$ then $\text{CPM}(t) = (e_0 \downarrow x)(\bar{e}_0 \downarrow y)$. Remember than in $\text{CPM}(D)$ the Ground is translated as a Cup we get one diffusion and one collision rule: $\text{CPM}(t) \rightsquigarrow (\bar{e}_0 \uparrow x)(\bar{e}_0 \downarrow y) \rightsquigarrow \delta_{x,y}$. \square

The notions of polarity, well-formedness and cycle-balancedness can be adapted, and we get strong normalization (Theorem 5.3.16), confluence (Corollary 5.3.19), and uniqueness of normal forms (Corollary 5.3.20) for well-formed and cycle-balanced token states.

5.5. Variations of the Token Machine

5.5.1. Pulse Rewriting

So far, the Token Machines that we presented required us to specify an initial state in order to compute the semantics of a ZX-Diagram. It is nonetheless possible to consider another version of the rewriting rules, called the *pulse-semantics* in which a node of the diagram will *pulse* and emit tokens in all of its input and output edges, directly reflecting its matrix semantics as in Theorem 5.3.25. In order to properly compute the semantics of the whole diagram, each node needs to pulse exactly once, in any order, and then one can simply apply the collisions. As each generator is forced to pulse, there is no need for the diagram to be connected. We may however have fringe cases, with connected components in the diagram that have no generators that pulse. To remedy this, it suffices to recall that the identity is nothing but  = , and make this last generator pulse in these cases. Notice that it is technically important to break the wire's name e from the left-hand-side in two $e_1 e_2$ on each side of the green node, so that the tokens obtained from the pulse are not removed by a collision. The rules for the Pulse Token Machine are given in Table 5.4.

The pulse token machine is pretty straightforward: following the idea from Theorem 5.3.25 one can notice that the token state obtained by the pulse of a generator is exactly the same as the standard interpretation of said generator.

The main results are obtained trivially: the pulse rewriting strategy enjoys termination (as each generator pulses only once), confluence (as pulse and collisions do not interact), that we reach the expected normal form with no token inside the diagram and only one token going up (resp. down) on any input (resp. output) wire (after each pulse each

generator. For instance, in the $\left[\begin{array}{c} \cdots \\ \bullet \\ \cdots \end{array} \right]$ -Diffusion]-rule the product $\prod_i (e_i \downarrow x_i)$ rewrites into $\delta_{x_1, \dots, x_n} e^{i\alpha x_1} \prod_i (e'_i \downarrow x_i)$. This notion of synchronicity is to be contrasted with [Dal17] where tokens have to wait for all other incoming tokens to reach the gate before going through it.

5.6. Sum Over Path Semantics

A serious drawback of the previous token machines is that the token state grows exponentially quickly in the number of nodes in the diagram. A more compact representation (linear in the size of the diagram as we will see in Prop. 5.6.7) can be obtained by adapting the concept of sums-over-paths (SOP) [Amy18] to our machine. This can be obtained naturally, as strong links between ZX-Calculus and SOP morphisms were already shown to exist [LWK20; Vil20]. Intuitively, SOP will allow us to manipulate token states in a symbolic way, where for instance $(e \downarrow 0) + (e \downarrow 1)$ will be represented by $(e \downarrow y)$. While the development of the sums-over-paths token machine was mostly done by Renaud Vilmart, we put it here for comprehensiveness, with added proof of Theorem 5.4.2 and Theorem 5.6.9 that were missing in the original paper.

5.6.1. SOP Token Machine for Pure Operators

Definition 5.6.1. *Let D be a ZX-diagram. A **SOP-token** is a triplet (p, d, B) belonging to $\mathcal{E}(D) \times \{\downarrow, \uparrow\} \times \mathbb{F}_2[\vec{y}]$ where $\vec{y} := (y_i)_{0 \leq i < n}$ are n variables from a set of variables \mathcal{V} ; and where $\mathbb{F}_2 := \mathbb{Z}/2\mathbb{Z}$. We denote the set of **SOP-tokens** on D with variables \vec{y} by $\mathbf{tk}_{\mathbf{SOP}}(D)[\vec{y}]$. A **SOP-token-state** is a quadruplet:*

$$(s, \vec{y}, P, \{t_j\}_{0 \leq j < p}) \in \mathbb{R} \times \mathcal{V}^n \times \mathbb{R}[\vec{y}] / (1, \{y_i^2 - y_i\}_{0 \leq i < n}) \times \mathbf{tk}_{\mathbf{SOP}}(D)[\vec{y}]^p$$

where $\mathbb{R}[\vec{y}] / (1, \{y_i^2 - y_i\}_{0 \leq i < n})$ is the set of real-valued multivariate polynomials (whose variables are \vec{y}), modulo 1 and modulo $(y_i^2 - y_i)$ for all variables y_i .

For any valuation of \vec{y} , $2\pi P(\vec{y})$ represents an angle, hence P is taken modulo 1. Since each y_i is a boolean variable, we can consider $y_i^2 - y_i = 0$. To better reflect what this quadruplet represents, we usually write it as:

$$s \sum_{\vec{y}} e^{2i\pi P(\vec{y})} (p_0, d_0, B_0(\vec{y})) \dots (p_{m-1}, d_{m-1}, B_{m-1}(\vec{y}))$$

We denote the set of **SOP-token-states** on D by $\mathbf{tkS}_{\mathbf{SOP}}(D)$.

Example 5.6.2. *Let $D = \begin{array}{c} e_0 \\ \downarrow \\ \uparrow \\ e_1 \end{array}$, then $\frac{1}{\sqrt{2}} \sum_{y_0, y_1} e^{2i\pi \frac{y_0 y_1}{2}} (e_0 \uparrow y_0)(e_1 \downarrow y_1) \in \mathbf{tkS}_{\mathbf{SOP}}(D)$.*

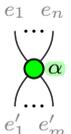
$e_0 \downarrow$	$(e_0 \downarrow B)(e_0 \uparrow B') \rightsquigarrow_c \frac{1}{2} \sum_z e^{2i\pi \frac{z}{2} (\widehat{B \oplus B'})}$	(Collision)
$e_0 \cup e_1$	$(e_b \downarrow B) \rightsquigarrow_d (e_{-b} \uparrow B)$	(\cup -diffusion)
$e_0 \cap e_1$	$(e_b \uparrow B) \rightsquigarrow_d (e_{-b} \downarrow B)$	(\cap -diffusion)
	$(e_k \downarrow B) \rightsquigarrow_d e^{2i\pi (\frac{\alpha}{2\pi} \widehat{B})} \prod_{j \neq k} (e_j \uparrow B) \prod (e'_j \downarrow B)$	
$e'_k \uparrow$	$(e'_k \uparrow B) \rightsquigarrow_d e^{2i\pi (\frac{\alpha}{2\pi} \widehat{B})} \prod_{j \neq k} (e_j \uparrow B) \prod_{j \neq k} (e'_j \downarrow B)$	(\odot -Diffusion)
$e_0 \downarrow$	$(e_0 \downarrow B) \rightsquigarrow_d \frac{1}{\sqrt{2}} \sum_z e^{2i\pi (\frac{z}{2} \widehat{B})} (e_1 \downarrow z)$	
	$(e_1 \uparrow B) \rightsquigarrow_d \frac{1}{\sqrt{2}} \sum_z e^{2i\pi (\frac{z}{2} \widehat{B})} (e_0 \uparrow z)$	(\boxplus -Diffusion)

 Table 5.5.: Rewrite rules for $\rightsquigarrow_{\text{SOP}}$.

We can link this formalism back to the previous one, by defining a map that associates any **SOP**-token-state to a “usual” token-state. This map simply evaluates the term by having all its variables span $\{0, 1\}$:

Definition 5.6.3. We define $[\cdot]^{\text{tk}} : \text{tkS}_{\text{SOP}}(D) \rightarrow \text{tkS}(D)$ by:

$$\left[s \sum_{\vec{y}} e^{2i\pi P(\vec{y})} \prod_j (p_j, d_j, B_j(\vec{y})) \right]^{\text{tk}} := s \sum_{\vec{y} \in \{0,1\}^n} e^{2i\pi P(\vec{y})} \prod_j (p_j, d_j, B_j(\vec{y}))$$

Notice that therefore, the gain in size of the token state will be purely on a *representation* point of view. Computing the actual matrices encoded by the token state will still require an exponential growth in the number of tokens.

Example 5.6.4.

$$\left[\frac{1}{\sqrt{2}} \sum_{y_0, y_1} e^{2i\pi \frac{y_0 y_1}{2}} (e_0 \uparrow y_0)(e_1 \downarrow y_1) \right]^{\text{tk}} = \frac{1}{\sqrt{2}} \begin{pmatrix} (e_0 \uparrow 0)(e_1 \downarrow 0) + (e_0 \uparrow 1)(e_1 \downarrow 0) \\ +(e_0 \uparrow 0)(e_1 \downarrow 1) - (e_0 \uparrow 1)(e_1 \downarrow 1) \end{pmatrix}$$

Once again, the rule for a red node can be obtained from the previous rewrite rules and the ZX-rule: $\begin{pmatrix} \dots \\ \bullet \\ \dots \end{pmatrix} := \begin{pmatrix} \dots \\ \bullet \\ \dots \end{pmatrix}$. For reference, see Table 5.6.

We give the adapted set of rewrite rules for our **SOP**-token-machine in Table 5.5. In the rewrite rules of our token machine, we have to map elements of $\mathbb{F}_2[\vec{y}]$ to elements of $\mathbb{R}[\vec{y}]/(1, \{y_i^2 - y_i\})$ for the Boolean polynomials to be sent to the phase polynomial. The

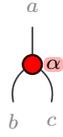
	$(a \downarrow x) \rightarrow \frac{1}{\sqrt{2}} \sum_y e^{2i\pi(\frac{xy}{2} + \frac{\alpha}{2\pi}y)}$
	$(a \downarrow x) \rightarrow \frac{1}{2\sqrt{2}} \sum_{y_i} e^{2i\pi(\frac{xy_1}{2} + \frac{\alpha}{2\pi}y_1 + \frac{y_1y_2}{2} + \frac{y_1y_3}{2})} (b \downarrow y_2)(c \downarrow y_3)$

Table 5.6.: Samples of asynchronous token-state evolution for red spiders with SOP Token Machine

map $\widehat{(\cdot)} : \mathbb{F}_2[\vec{y}] \rightarrow \mathbb{R}[\vec{y}]/1(1, \{y_i^2 - y_i\})$ that does this is defined as:

$$\widehat{B \oplus B'} = \widehat{B} + \widehat{B'} - 2\widehat{BB'} \quad \widehat{BB'} = \widehat{B}\widehat{B'} \quad \widehat{y}_i = y_i \quad \widehat{0} = 0 \quad \widehat{1} = 1$$

The provided rewrite rules do not give the full picture, for simplicity. If a rule gives $(e, d, b) \rightsquigarrow_{\mathbf{SOP}} s' \sum_{\vec{y}'} e^{2i\pi P'} \prod_j (e'_j, d'_j, b'_j)$, we have to apply it to a full **SOP**-token-state as follows:

$$s \sum_{\vec{y}} e^{2i\pi P} (e, d, b) \prod_j (e_j, d_j, b_j) \rightsquigarrow s s' \sum_{\vec{y}, \vec{y}'} e^{2i\pi(P+P')} \prod_j (e'_j, d'_j, b'_j) \prod_j (e_j, d_j, b_j).$$

Just as before, the rewrite system is defined by first applying a diffusion rule then all possible collision rules.

This set of rules mimics the previous one for **SOP**-token-states, except that it “synchronizes” rewrites on all the terms at once (but not on all tokens).

Example 5.6.5. *Let us compare the behaviour of the token machine from Section 5.3 to the SOP machine. We send a sum of tokens in states 0 and 1 down the wire a in the diagram . In the former machine, this leads to*

$$(a \downarrow 0) + (a \downarrow 1) \rightsquigarrow (b \downarrow 0)(c \downarrow 0) + (a \downarrow 1) \rightsquigarrow (b \downarrow 0)(c \downarrow 0) + (b \downarrow 1)(c \downarrow 1)$$

while in the latter: $\sum_y (a \downarrow y) \rightsquigarrow_{\mathbf{SOP}} \sum_y (b \downarrow y)(c \downarrow y)$.

In both cases the result is the same when interpreted as usual token states. We notice that the $\rightsquigarrow_{\mathbf{SOP}}$ token machine only takes one step compared to the standard one, which leads to the following proposition:

Proposition 5.6.6. *For any $D \in \mathbf{ZX}$ and $s, s' \in \mathbf{tkS}_{\mathbf{SOP}}(D)$, whenever $s \rightsquigarrow_{\mathbf{SOP}} s'$ we have $[s]^{\mathbf{tk}} \rightsquigarrow^* [s']^{\mathbf{tk}}$.*

Proof. By induction on $\rightsquigarrow_{\mathbf{SOP}}$:

- Collision: $s = (e_0 \downarrow B)(e_0 \uparrow B') \rightsquigarrow_{\text{sop}} \frac{1}{2} \sum_t e^{2i\pi \frac{t}{2} (\widehat{B \oplus B'})} = s'$

We get

$$\begin{aligned} [s]^{\text{tk}} &= \sum_{\vec{y} \in \{0,1\}^n, \vec{z} \in \{0,1\}^m} (e_0 \downarrow B(\vec{y}))(e_0 \uparrow B'(\vec{z})) \\ &\rightsquigarrow \sum_{\vec{y} \in \{0,1\}^n, \vec{z} \in \{0,1\}^m} \delta_{B(\vec{y}), B'(\vec{z})} \end{aligned}$$

and

$$\begin{aligned} [s']^{\text{tk}} &= \frac{1}{2} \sum_{\vec{y} \in \{0,1\}^n, \vec{z} \in \{0,1\}^m} \sum_t e^{2i\pi \frac{t}{2} (B(\vec{y}) \oplus B'(\vec{z}))} \\ &= \frac{1}{2} \sum_{\vec{y} \in \{0,1\}^n, \vec{z} \in \{0,1\}^m, t \in \{0,1\}} \delta_{B(\vec{y}), B'(\vec{z})} \\ &= \frac{1}{2} \sum_{\vec{y} \in \{0,1\}^n, \vec{z} \in \{0,1\}^m} \delta_{B(\vec{y}), B'(\vec{z})} \sum_{t \in \{0,1\}} 1 \\ &= \sum_{\vec{y} \in \{0,1\}^n, \vec{z} \in \{0,1\}^m} \delta_{B(\vec{y}), B'(\vec{z})} \end{aligned}$$

- Cup (Cap being similar): $s = (e_b \downarrow B) \rightsquigarrow_{\text{sop}} (e_{-b} \uparrow B) = s'$ then we get $[s]^{\text{tk}} = \sum_{\vec{y} \in \{0,1\}^n} (e_b \downarrow B(\vec{y})) \rightsquigarrow \sum_{\vec{y} \in \{0,1\}^n} (e_{-b} \uparrow B(\vec{y})) = [s']^{\text{tk}}$
- $Z_m^n(\alpha)$: $s = (e_k \downarrow B) \rightsquigarrow_{\text{sop}} e^{2i\pi(\frac{\alpha}{2\pi}\widehat{B})} \prod_{j \neq k} (e_j \uparrow B) \prod_j (e'_j \downarrow B) = s'$

We get

$$\begin{aligned} [s]^{\text{tk}} &= \sum_{\vec{y} \in \{0,1\}^n} (e_k \downarrow B(\vec{y})) \\ &\rightsquigarrow \sum_{\vec{y} \in \{0,1\}^n} e^{i\alpha B(\vec{y})} \prod_{j \neq k} (e_j \uparrow B(\vec{y})) \prod_j (e'_j \downarrow B(\vec{y})) \\ &= \sum_{\vec{y} \in \{0,1\}^n} e^{2i\pi(\frac{\alpha}{2\pi}\widehat{B}(\vec{y}))} \prod_{j \neq k} (e_j \uparrow B(\vec{y})) \prod_j (e'_j \downarrow B(\vec{y})) = [s']^{\text{tk}} \end{aligned}$$

- Hadamard: $s = (e_0 \downarrow B) \rightsquigarrow_{\text{sop}} \frac{1}{\sqrt{2}} \sum_z e^{2i\pi(\frac{z}{2}\widehat{B})} (e_1 \downarrow z) = s'$

We have

$$[s]^{\text{tk}} = \sum_{\vec{y} \in \{0,1\}^n} (e_0 \downarrow B(\vec{y}))$$

$$\rightsquigarrow \frac{1}{\sqrt{2}} \sum_{\vec{y} \in \{0,1\}^n} (-1)^{B(\vec{y})} (e_1 \downarrow B(\vec{y})) + (e_1 \downarrow \neg B(\vec{y}))$$

Take $[s']^{\mathbf{tk}}$ and sum z over $\{B(\vec{y}), \neg B(\vec{y})\}$ and remember that $B^2 = B$.

then:

$$\begin{aligned} & \frac{1}{\sqrt{2}} \sum_{\vec{y} \in \{0,1\}^n} e^{2i\pi(\frac{B(\vec{y})}{2} \widehat{B(\vec{y})})} (e_1 \downarrow B(\vec{y})) + e^{2i\pi(\frac{\neg B(\vec{y})}{2} \widehat{B(\vec{y})})} (e_1 \downarrow \neg B(\vec{y})) \\ &= \frac{1}{\sqrt{2}} \sum_{\vec{y} \in \{0,1\}^n} e^{2i\pi(\frac{B(\vec{y})}{2})} (e_1 \downarrow B(\vec{y})) + (e_1 \downarrow \neg B(\vec{y})) \\ &= \frac{1}{\sqrt{2}} \sum_{\vec{y} \in \{0,1\}^n} (-1)^{B(\vec{y})} (e_1 \downarrow B(\vec{y})) + (e_1 \downarrow \neg B(\vec{y})) = [s']^{\mathbf{tk}} \quad \square \end{aligned}$$

We can show a result on the growth size of the token-state as it rewrites, which was the motivation for the use of this formalism.

Proposition 5.6.7. *Let $D \in \mathbf{ZX}$ and $s, s' \in \mathbf{tkS}_{\mathbf{SOP}}(D)$ such that all Boolean polynomials B_j in s are reduced to a single term of degree ≤ 1 , and such that $s \rightsquigarrow_{\mathbf{SOP}} s'$. Then, the size of s' is bounded by: $\mathcal{S}(s') \leq \mathcal{S}(s) + \Delta(D)$ where \mathcal{S} denotes the cumulative number of terms in the phase polynomial and the number of tokens in the token-state, and where $\Delta(D)$ represents the maximum arity of generators in D .*

Proof. Let $D \in \mathbf{ZX}$ and $s \in \mathbf{tkS}_{\mathbf{SOP}}(D)$ such that its $B_j \in \{0, 1, y\}_{y \in V}$ for all j . Note that, at worse, all collisions do not change the size of the term (at best reduce the size). Indeed, we turn two tokens into at most two terms in the phase polynomial, since $\frac{\tilde{z}}{2}(B_{j_1} \oplus B_{j_2}) = \frac{\tilde{z}}{2}(B_{j_1} + B_{j_2} - 2B_{j_1}B_{j_2}) = \frac{\tilde{z}}{2}(B_{j_1} + B_{j_2})$ because we work modulo 1 in the phase polynomial.

Hence, since a rewrite step consists in a diffusion step followed by some collision rule, showing the result only for diffusions is enough.

- Diffusions through Cups and Caps do not change the size.
- A diffusion through H adds a single term in the phase polynomial. However, since H is in the diagram, $\Delta(D) \geq 2$, so the proposition holds.
- A diffusion through a Green-spider with arity δ adds $\delta - 2$ tokens, and a single term in the phase polynomial. However, $\delta \leq \Delta(D)$. □

The requirement on Boolean polynomials may seem overly restrictive. However, it is invariant under rewriting: starting with a token-state in this form ensures polynomial growth.

Polarity can be defined in this setting (and is even more natural, as we do not need to consider each term individually) providing the notions of well-formedness and cycle-balancedness. The main results from Section 5.3 are valid in this setting. We recover strong normalization for well-formed, cycle-balanced token-states (Theorem 5.3.16), Local Confluence (Proposition 5.3.18) and their corollaries, such as uniqueness of normal forms (Corollary 5.3.20).

Non-empty terminal token states can also be interpreted as SOP-morphisms. Suppose an SOP-token state

$$S = s \sum_{\vec{y}'} e^{2i\pi P} \prod_i (b_i \downarrow B_i(\vec{y}')) \prod_i (a_i \uparrow A_i(\vec{y}'))$$

on a diagram D with $\mathcal{I}(D) = [a_i]_{1 \leq i \leq n}$ and $\mathcal{O}(D) = [b_i]_{1 \leq i \leq m}$.

Then $[S]^{\mathbf{SOP}} := s \sum_{\vec{y}} e^{2i\pi P(\vec{y})} |B_0(\vec{y}), \dots\rangle \langle A_0(\vec{y}), \dots|$ is the SOP morphism associated to S . We have the following commutative diagram:

$$\begin{array}{ccc} \mathbf{tkS}_{\mathbf{SOP}} \downarrow & \xrightarrow{[\cdot]^{\mathbf{tk}}} & \mathbf{tkS} \downarrow \\ \downarrow [\cdot]^{\mathbf{SOP}} & & \downarrow [\cdot] \\ \mathbf{SOP} & \xrightarrow{[\cdot]} & \mathbf{Qubit} \end{array}$$

Where $\mathbf{tkS}_{\mathbf{SOP}} \downarrow$ (resp. $\mathbf{tkS} \downarrow$) is the set of non-empty well-formed terminal SOP-token states (resp. token states), and $\mathbf{tkS} \downarrow \xrightarrow{[\cdot]} \mathbf{Qubit}$ is the interpretation obtained from Theorem 5.3.25.

5.6.2. SOP Token Machine for Mixed Processes

We now aim at adapting the SOP token machine to \mathbf{ZX}^{\pm} , the formalism for completely positive maps.

Definition 5.6.8. *Let D be a ZX-diagram. A \mathbf{SOP}^{\pm} -token is a quadruplet $(p, d, B, B') \in \mathcal{E}(D) \times \{\downarrow, \uparrow\} \times \mathbb{F}_2[\vec{y}] \times \mathbb{F}_2[\vec{y}]$ where $\vec{y} := (y_i)_{0 \leq i < n}$ are variables from a set of variables \mathcal{V} . We denote the set of \mathbf{SOP}^{\pm} -tokens on D with variables \vec{y} by $\mathbf{tk}_{\mathbf{SOP}^{\pm}}(D)[\vec{y}]$. Similar to what was done in Definition 5.6.1, a \mathbf{SOP}^{\pm} -token-state is a quadruplet*

$$(s, \vec{y}, P, \{t_j\}_{0 \leq j < p}) \in \mathbb{R} \times \mathcal{V}^n \times \mathbb{R}[\vec{y}] / (1, \{y_i^2 - y_i\}_{0 \leq i < n}) \times \mathbf{tk}_{\mathbf{SOP}^{\pm}}(D)[\vec{y}]^P$$

To better reflect what this quadruplet represents, we usually write it as:

$$s \sum_{\vec{y}} e^{2i\pi P(\vec{y})} (p_0, d_0, B_0(\vec{y}), B'_0(\vec{y})) \dots (p_{m-1}, d_{m-1}, B_{m-1}(\vec{y}), B'_{m-1}(\vec{y}))$$

We denote the set of \mathbf{SOP}^{\pm} -token-states on D by $\mathbf{tkS}_{\mathbf{SOP}^{\pm}}(D)$.

In other words, the difference with the previous machine is that tokens here have an additional Boolean function (e.g. y in $(a \downarrow x, y)$). The rewrite rules can be found in Table 5.7.

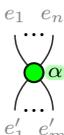
$e_0 \downarrow$	$(e_0 \downarrow B_0, B_1)$	$\rightsquigarrow_c \frac{1}{4} \sum_{z_0, z_1} e^{2i\pi(\frac{z_0}{2} \widehat{B_0 \oplus B_1} + \frac{z_1}{2} \widehat{B'_0 \oplus B'_1})}$	(Collision)
$e_0 \cup e_1$	$(e_b \downarrow B, B')$	$\rightsquigarrow_d (e_{-b} \uparrow B, B')$	(\cup -diffusion)
$e_0 \cap e_1$	$(e_b \uparrow B, B')$	$\rightsquigarrow_d (e_{-b} \downarrow B, B')$	(\cap -diffusion)
	$(e_k \downarrow B_0, B_1)$	$\rightsquigarrow_d e^{2i\pi \frac{\alpha}{2\pi} (\widehat{B_0} - \widehat{B_1})} \prod_{j \neq k} (e_j \uparrow B_0, B_1)$	(α -Diffusion)
	$(e'_k \uparrow B_0, B_1)$	$\rightsquigarrow_d e^{2i\pi \frac{\alpha}{2\pi} (\widehat{B_0} - \widehat{B_1})} \prod_j (e_j \uparrow B_0, B_1)$	
$e_0 \downarrow$	$(e_0 \downarrow B, B')$	$\rightsquigarrow_d \frac{1}{2} \sum_{z, z'} e^{2i\pi(\frac{z}{2} \widehat{B} + \frac{z'}{2} \widehat{B'})} (e_1 \downarrow z, z')$	(\downarrow -Diffusion)
$e_1 \downarrow$	$(e_1 \uparrow B, B')$	$\rightsquigarrow_d \frac{1}{2} \sum_{z, z'} e^{2i\pi(\frac{z}{2} \widehat{B} + \frac{z'}{2} \widehat{B'})} (e_0 \uparrow z, z')$	
$\underline{\underline{e_0}}$	$(e_0 \downarrow B, B')$	$\rightsquigarrow_d \frac{1}{2} \sum_z e^{2i\pi \frac{z}{2} (\widehat{B \oplus B'})}$	(Trace-Out)

Table 5.7.: The rewrite rules for \rightsquigarrow_{\pm} .

It is possible to link this formalism back to the mixed processes-free **SOP**-token-states, using the existing CPM construction for ZX-diagrams. We extend this map by CPM : $\mathbf{tkS}_{\mathbf{SOP}}^{\pm}(D) \rightarrow \mathbf{tkS}_{\mathbf{SOP}}(\text{CPM}(D))$, defined as:

$$s \sum_{\vec{y}} e^{2i\pi P(\vec{y})} \prod_j (p_j, d_j, B_j(\vec{y}), B'_j(\vec{y})) \mapsto s \sum_{\vec{y}} e^{2i\pi P(\vec{y})} \prod_j (p_j, d_j, B_j(\vec{y})) (\overline{p}_j, d_j, B'_j(\vec{y})).$$

As described in Section 2.1, $\text{CPM}(D)$ can be seen as two copies of D where $\underline{\underline{\quad}}$ is replaced by a cup between the two copies. Each token in D corresponds to two tokens in $\text{CPM}(D)$, at the same spot but in the two copies of D . The two Boolean polynomials B and B' represent the Boolean polynomials of the two corresponding tokens.

We can then show that this rewriting system is consistent:

Theorem 5.6.9. *Let D be a \mathbf{ZX}^{\pm} -diagram, and $t_1, t_2 \in \mathbf{tkS}_{\mathbf{SOP}}^{\pm}(D)$. Then whenever $t_1 \rightsquigarrow_{\pm} t_2$ we have $\text{CPM}(t_1) \rightsquigarrow_{\mathbf{SOP}}^{\{1,2\}} \text{CPM}(t_2)$.*

Proof. Similar to proof of Theorem 5.4.2, done by induction on \rightsquigarrow_{\pm}

- Collision: $t_1 = (e_0 \downarrow B_0, B_1)(e_0 \uparrow B'_0, B'_1) \rightsquigarrow_{\pm} \frac{1}{4} \sum_{z_0, z_1} e^{2i\pi(\frac{z_0}{2}\widehat{B_0 \oplus B_1} + \frac{z_1}{2}\widehat{B'_0 \oplus B'_1})} = t_2$

$$\begin{aligned} \text{Then } \text{CPM}(t_1) &= (e_0 \downarrow B_0)(\bar{e}_0 \uparrow B'_0)(e_0 \downarrow B_1)(\bar{e}_0 \uparrow B'_1) \\ &\rightsquigarrow_{\text{sof}}^2 (\frac{1}{2} \sum_{z_0} e^{2i\pi\frac{z_0}{2}(\widehat{B_0 \oplus B'_0})})(\frac{1}{2} \sum_{z_1} e^{2i\pi\frac{z_1}{2}(\widehat{B_1 \oplus B'_1})}) = \text{CPM}(t_2) \end{aligned}$$

- Cup (Cap being similar): $t_1 = (e_b \downarrow B, B') \rightsquigarrow_{\pm} (e_{-b} \uparrow B, B') = t_2$ with $\text{CPM}(t_1) = (e_0 \downarrow B)(\bar{e}_0 \downarrow B') \rightsquigarrow_{\pm}^2 = (e_{-b} \uparrow B)(\bar{e}_{-b} \uparrow B') \text{CPM}(t_2)$

- $Z_m^n(\alpha)$: $t_1 = (e_k \downarrow B_0, B_1) \rightsquigarrow_{\pm} e^{2i\pi\frac{\alpha}{2\pi}(\widehat{B_0} - \widehat{B_1})} \prod_{j \neq k} (e_j \uparrow B_0, B_1) \prod_j (e'_j \downarrow B_0, B_1) = t_2$

$$\begin{aligned} \text{We get } \text{CPM}(t_1) &= (e_k \downarrow B_0)(\bar{e}_k \downarrow B_1) \\ &\rightsquigarrow_{\text{sof}}^2 (e^{2i\pi(\frac{\alpha}{2\pi}\widehat{B_0})} \prod_{j \neq k} (e_j \uparrow B_0) \prod_j (e'_j \downarrow B_0))(e^{2i\pi(\frac{-\alpha}{2\pi}\widehat{B_1})} \prod_{j \neq k} (\bar{e}_j \uparrow B_0) \prod_j (\bar{e}'_j \downarrow B_1)) \\ &= (e^{2i\pi(\frac{\alpha}{2\pi}\widehat{B_0})} \times e^{2i\pi(\frac{-\alpha}{2\pi}\widehat{B_1})})(\prod_{j \neq k} (e_j \uparrow B_0) \prod_j (e'_j \downarrow B_0) \prod_{j \neq k} (\bar{e}_j \uparrow B_0) \prod_j (\bar{e}'_j \downarrow B_1)) = \text{CPM}(t_2) \end{aligned}$$

- Hadamard: $t_1 = (e_0 \downarrow B, B') \rightsquigarrow_{\pm} \frac{1}{2} \sum_{z, z'} e^{2i\pi(\frac{z}{2}\widehat{B} + \frac{z'}{2}\widehat{B'})} (e_1 \downarrow z, z') = t_2$

$$\begin{aligned} \text{with } \text{CPM}(t_1) &= (e_0 \downarrow B)(\bar{e}_0 \downarrow B') \\ &\rightsquigarrow_{\text{sof}}^2 (\frac{1}{\sqrt{2}} \sum_z e^{2i\pi(\frac{z}{2}\widehat{B})} (e_1 \downarrow z)) (\frac{1}{\sqrt{2}} \sum_{z'} e^{2i\pi(\frac{z'}{2}\widehat{B'})} (\bar{e}_1 \downarrow z')) = \text{CPM}(t_2) \end{aligned}$$

- Ground: $t_1 = (e_0 \downarrow B, B') \rightsquigarrow_{\pm} \frac{1}{2} \sum_z e^{2i\pi\frac{z}{2}(\widehat{B \oplus B'})} = t_2$

Do not forget that the ground is translated as a cup, so we end up with two wires e_0 and \bar{e}_0 on both side of the cup, so:

$\text{CPM}(t_1) = (e_0 \downarrow B)(\bar{e}_0 \downarrow B')$ then we just need to pass one of the two token on the other side of the cup and apply a collision:

$$\text{CPM}(t_1) \rightsquigarrow_{\text{sof}} (\bar{e}_0 \uparrow B)(\bar{e}_0 \downarrow B') \rightsquigarrow_{\text{sof}} \frac{1}{2} \sum_z e^{2i\pi\frac{z}{2}(\widehat{B \oplus B'})} = \text{CPM}(t_2). \quad \square$$

In fact, the \rightsquigarrow_{\pm} rewriting rule will only be simulated by 2 rewriting rules ($\rightsquigarrow_{\text{sof}}$), except in the case of the Trace-out where ($\rightsquigarrow_{\text{sof}}$) only needs to apply one rule.

Again, the notions of polarity, well-formedness and cycle-balancedness can be adapted, and again, we get strong (Theorem 5.3.16), confluence (Corollary 5.3.19), and uniqueness of normal forms (Corollary 5.3.20) for well-formed and cycle-balanced token states.

5.7. Conclusion and Future Work

In this chapter presented a novel *particle-style* semantics for ZX-Calculus. Based on a token-machine automaton, it emphasizes the *asynchronicity* and *non-orientation* of the computational content of a ZX-diagram. Compared to existing token-based semantics of quantum computation such as [Dal17], our proposal furthermore supports decentralized tokens where the position of a token can be in superposition.

The Token Machine has been adapted to the case of mixed-processes, and we give another version using the sum-over-paths semantics which allows the number of tokens to stays bounded in size.

As quantum circuits can be mapped to ZX-diagrams, our token machines induce a notion of asynchronicity for quantum circuits. This contrasts with the notion of token machine defined in [Dal17] where some form of synchronicity is enforced: in their works each token represents a qubit and therefore multiple tokens need to be synchronized in order to properly handle entanglement, while our tokens do not represent qubits, but they collect the operation that will be applied to the input qubits.

Our token machine gives us a new way to look at how a ZX-diagram computes with a more local, operational approach, and in fact could be applied to any tensor network.

This work is a first step towards adding more expressive logical and computational constructions in the ZX-Calculus to get closer to the theory of proof nets, such as considering biproducts (that we define in the next chapter) or even recursion (that we leave as future works.)

Chapter 6.

Many Worlds Calculus

Abstract

In this chapter, we explore the interaction between two monoidal structures: a multiplicative one, for the encoding of pairing, and an additive one, for the encoding of choice. We propose a PROP to model computation in this framework, where the choice is parametrized by an algebraic side effect: the model can support regular tests, probabilistic and non-deterministic branching, as well as quantum branching, i.e., superposition.

The graphical language comes equipped with a (i) a token-based semantics (ii) a worlds labelling system, giving us information on how and where some equation can take place and (iii) an equational theory based on the worlds labelling. We also show how a quantum version of the language of isos from Chapter 4 can be encoded into the Many-Worlds.

References: Results of this chapter is a draft under submission, in collaboration with Marc de Visme, Benoît Valiron and Renaud Vilmart.

6.1. Introduction

Two Canonical Monoidal Structures. The basic execution flow of a computation is arguably based on three notions: sequences, tuples and branches. Sequences form the building block of compositionality, tuples are what makes it possible to consider multiple pieces of information together, while branches allow the behaviour to change depending on the inputs or on the state of the system. In a graphical language, described by a PROP $(\mathcal{C}, \top, \boxtimes)$, the monoidal structure formalizes how the bunching of wires behaves. This monoidal structure is very versatile. On one hand, it can be considered in a *multiplicative* way, with $A \boxtimes B$ seen as the pairing of an element of type A and an element of type B . This approach is one followed in the design of MLL proof-nets for instance [Dal17], or in the ZX-Calculus. On the other hand, one can consider the monoidal structure in an *additive* way, with \boxtimes for instance being a co- or a bi-product. Standard examples are the category FinRel of finite sets and relations, forming an additive PROP with \boxtimes being the disjoint union, or the category of finite dimensional vector spaces (or semi-modules) and linear maps, with \boxtimes being the cartesian product.

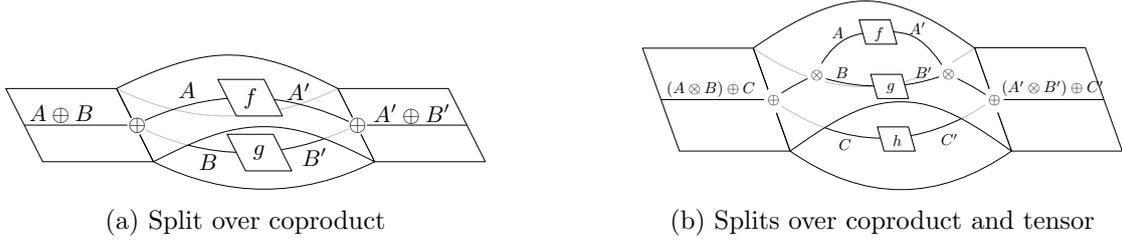


Figure 6.1.: Examples of branchings

Graphical languages based on linear optics such as the PBS-Calculus [CP20] make use of an additive structure. From a computational perspective, an additive monoidal structure can be regarded as the possibility to *choose* a computational path upon the state of the input. Depending on the underlying system, this choice can be regarded as deterministic (if based on Set), non-deterministic (if based on Rel), probabilistic (if based on a suitable semi-module), *etc.*

To be able to handle both pairing and branching in a PROP, we cannot uniquely identify \boxtimes as being multiplicative and additive. We instead need to *extend* the PROP with two additional monoidal structures, one for pairing (\otimes) and one for branching (\oplus).

In this chapter, we focus on a framework where these two monoidal structures are available. Graphical languages for such a setting usually rely on a notion of *sheet*, or *worlds*, to handle general branching [Dun09; Mel14]. Figure 6.1a shows for instance how to represent the construction of the morphism $f \oplus g : A \oplus A' \rightarrow B \oplus B'$ out of $f : A \rightarrow A'$ and $g : B \rightarrow B'$. The symbol “ \oplus ” stands for the “split” of worlds. Such a graphical language therefore comes with two distinct “splits”: one for the monoidal structure—leaving inside one specific world—, and one for the coproduct—splitting worlds—. They can be intertwined, as shown in Figure 6.1b. Another approach followed by [CDH20] externalizes the two products (tensor product and coproduct) into the structure of the diagrams themselves, at the price of a less intuitive tensor product and a form of synchronization constraint.

However, in the state of the art this “splitting-world” understanding has only been carried for deterministic or probabilistic branching [Dal17; Dun09; Sta15]. These existing approaches do not support more exotic branching, such as *quantum superposition*.

Limitation of Current Approaches and Objective of the Chapter. Although there is a finer and finer understanding of superposition of causal orders in the literature, none of the existing PROPs can support both the quantum switch on complex data built from tensors and coproducts. We claim in this chapter that the same intuition underlying probabilistic branching can be followed for quantum (and more general) branching. In the conventional case, $1 \oplus 1$ is a regular boolean: either “left” (standing e.g., for True) *or* “right” (standing e.g., for False). In quantum computation, the sum-type $1 \oplus 1$

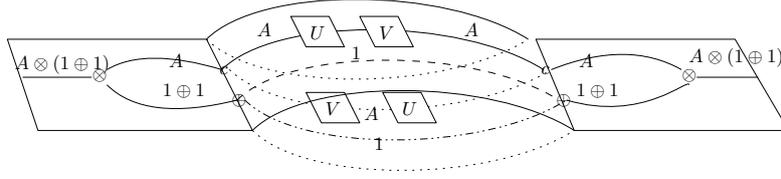


Figure 6.2.: Quantum Switch with Worlds

can however be understood as a *sum of vector spaces*, giving an alternative interpretation to $1 \oplus 1$: it can be regarded as the type of a quantum bit, superposition of True and False. One should note that this appealing standpoint should be taken cautiously: (Pure) quantum information imposes strong constraints on the structure of the data in superposition: orthogonality and unit-norm must be preserved [AG05; SVV18].

The Quantum Switch can then be naturally understood in this framework. Consider for instance Figure 6.2, read from left to right: as input, a pair of an element of type A and a quantum bit. Based on the value of the qubit (True or False), the wire A goes in the upper or the lower sheet, and is fed with U then V or V then U . Then everything is merged back together.

Organization of the chapter In this chapter, we introduce a new graphical language for quantum computation, based on compact category with biproduct [HV19]. This language allows us to express any process with both pairing and a general notion of algebraic branching, encompassing deterministic, non-deterministic, probabilistic and quantum branching. In Section 6.3 we develop first a token-based semantics as in Chapter 5. The development of the token machine is split in two. First, in Section 6.3.2 we introduce a pulse token machine, following the intuition from Section 5.5.1. We show that the token machine is terminating and confluent. Then in Section 6.3.3 we develop the asynchronous token machine as a special case of the pulse one. We show how to simulate a run of the pulse token machine with the asynchronous one and show confluence and termination. From there, we develop the worlds labelling system in Section 6.4 and the equational theory in Section 6.6. We show as an example how to encode some basic quantum primitives into the language and then show how to encode the Quantum Switch into. We then compare our language with other known graphical languages in Section 6.8. We finish this chapter showing how a quantum variant of the language from Chapter 4 can be interpreted as diagrams of the language.

6.2. The Many-Worlds Calculus

Our calculus is parametrized by a commutative semiring $(R, +, 0, \times, 1)$. It can be instantiated by the complex numbers $(\mathbb{C}, +, 0, \times, 1)$ to represent pure quantum computations,

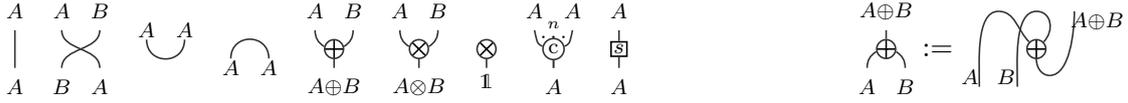
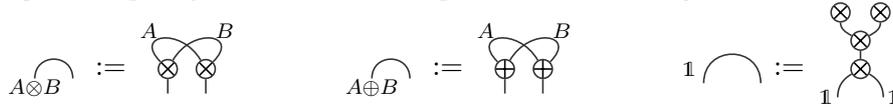


Figure 6.3.: Generators of our First Graphical Language ($n \geq 0, s \in R$)

of data. In order to handle these cases, we will introduce a worlds labelling system in Section 6.4. We write $f : \mathbb{Z}^n(A_i) \rightarrow \mathbb{Z}^m(B_j)$ for a diagram f with n input wires of type A_1, \dots, A_n and m output wires of type B_1, \dots, B_m .

Remark 6.2.1. *Instead of having the Cup and the Cap as generators and defining the mirrored version of each generator through them, one could proceed the other way around by defining the Cap as follows, and the Cup in a mirrored way:*



The Many-Worlds Calculus was developed as an extension of the works from Chapter 5 in which a biproduct was added to the ZX-Calculus. While it wasn't clear how to give a denotational semantics to such a language, the development of the token machine gave us a more intuitive look at how the generators of the languages should behave, and from there the denotational semantics and then the worlds labelling was developed. However, in this thesis we only focus on the token machine, equational theory and encoding of the programming language, as the categorical definitions and denotational semantics along with its property was developed by Marc de Visme and Renaud Vilmart.

6.3. The Token Machine

The Token machine follows the principles of Chapter 5:

- We give a distinct name to each of its wires. By convention, we will use the names e_0, e_1, e_2, \dots for naming the wires of the diagram. We also consider the two sides of a cup or cap as different wires.
- The notion of path, cycle, length of a path are the same.

In Section 6.3.1 we introduce the base formalism of tokens and token states. In Section 6.3.2 we present a first token machine, based on a pulse rewriting strategy, akin to the one discussed in Section 5.5.1. In Section 6.3.3 we introduce the asynchronous token machine as a special case of the pulse one: a rewriting of the asynchronous machine will correspond to a pulse of a generator where a token is entering from one of the input or output wire. Finally, in Section 6.3.4 we discuss how some “incorrect” diagrams are interpreted by the token machine in order to motivate the worlds system introduced in Section 6.4.

6.3.1. Tokens and Token States

Each type can be assigned a set of basis elements, by analogy with vector spaces and with the set of closed values of type A from the language of Chapter 4.

Definition 6.3.1 (Basis Element). *Basis elements are defined inductively as follows:*

- \star is the unique basis element of type $\mathbb{1}$

For t_A and t_B basis elements of types A and B respectively:

- $\langle t_A, t_B \rangle$ is a basis element of type $A \otimes B$
- $\text{inj}_l t_A$ and $\text{inj}_r t_B$ are basis elements of $A \oplus B$

We can define an inner product of basis elements t and t' of a type A simply by

$$\langle t | t' \rangle = \begin{cases} 1 & \text{if } t = t' \\ 0 & \text{if } t \neq t' \end{cases}$$

By analogy with vector spaces, the set of basis elements (or the set of closed value from Chapter 4) of a type A is an orthonormal basis.

In addition to its basis elements, each type is provided with a specific element called annihilator, denoted by \bullet . Intuitively, these states are here to indicate parts of the diagram that are being ignored during evaluation, e.g., one of the two if-then-else branches during the evaluation of a classical program. Although it may be quite mysterious for now, we allow this element to be used in inner products, as follows: $\langle \bullet | \bullet \rangle = 1$ and $\langle \bullet | t \rangle = \langle t | \bullet \rangle = 0$ for any basis element t .

Definition 6.3.2 (Token). *A token on diagram D is a 3-tuple made of:*

- A name of a wire of D , corresponding to the position of the token
- A direction in $\{\uparrow, \downarrow\}$
- A state, being either: \bullet or a basis element of the type of the wire it rests on

Tokens only define part of the evaluation of a diagram. To completely capture it, we need to consider collections of tokens that interact with each other.

Definition 6.3.3 (Token State). *A token state on a diagram D is a \mathbb{C} -valued multivariate polynomial evaluated in tokens on D , e.g., it is a \mathbb{C} -weighted sum of products of tokens.*

6.3.2. Token Machine's Pulse Rewrite Strategy

We start by developing a *Pulse Rewrite Strategy* for the Token Machine, in which we show strong normalization and confluence. Then, following the intuition from Section 5.5.1, we develop an asynchronous token machine for the Many-Worlds and show that both strategies are equivalent and that results from the pulse rewriting strategy can be transposed to the asynchronous one.

The idea is that each generator “pulses” – it generates all necessary token states to fully capture its dynamics – then tokens that end up colliding interact, hence capturing the interaction between neighbouring generators.

Again, the token state's evolution rules are put in two categories. The first is the set of collision rules, which thanks to the above definition of the inner product can be summed up in one line:

$$e_0 \mid :: (e_0 \downarrow x)(e_0 \uparrow y) \rightarrow_c \langle x \mid y \rangle$$

That is: tokens that collide on a wire perform the inner product of their respective states, either reducing to 0, the absorbing element of products of tokens that represents an error that cancel a product from the sum of product of the tokens state, or to 1, the neutral element.

The other set of rules indicates how a generator “pulses”:

$$\begin{array}{c} A \quad B \\ e_0 \quad e_1 \\ \otimes \\ e_2 \\ A \otimes B \end{array} :: -- \rightarrow_p \sum_{\substack{t_A \in \mathbb{B}_A \\ t_B \in \mathbb{B}_B}} (e_0 \uparrow t_A)(e_1 \uparrow t_B)(e_2 \downarrow \langle t_A, t_B \rangle) + (e_0 \uparrow \bullet)(e_1 \uparrow \bullet)(e_2 \downarrow \bullet)$$

$$\begin{array}{c} A \otimes B \\ e_2 \\ \otimes \\ e_0 \quad e_1 \\ A \quad B \end{array} :: -- \rightarrow_p \sum_{\substack{t_A \in \mathbb{B}_A \\ t_B \in \mathbb{B}_B}} (e_0 \downarrow t_A)(e_1 \downarrow t_B)(e_2 \uparrow \langle t_A, t_B \rangle) + (e_0 \uparrow \bullet)(e_1 \downarrow \bullet)(e_2 \uparrow \bullet)$$

$$\begin{array}{c} A \quad B \\ e_0 \quad e_1 \\ \oplus \\ e_2 \\ A \oplus B \end{array} :: -- \rightarrow_p \sum_{t_A \in \mathbb{B}_A} (e_0 \uparrow t_A)(e_1 \uparrow \bullet)(e_2 \downarrow \text{inj}_l t_A) + \sum_{t_B \in \mathbb{B}_B} (e_0 \uparrow \bullet)(e_1 \uparrow t_B)(e_2 \downarrow \text{inj}_r t_B) + (e_0 \uparrow \bullet)(e_1 \uparrow \bullet)(e_2 \downarrow \bullet)$$

$$\begin{array}{c} A \oplus B \\ e_2 \\ \oplus \\ e_0 \quad e_1 \\ A \quad B \end{array} :: -- \rightarrow_p \sum_{t_A \in \mathbb{B}_A} (e_0 \downarrow t_A)(e_1 \downarrow \bullet)(e_2 \uparrow \text{inj}_l t_A) + \sum_{t_B \in \mathbb{B}_B} (e_0 \downarrow \bullet)(e_1 \downarrow t_B)(e_2 \uparrow \text{inj}_r t_B) + (e_0 \downarrow \bullet)(e_1 \downarrow \bullet)(e_2 \uparrow \bullet)$$

$$\begin{array}{c} A \quad A \\ e_1 \quad \dots \quad e_n \\ \textcircled{\bullet} \\ e_0 \\ A \end{array} \quad \text{---} \rightarrow_p \quad \sum_{i=1}^n \sum_{t_A \in \mathbb{B}_A} (e_1 \uparrow \bullet) \dots (e_i \uparrow t_A) \dots (e_n \uparrow \bullet) (e_0 \downarrow t_A) \\ + (e_1 \uparrow \bullet) \dots (e_n \uparrow \bullet) (e_0 \downarrow \bullet)$$

$$\begin{array}{c} A \\ e_0 \\ \textcircled{\bullet} \\ e_1 \quad \dots \quad e_n \\ A \quad A \end{array} \quad \text{---} \rightarrow_p \quad \sum_{i=1}^n \sum_{t_A \in \mathbb{B}_A} (e_1 \downarrow \bullet) \dots (e_i \downarrow t_A) \dots (e_n \downarrow \bullet) (e_0 \uparrow t_A) \\ + (e_1 \downarrow \bullet) \dots (e_n \downarrow \bullet) (e_0 \uparrow \bullet)$$

$$\begin{array}{c} A \\ \boxed{\text{S}} \\ e_0 \\ | \\ e_1 \\ A \end{array} \quad \text{---} \rightarrow_p \quad s \cdot \sum_{t_A \in \mathbb{B}_A} (e_0 \uparrow t_A) (e_1 \downarrow t_A) + (e_0 \uparrow \bullet) (e_1 \downarrow \bullet)$$

$$\begin{array}{c} e_0 \quad \text{---} \quad e_1 \\ A \quad A \end{array} \quad \text{---} \rightarrow_p \quad \sum_{t_A \in \mathbb{B}_A} (e_0 \downarrow t_A) (e_1 \downarrow t_A) + (e_0 \downarrow \bullet) (e_1 \downarrow \bullet)$$

$$\begin{array}{c} A \quad A \\ e_0 \quad \text{---} \quad e_1 \end{array} \quad \text{---} \rightarrow_p \quad \sum_{t_A \in \mathbb{B}_A} (e_0 \uparrow t_A) (e_1 \uparrow t_A) + (e_0 \uparrow \bullet) (e_1 \uparrow \bullet)$$

$$\begin{array}{c} \otimes \\ e_0 \\ | \\ 1 \end{array} \quad \text{---} \rightarrow_p \quad (e_0 \downarrow \star) + (e_0 \downarrow \bullet)$$

$$\begin{array}{c} 1 \\ e_0 \\ | \\ \otimes \end{array} \quad \text{---} \rightarrow_p \quad (e_0 \uparrow \star) + (e_0 \uparrow \bullet)$$

Remark 6.3.4. *The upside-down versions of each of these generators is simply exchanging \uparrow and \downarrow .*

Notice that we denote the collision rules by \rightarrow_c and the pulse rules by \rightarrow_p . We are now ready to define the pulse strategy:

Definition 6.3.5 (Pulse on a generator). *Given a diagram D with a generator g , we write $\rightarrow_{p\{g\}}$ for the pulse rewriting that pulses the generator g .*

Definition 6.3.6 (Restricted Pulse). *Given a set $S = \{g_1, \dots, g_n\}$ of the generators of a diagram D , we define the S -restricted pulse by making all generators of S pulse exactly once: $\rightarrow_{p|S}^{\text{all}} = \rightarrow_{p|\{g_1\}} \cdots \rightarrow_{p|\{g_n\}}$*

Definition 6.3.7 (Maximum collisions). *We denote by $t \rightarrow_c^{\text{max}} s$ the reduction $t \rightarrow_c^* s$ such that s is collision free.*

Definition 6.3.8 (All Pulsing). *Given a diagram D with generators S We define $\rightarrow_p^{\text{all}}$ as $\rightarrow_{p|S}^{\text{all}}$, that is, all generators of the diagram pulse exactly once.*

Definition 6.3.9 (Pulse Strategy). *The pulse strategy is defined as being: $\rightarrow_p^{\text{all}}; \rightarrow_c^{\text{max}}$, that is we make every generator pulse exactly once, then we apply collision rules as long as we can.*

Remark 6.3.10. *Notice that there is an ambiguity here: as the first pulse may create several terms, what we mean is that the next pulse rewrite must be applied on all of them, and so on. Notice also that this amounts to multiplying together the terms obtained from each individual pulse rewrite. We could technically allow more freedom on the order of application of the pulses, but that would defeat the purpose of having a straightforward canonical rewrite strategy, which is the aim here.*

For example, if we have two generators g_1, g_2 and we first pulse the generator g_1 and obtain the token state $t_1 + t_2$, then the generator g_2 pulse for both t_1 and t_2 , obtaining $t'_1 + t'_2$.

Remark 6.3.11. *As each generator is forced to pulse, there is no need for the diagram to be connected. We may however have fringe cases, with connected components in the diagram that have no generators that pulse. To remedy this, via the equational theory defined in Section 6.6 we get that the identity is nothing but $\begin{array}{c} | \\ \square \\ | \end{array} = \square$, and make this last generator pulse in these cases. Notice that it is technically important to break the wire's name in two, so that the tokens obtained from the pulse are not removed by a collision.*

When looking at the rule of the Plus, we can notice that in each summation where a token carrying some value of type A (resp. B) goes to the right (resp. left), an annihilator is sent to the left (resp. right): this represents the fact that, if you imagine a token in the state $\text{inj}_l t$ (resp. $\text{inj}_r t$) entering the node through the bottom, it can only go on the left branch (resp. right branch), and hence the annihilator is here to *disable* the other branch. This is similar for the n -ary contraction, except that an annihilator is sent to *all* the other branches.

Example 6.3.12 (Not Diagram). *Take the type of boolean $\mathbb{B} = \mathbf{1} \oplus \mathbf{1}$, then we compute the pulse semantic of the NOT-diagram:*

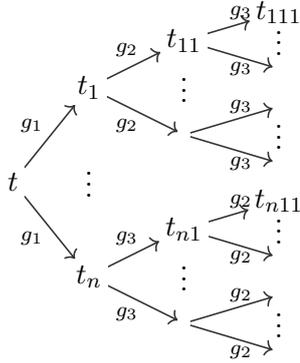
$$\begin{array}{c}
 \mathbf{1} \oplus \mathbf{1} \\
 \oplus \\
 e_0 \\
 \oplus \\
 e_1 \quad e_2 \\
 \oplus \\
 e_3 \\
 \oplus \\
 \mathbf{1} \oplus \mathbf{1}
 \end{array}
 \quad \dashrightarrow_p^2 \quad \left(\begin{array}{l}
 (e_0 \uparrow \mathbf{inj}_l \star)(e_1 \downarrow \star)(e_2 \downarrow \bullet) \\
 + (e_0 \uparrow \mathbf{inj}_r \star)(e_1 \downarrow \bullet)(e_2 \downarrow \star) \\
 + (e_0 \uparrow \bullet)(e_1 \downarrow \bullet)(e_2 \downarrow \bullet)
 \end{array} \right) \cdot$$

$$\begin{array}{l}
 \left(\begin{array}{l}
 (e_2 \uparrow \star)(e_1 \uparrow \bullet)(e_3 \downarrow \mathbf{inj}_l \star) \\
 + (e_2 \uparrow \bullet)(e_1 \uparrow \star)(e_3 \downarrow \mathbf{inj}_r \star) \\
 + (e_2 \uparrow \bullet)(e_1 \uparrow \bullet)(e_3 \downarrow \bullet)
 \end{array} \right) \\
 \rightarrow_c^* (e_0 \uparrow \mathbf{inj}_l \star)(e_3 \downarrow \mathbf{inj}_l \star) \langle \bullet \mid \star \rangle \langle \star \mid \bullet \rangle \\
 + (e_0 \uparrow \mathbf{inj}_l \star)(e_3 \downarrow \mathbf{inj}_r \star) \langle \star \mid \star \rangle \langle \bullet \mid \bullet \rangle \\
 + (e_0 \uparrow \mathbf{inj}_l \star)(e_3 \downarrow \bullet) \langle \bullet \mid \bullet \rangle \langle \bullet \mid \star \rangle \\
 + (e_0 \uparrow \mathbf{inj}_r \star)(e_3 \downarrow \mathbf{inj}_l \star) \langle \bullet \mid \bullet \rangle \langle \star \mid \star \rangle \\
 + (e_0 \uparrow \mathbf{inj}_r \star)(e_3 \downarrow \mathbf{inj}_r \star) \langle \star \mid \bullet \rangle \langle \bullet \mid \star \rangle \\
 + (e_0 \uparrow \mathbf{inj}_r \star)(e_3 \downarrow \bullet) \langle \bullet \mid \star \rangle \langle \bullet \mid \bullet \rangle \\
 + (e_0 \uparrow \bullet)(e_3 \downarrow \mathbf{inj}_l \star) \langle \star \mid \bullet \rangle \langle \bullet \mid \bullet \rangle \\
 + (e_0 \uparrow \bullet)(e_3 \downarrow \mathbf{inj}_r \star) \langle \bullet \mid \star \rangle \langle \bullet \mid \bullet \rangle \\
 + (e_0 \uparrow \bullet)(e_3 \downarrow \bullet) \langle \bullet \mid \bullet \rangle \langle \bullet \mid \bullet \rangle \\
 = (e_0 \uparrow \mathbf{inj}_l \star)(e_3 \downarrow \mathbf{inj}_r \star) \\
 + (e_0 \uparrow \mathbf{inj}_r \star)(e_3 \downarrow \mathbf{inj}_l \star) \\
 + (e_0 \uparrow \bullet)(e_3 \downarrow \bullet)
 \end{array}$$

The three elements of the sum represent the three possible executions: The first element of the sum, $(e_0 \uparrow \mathbf{inj}_l \star)(e_3 \downarrow \mathbf{inj}_r \star)$ represents the fact that given input $\mathbf{inj}_l \star$, the token machine will output the token state $\mathbf{inj}_r \star$, the second element of the sum does the opposite : those two elements represent sending swapping the value of a boolean. The third represents the choice of not executing the diagram at all.

In the Example 6.3.12, we took the liberty to do both pulses at once, and distributing the terms obtained between the two token states obtained. Formally, in a run, there is an order of application of the pulses. More than that: since a first rewrite may yield several terms, we can choose the order of application of the remaining pulses independently for all the terms. To make things clearer, let us denote a rewrite run as a directed graph where vertices represent terms, and edges are labelled by a rewrite from a term to one of the terms it rewrites to (we simply write g to represent $\rightarrow_{p|\{g\}}$). For instance, if a diagram has 3 generators $(g_i, i \in \{1, 2, 3\})$, one possible run of the pulse part of the

rewrite strategy may be:



It is important to underline the fact that a run where all pulses have occurred is one where every path from the root to a leaf goes through each generator exactly once.

The pulse strategy is very well-behaved:

Proposition 6.3.13.

The following properties hold for the pulse strategy:

- The “pulsing” strategy terminates.

When starting with an empty token state:

- In the terminal token state, there is no token on internal wires.
- In every term of the terminal token state, there is exactly one ascending (resp. descending) token per input (resp. output).

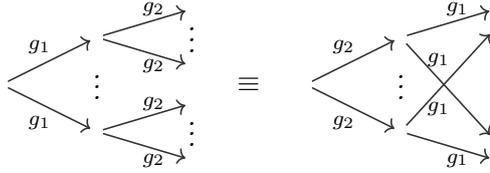
Proof.

- **Termination.** In each branch, every generator pulses exactly once, and there is a finite number of collisions (actually one per wire connecting two generators); and there is a finite number of these branches, as each pulse creates a finite number of terms.
- **Absence of tokens on internal wires.** We have to notice that each generator pulses once in a sum of terms, each of which sports an outgoing token per wire (after the rewrite step). Hence, after the pulse phase, in each term, each internal wire has one descending token from the top generator, and one ascending token from the bottom generator. There will then be a collision per term per internal wire, ridding all internal wires from tokens in the end.
- **Tokens on boundary wires.** The reasoning is the same as previously, except that on an input wire, there is no top generator able to create a descending token; and similarly for output wires. In each term, there will hence be one remaining ascending token per input, and one descending token per output wire. \square

The graph may not necessarily be a tree, as two terms that are colinear αt and βt will be merged into a single vertex $(\alpha + \beta)t$. However, as we will see in the following, this does not happen for the pulse part of the rewrite.

It is easy to check the following lemmas about the pulse part of a rewrite, which both stem from the fact that pulses are totally independent: a token on a given wire and with a given direction can only come from a single generator (the one connected to the wire and which it is pointing away from):

Lemma 6.3.14. *Pulse rewrites (on different generators) commute, i.e.:*



Proof. As we pulse all the generators before applying the collision, the commutation of the order of the pulse is direct. \square

Lemma 6.3.15. *The pulse part (and subparts) of a rewrite run is a directed rooted tree.*

Proof. Direct by definition of the pulse. \square

It is also easy to get the following results on the collision part of the rewrite run:

Lemma 6.3.16. *Collision rewrites commute.*

Proof. As each wire have at most two tokens on it, two different collisions does not interact with each other and hence commute. \square

Lemma 6.3.17. *The collision part of the pulse strategy consists in an in-forest (a forest whose edge points to the roots).*

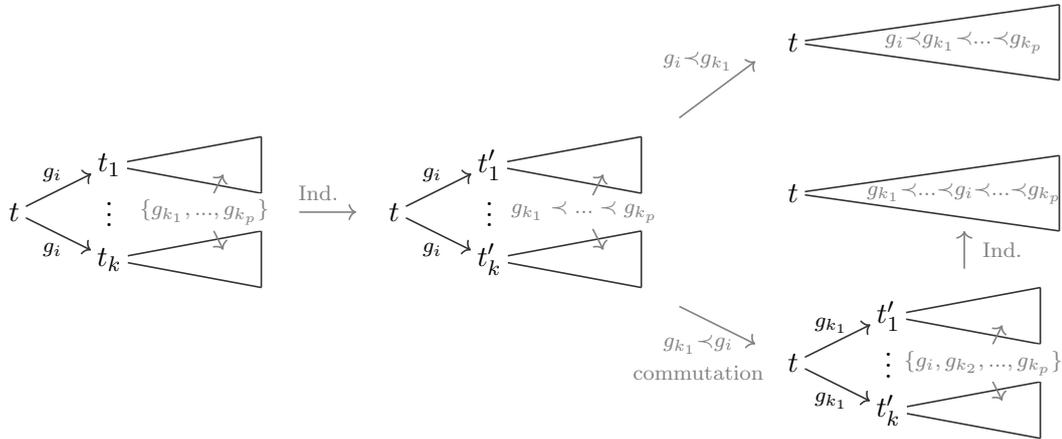
Proof. As some collision may merge, it makes it an in-tree. All terms at the end hence form an in-forest. \square

This can be used to show that all possible orderings of the application of the pulses are equivalent:

Theorem 6.3.18. *The pulse rewrite strategy is confluent.*

Proof. To show confluence, we can show that, for a given diagram, there is a generic run to which every other pulse run is equivalent. Provide an arbitrary (total) ordering of the generators ($g_1 \prec \dots \prec g_n$), and similarly on the wires ($e_1 \prec \dots \prec e_m$). The generic canonical run pulses the generators always in the selected order, and then does the same for the collisions.

Let us first focus on the pulse part of a given run. We can show that using commutation of the pulses, we can get it in the same form as the canonical one: by induction, we can show that every subtree can be put in a form that preserves the ordering. The base case is obvious. For the induction part, consider a vertex t of the tree, and consider all its subtrees t_1, \dots, t_k , each on their own subalphabet of $\{g_1, \dots, g_n\}$. It is easy to see that all the subalphabets are the same, as every path goes through each generator exactly once, and all edges from t to its children go through the same g_i . By induction hypothesis, all the subtrees can be put in a form that preserves the ordering of generators. Let g_{k_1} be the first in this subalphabet, then either $g_i \prec g_{k_1}$, in which case t is already in the right form; or $g_{k_1} \prec g_i$, in which case we can commute g_i and g_{k_1} and use the induction hypothesis again on the subtrees.



Hence, each pulse part of any pulse rewrite yields the same terms. From there, it is easy to see that collisions can be commuted when necessary to get the canonical form, where again the final terms do not change. This finishes the proof of confluence. \square

The pulse token machine will allow us to first define the asynchronous token machine, following the same intuition as the pulse token machine for the ZX-Calculus described in Section 5.5.1.

6.3.3. Token Machine's Asynchronous Rewriting

We now introduce the set of rules for the Asynchronous Token Machine. When a token flows through the graph it will rewrite according to the set of local rules. The rules are

split in two: a single *collision rule* and a set of *diffusion rules*. The collision rule is the same as before: $e_0 \vdash :: (e_0 \downarrow x)(e_0 \uparrow y) \rightarrow_c \langle x \mid y \rangle$.

The diffusion rule tells us what to do when a token enters a node: tokens carrying an annihilator will simply travel through the graph, duplicating itself through the other input / output wire of the node. Otherwise, the token will travel through the node, updating the state it carries and eventually generating new tokens on the other wires of the node.

The diffusion rules are given by:

$$\begin{array}{l}
 \begin{array}{c} A \quad B \\ e_1 \quad e_2 \\ \text{---} \otimes \text{---} \\ A \otimes B \end{array} \quad :: \quad \begin{array}{l} (e_0 \downarrow \bullet) \rightarrow_d (e_1 \uparrow \bullet)(e_2 \downarrow \bullet) \\ (e_1 \downarrow \bullet) \rightarrow_d (e_0 \uparrow \bullet)(e_2 \downarrow \bullet) \\ (e_2 \uparrow \bullet) \rightarrow_d (e_0 \uparrow \bullet)(e_1 \uparrow \bullet) \\ (e_0 \downarrow t_A) \rightarrow_d \sum_{t_B \in \mathbb{B}_B} (e_1 \uparrow t_B)(e_2 \downarrow \langle t_A, t_B \rangle) \\ (e_1 \downarrow t_B) \rightarrow_d \sum_{t_A \in \mathbb{B}_A} (e_0 \uparrow t_A)(e_2 \downarrow \langle t_A, t_B \rangle) \\ (e_2 \uparrow \langle t_A, t_B \rangle) \rightarrow_d (e_0 \uparrow t_A)(e_1 \uparrow t_B) \end{array} \\
 \\
 \begin{array}{c} A \quad B \\ e_1 \quad e_2 \\ \text{---} \oplus \text{---} \\ A \oplus B \end{array} \quad :: \quad \begin{array}{l} (e_0 \downarrow \bullet) \rightarrow_d \sum_{t_B \in \mathbb{B}_B} (e_1 \uparrow t_B)(e_2 \downarrow \text{inj}_r t_B) + (e_1 \uparrow \bullet)(e_2 \downarrow \bullet) \\ (e_1 \downarrow \bullet) \rightarrow_d \sum_{t_A \in \mathbb{B}_A} (e_0 \uparrow t_A)(e_2 \downarrow \text{inj}_l t_A) + (e_0 \uparrow \bullet)(e_2 \downarrow \bullet) \\ (e_2 \uparrow \bullet) \rightarrow_d (e_0 \uparrow \bullet)(e_1 \uparrow \bullet) \\ (e_0 \downarrow t_A) \rightarrow_d (e_1 \uparrow \bullet)(e_2 \downarrow \text{inj}_l t_A) \\ (e_1 \downarrow t_B) \rightarrow_d (e_0 \uparrow \bullet)(e_2 \downarrow \text{inj}_r t_B) \\ (e_2 \uparrow \text{inj}_l t_A) \rightarrow_d (e_0 \uparrow t_A)(e_1 \uparrow \bullet) \\ (e_2 \uparrow \text{inj}_r t_B) \rightarrow_d (e_0 \uparrow \bullet)(e_1 \uparrow t_B) \end{array} \\
 \\
 \begin{array}{c} A \quad A \\ e_1 \quad \dots \quad e_n \\ \text{---} \text{---} \text{---} \text{---} \\ \text{---} \text{---} \text{---} \text{---} \\ e_0 \\ A \end{array} \quad :: \quad \begin{array}{l} (e_i \downarrow t_A) \rightarrow_d (e_0 \downarrow t_A) \prod_{k \in \{1, \dots, n\} \setminus \{i\}} (e_k \uparrow \bullet) \quad \text{for } i \in \{1, \dots, n\} \\ (e_i \downarrow \bullet) \rightarrow_d \sum_{t_A \in \mathbb{B}_A} (e_k \uparrow t_A)(e_0 \downarrow t_A) \prod_{j \in \{1, \dots, n\} \setminus \{i, k\}} (e_j \uparrow \bullet) \\ + (e_0 \downarrow \bullet) \prod_{j \in \{1, \dots, n\} \setminus \{i\}} (e_j \uparrow \bullet) \quad \text{for } i \in \{1, \dots, n\} \\ (e_0 \uparrow \bullet) \rightarrow_d \prod_{j \in \{1, \dots, n\}} (e_j \uparrow \bullet) \\ (e_0 \uparrow t_A) \rightarrow_d \sum_{k \in \{1, \dots, n\}} (e_k \uparrow t_A) \prod_{j \in \{1, \dots, n\} \setminus \{k\}} (e_j \uparrow \bullet) \end{array}
 \end{array}$$

$$\begin{array}{c}
 A \\
 |^{e_0} \\
 \boxplus \\
 |^{e_1} \\
 A
 \end{array}
 \begin{array}{l}
 (e_0 \downarrow \bullet) \rightarrow_d (e_1 \downarrow \bullet) \\
 (e_1 \uparrow \bullet) \rightarrow_d (e_0 \uparrow \bullet) \\
 \vdots \\
 (e_0 \downarrow t_A) \rightarrow_d s(e_1 \downarrow t_A) \\
 (e_1 \uparrow t_A) \rightarrow_d s(e_0 \uparrow t_A)
 \end{array}$$

$$\begin{array}{c}
 e_0 \quad e_1 \\
 \curvearrowright \quad \curvearrowleft \\
 A \quad A
 \end{array}
 \begin{array}{l}
 (e_0 \uparrow \bullet) \rightarrow_d (e_1 \downarrow \bullet) \\
 (e_1 \uparrow \bullet) \rightarrow_d (e_0 \downarrow \bullet) \\
 \vdots \\
 (e_0 \uparrow t_A) \rightarrow_d (e_1 \downarrow t_A) \\
 (e_1 \uparrow t_A) \rightarrow_d (e_0 \downarrow t_A)
 \end{array}$$

$$\begin{array}{c}
 A \quad A \\
 e_0 \quad e_1 \\
 \curvearrowleft \quad \curvearrowright
 \end{array}
 \begin{array}{l}
 (e_0 \downarrow \bullet) \rightarrow_d (e_1 \uparrow \bullet) \\
 (e_1 \downarrow \bullet) \rightarrow_d (e_0 \uparrow \bullet) \\
 \vdots \\
 (e_0 \downarrow t_A) \rightarrow_d (e_1 \uparrow t_A) \\
 (e_1 \downarrow t_A) \rightarrow_d (e_0 \uparrow t_A)
 \end{array}$$

$$\begin{array}{c}
 \otimes \\
 |^{e_0} \\
 \mathbf{1}
 \end{array}
 \begin{array}{l}
 (e_0 \uparrow \bullet) \rightarrow_d 1 \\
 \vdots \\
 (e_0 \uparrow \star) \rightarrow_d 1
 \end{array}$$

The upside-down versions of each of these generators can easily be obtained by simply exchanging \uparrow and \downarrow .

With this token machine, it is important that we give priority to collisions, as to not allow token to cross each other without colliding, just as in Chapter 5. We then take the same definition of collision-free token state (Definition 5.3.6) and of rewriting system (Definition 5.3.7.)

Definition 6.3.19 (Asynchronous Token Machine Rewriting System). *A rewrite step (\rightarrow) is defined as a diffusion rule, followed by all possible collisions involving tokens arising from this rewrite, until none apply, $\rightarrow := \rightarrow_d; \rightarrow_c^{\max}$*

Remark 6.3.20. *Another possible definition of the rewriting system is to use the Pulse Machine. Following intuition given in Section 5.5.1, a diffusion rule from the asynchronous machine is equivalent to a pulse (on the same generator) followed by a collision (on the wire that bore the initial token): $\rightarrow_d = \rightarrow_{p|g}; \rightarrow_c$. We will use this extensively in the proofs.*

Example 6.3.21. *Taking back the NOT diagram from Example 6.3.12, we now use the Asynchronous rewriting machine with initial state $(e_0 \downarrow \text{inj}_l \star)$, underlying the term being rewritten:*

$$\begin{array}{c}
 \mathbf{1} \oplus \mathbf{1} \\
 |^{e_0} \\
 \oplus \\
 e_1 \quad e_2 \\
 \oplus \\
 |^{e_3} \\
 \mathbf{1} \oplus \mathbf{1}
 \end{array}
 \begin{array}{l}
 \vdots \\
 (e_0 \downarrow \text{inj}_l \star) \\
 \rightarrow_d (e_1 \downarrow \star)(e_2 \downarrow \bullet)
 \end{array}$$

$$\begin{aligned} & \rightarrow_d (e_3 \downarrow \text{inj}_r \star)(e_2 \uparrow \bullet)(e_2 \downarrow \bullet) \\ & \rightarrow_c (e_3 \downarrow \text{inj}_r \star) \end{aligned}$$

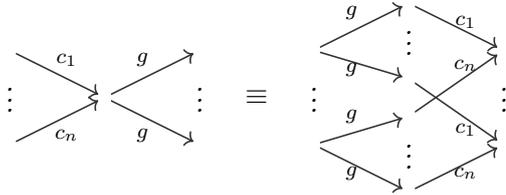
When a token in the state $\text{inj}_l -$ enters a plus node, it will go on the left output, producing an annihilator on the right output as a way of indicating that this branch cannot be taken. Instead, if we had rewritten the annihilator first we would have gotten:

$$\begin{aligned} & (e_1 \downarrow \star)(e_2 \downarrow \bullet) \\ & \rightarrow_d (e_1 \downarrow \star)(e_1 \uparrow \star)(e_3 \downarrow \text{inj}_r \star) + (e_1 \downarrow \star)(e_1 \uparrow \bullet)(e_3 \downarrow \bullet) \\ & \rightarrow_c (e_3 \downarrow \text{inj}_r \star) + (e_1 \downarrow \star)(e_1 \uparrow \bullet)(e_3 \downarrow \bullet) \\ & \rightarrow_c (e_3 \downarrow \text{inj}_r \star) \end{aligned}$$

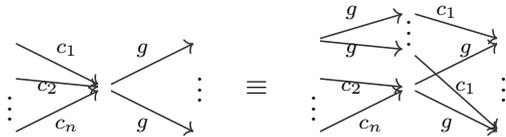
Finally, if we took for initial token state $(e_0 \uparrow \text{inj}_l \star)(e_0 \downarrow \text{inj}_l \star) + (e_0 \uparrow \text{inj}_r \star)(e_0 \downarrow \text{inj}_r \star) + (e_0 \uparrow \bullet)(e_0 \downarrow \bullet)$ we would have gotten the same results as the one from Example 6.3.12. This token state is reminiscent of the initial token state from Theorem 5.3.25.

An important fact is the following:

Lemma 6.3.22. *In the run of a well-formed cycle-balanced token state, collisions and pulses can commute in the following sense:*

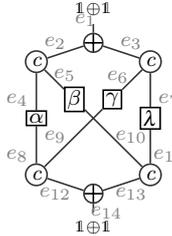


Proof. Let us consider generator g on which the pulse applies, and term t_1 with collision c_1 that happens before the pulse (one of the other collisions may happen on the same wire). The collision necessarily happens between two tokens on the same wire e_1 . Notice that because of well-formedness, e_1 cannot be connected to g . Hence, the collision and the pulse are independent, we have:



Continuing with all collisions gives the desired result. □

Example 6.3.23. We run a simple example of the execution of the Token Machine. Underlined terms are the one being rewritten.



$$\begin{aligned}
 & \text{:: } \underline{(e_1 \downarrow \text{inj}_l \star)} \rightarrow_p (e_2 \downarrow \star)(e_3 \downarrow \bullet) \\
 & \rightarrow_p \underline{(e_4 \downarrow \star)}(e_3 \downarrow \bullet)(e_5 \downarrow \bullet) + \underline{(e_5 \downarrow \star)}(e_3 \downarrow \bullet)(e_4 \downarrow \bullet) \\
 & \rightarrow_p^2 \alpha \underline{(e_8 \downarrow \star)}(e_3 \downarrow \bullet)(e_5 \downarrow \bullet) + \beta \underline{(e_{10} \downarrow \star)}(e_3 \downarrow \bullet)(e_4 \downarrow \bullet) \\
 & \rightarrow_p^2 \alpha \underline{(e_{12} \downarrow \star)}(e_3 \downarrow \bullet)(e_5 \downarrow \bullet)(e_9 \uparrow \bullet) + s \\
 & \rightarrow_p^2 \alpha \underline{(e_{12} \downarrow \star)}(e_6 \downarrow \bullet)(e_7 \downarrow \bullet)(e_{10} \downarrow \bullet)(e_9 \uparrow \bullet) + s \\
 & \rightarrow_p \alpha \underline{(e_{12} \downarrow \star)}(e_9 \downarrow \bullet)(e_7 \downarrow \bullet)(e_{10} \downarrow \bullet)(e_9 \uparrow \bullet) + s \\
 & \rightarrow_c \alpha \underline{(e_{12} \downarrow \star)}(e_7 \downarrow \bullet)(e_{10} \downarrow \bullet) + s \\
 & \rightarrow_p \alpha \underline{(e_{12} \downarrow \star)}(e_{11} \downarrow \bullet)(e_{10} \downarrow \bullet) + s \\
 & \rightarrow_p \alpha \underline{(e_{14} \downarrow \text{inj}_l \star)}(e_{13} \uparrow \bullet)(e_{11} \downarrow \bullet)(e_{10} \downarrow \bullet) + s \\
 & \rightarrow_p \alpha \underline{(e_{14} \downarrow \text{inj}_l \star)}(e_{10} \uparrow \bullet)(e_{11} \uparrow \bullet)(e_{10} \downarrow \bullet)(e_{11} \downarrow \bullet) + s \\
 & \rightarrow_c^2 \alpha \underline{(e_{14} \downarrow \text{inj}_l \star)} + \underline{s} \\
 & \rightarrow^* \alpha \underline{(e_{14} \downarrow \text{inj}_l \star)} + \beta \underline{(e_{14} \downarrow \text{inj}_r \star)}
 \end{aligned}$$

Where $s = \beta(e_{13} \downarrow \star)(e_3 \downarrow \bullet)(e_4 \downarrow \bullet)(e_1 \uparrow \bullet)$ and rewrites similarly as the other operand of the +.

This diagram describes a linear operator from the spaces of input token of type $\mathbb{1} \oplus \mathbb{1}$ to the spaces of output token of type $\mathbb{1} \oplus \mathbb{1}$, that sends $(e_0 \text{inj}_l \star)$ to $\alpha(e_{14} \downarrow \text{inj}_l \star) + \beta(e_{14} \downarrow \text{inj}_r \star)$ and $(e_1 \text{inj}_r \star)$ to $\gamma(e_{14} \downarrow \text{inj}_l \star) + \lambda(e_{14} \downarrow \text{inj}_r \star)$. This map can therefore be seen as the $\begin{pmatrix} \alpha & \gamma \\ \beta & \lambda \end{pmatrix}$ matrix.

This is very reminiscent of the rewrite strategy from Chapter 5, and indeed several results can be transported to our framework. Not all initial configurations make sense, or even terminate in the token machine's semantics, so we start by narrowing the set of allowed states, such as having two tokens on the same output wire without colliding.

Definition 6.3.24 (Polarity of a Term in a Path). Let D be a diagram, and $p \in \text{Paths}(D)$ be a path in D . Let $t = (e, d, x)$ be a token on D . Then:

$$P(p, t) = \begin{cases} 1 & \text{if } e \in p \text{ and } e \text{ is } d\text{-oriented} \\ -1 & \text{if } e \in p \text{ and } e \text{ is } \neg d\text{-oriented} \\ 0 & \text{if } e \notin p \end{cases}$$

We extend the definition to any term $\alpha t_1 \dots t_m$ of a token-state s :

$$P(p, 0) = P(p, 1) = 0, \quad P(p, \alpha t_1 \dots t_m) = P(p, t_1) + \dots + P(p, t_m).$$

Definition 6.3.25 (Well-formedness). *Let D be a diagram, and s a token state on D . We say that s is well-formed if for every term t in s and every path $p \in \text{Paths}(D)$ we have $P(p, t) \in \{-1, 0, 1\}$.*

As before, well-formedness is invariant under any of the token machine rewriting rule defined above. It prevents “bad”, unwanted configurations, such as for instance the possibility for two tokens in the same term to be on the same wire, facing the same direction, but with different states. It does not, however, prevent infinite runs. We can do so by requiring the cycle-balancedness:

Definition 6.3.26 (Cycle-Balanced Token State). *Let D be a diagram, and t a term in a token state on D . We say that t is cycle-balanced if for all cycles $c \in \text{Cycles}(D)$ we have $P(c, t) = 0$. We say that a token state is cycle-balanced if all its terms are cycle-balanced.*

We recover Proposition 5.3.11, Proposition 5.3.12, Proposition 5.3.13, Lemma 5.3.15 and Theorem 5.3.16 in this new setting, the proofs are the same:

Proposition 6.3.27 (Invariance of Well-Formedness). *Well-formedness is preserved by (\rightsquigarrow) : if $s \rightsquigarrow^* s'$ and s is well-formed, then s' is well-formed. \square*

Proposition 6.3.28 (Full Characterization of Well-Formed Terms). *Let D be a Many-Worlds diagram, and $s \in \mathbf{tkS}(D)$ be ill-formed, i.e. there exists a term t in s , and $p \in \text{Paths}(D)$ such that $|P(p, t)| \geq 2$. Then we can rewrite $s \rightsquigarrow s'$ such that a term in s' has a product of at least two tokens of the form $(e_0, d, -)$. \square*

Proposition 6.3.29 (Invariant on Cycles). *Let D be a Many-Worlds diagram, and $c \in \text{Cycles}(D)$ a cycle. Let t_1, \dots, t_n be tokens, and s be a token state such that $t_1 \dots t_n \rightsquigarrow^* s$. Then for every non-null term t in s we have $P(c, t_1 \dots t_n) = P(c, t)$. \square*

Lemma 6.3.30 (Rewinding). *Let D be a Many-Worlds diagram, and t be a term in a well-formed token state on D , and such that $t \rightsquigarrow^* \sum_i \lambda_i t_i$, with $(e_n, d, x) \in t_1$. If t is cycle-balanced, then there exists a path $p = (e_0, \dots, e_n) \in \text{Paths}(D)$ such that e_n is d -oriented in p , and $P(p, t) = 1$. \square*

Theorem 6.3.31 (Termination of well-formed, cycle-balanced token state). *Let D be a Many-Worlds diagram, and $s \in \mathbf{tkS}(D)$ be well-formed. The token state s is strongly normalizing if and only if it is cycle-balanced. \square*

Proposition 6.3.32 (Local Confluence). *Let D be a Many-Worlds diagram, and $s \in \mathbf{tkS}(D)$ be well-formed and collision-free. Then, for all $s_1, s_2 \in \mathbf{tkS}(D)$ such that $s_1 \leftarrow s \rightsquigarrow s_2$, there exists $s' \in \mathbf{tkS}(D)$ such that $s_1 \rightsquigarrow^* s' \leftarrow^* s_2$.*

Proof. We can directly adapt the proof from Proposition 5.3.18. When two rewrites are applied to tokens at position e and e' , we once again reason on the distance between the two tokens. All the initial cases are the same, so we only need to look at when $d(e, e') = 1$. We recall that the two tokens have to point to the same generator (otherwise they would not respect well-formedness). We then show the property for all generators and by case analysis on the state carried by the tokens:

- $e_0 \overset{A}{\cup} e_1$, for any state t, t' we have:

$$\begin{array}{ccc} (e_0 \downarrow t)(e_1 \downarrow t') & \rightarrow_d & (e_1 \uparrow t)(e_1 \downarrow t') \\ & \downarrow d & \downarrow c \\ (e_0 \downarrow t)(e_0 \uparrow t') & \rightarrow_c & \langle t \mid t' \rangle \end{array}$$

- Cap is similar

- $e_1 \overset{A}{\oplus} e_2$: We reason by case hypothesis on the state of the tokens.

$$\begin{array}{ccc} (e_0 \downarrow t)(e_2 \uparrow \text{inj}_l t') & \rightarrow_d & (e_1 \uparrow \bullet)(e_2 \downarrow \text{inj}_l t)(e_2 \uparrow \text{inj}_l t') \\ - & \downarrow d & \downarrow c \\ (e_0 \downarrow t)(e_0 \uparrow t')(e_1 \uparrow \bullet) & \rightarrow_c & (e_1 \uparrow \bullet) \langle t \mid t' \rangle \end{array}$$

$$\begin{array}{ccc} (e_0 \downarrow t)(e_2 \uparrow \text{inj}_r t') & \rightarrow_d & (e_1 \uparrow \bullet)(e_2 \downarrow \text{inj}_l t)(e_2 \uparrow \text{inj}_r t') \\ - & \downarrow d & \downarrow c \\ (e_0 \downarrow t)(e_1 \uparrow \bullet)(e_2 \uparrow t') & \rightarrow_c & 0 \end{array}$$

$$\begin{array}{ccc} (e_0 \downarrow \bullet)(e_2 \uparrow \bullet) & \rightarrow_d & \left(\sum_{t_B \in \mathbb{B}_B} (e_1 \uparrow t_B)(e_2 \downarrow \text{inj}_r t_B) + (e_1 \uparrow \bullet)(e_2 \downarrow \bullet) \right) (e_2 \uparrow \bullet) \\ - & \downarrow d & \downarrow c \\ (e_0 \downarrow \bullet)(e_0 \uparrow \bullet)(e_1 \uparrow \bullet) & \rightarrow_c & (e_1 \uparrow \bullet) \end{array}$$

$$\begin{array}{ccc} (e_0 \downarrow t)(e_2 \uparrow \bullet) & \rightarrow_d & (e_1 \uparrow \bullet)(e_2 \downarrow \text{inj}_l t)(e_2 \uparrow \bullet) \\ - & \downarrow d & \downarrow c \\ (e_0 \downarrow t)(e_0 \uparrow \bullet)(e_1 \uparrow \bullet) & \rightarrow_c & (e_1 \uparrow \bullet) \end{array}$$

$$\begin{array}{ccc} (e_0 \downarrow t_1)(e_1 \downarrow t_2) & \rightarrow_d & (e_1 \uparrow \bullet)(e_2 \downarrow \text{inj}_l t_1)(e_1 \downarrow t_2) \\ - & \downarrow d & \downarrow c \\ (e_0 \downarrow t_1)(e_2 \downarrow \text{inj}_r t_2)(e_0 \uparrow \bullet) & \rightarrow_c & 0 \end{array}$$

- All the other cases being similar.

- The cases  and  are akin to the case of the .

- 

$$\begin{array}{l}
 (e_0 \downarrow \bullet)(e_1 \uparrow t) \rightarrow_d (e_1 \downarrow \bullet)(e_1 \uparrow t) \\
 - \quad \quad \quad \downarrow_d \quad \quad \quad \downarrow_c \\
 s(e_0 \downarrow \bullet)(e_0 \uparrow t) \rightarrow_c \quad \quad \quad 0
 \end{array}$$

$$\begin{array}{l}
 (e_0 \downarrow t_1)(e_1 \uparrow t_2) \rightarrow_d s(e_1 \downarrow t_1)(e_1 \uparrow t_2) \\
 - \quad \quad \quad \downarrow_d \quad \quad \quad \downarrow_c \\
 s(e_0 \downarrow t_1)(e_0 \uparrow t_2) \rightarrow_c \quad \quad \quad s \langle t_1 \mid t_2 \rangle
 \end{array}$$

- All the other cases being similar.

The upside-down version of each generator is similar. □

Corollary 6.3.33 (Confluence). *Let s be a well-formed, cycle-balanced token state on a diagram D . Then (\rightarrow) terminates on s , and it is confluent.*

Finally, we can now relate the pulse token machine to the asynchronous one:

Theorem 6.3.34 (Relation to the Pulse Strategy). *Let t be a well-formed cycle-balanced term on a connected diagram D . A generator will be visited by tokens during the rewrite if there exists a path p from a token of t leading to the generator such that $P(p, t) = 1$. Call S the set of generators that will be visited.*

Then, if $t \rightarrow^{\max} \downarrow s$ then $t \rightarrow_{p|S}^{\text{all}}; \rightarrow_c^{\max} \downarrow s$.

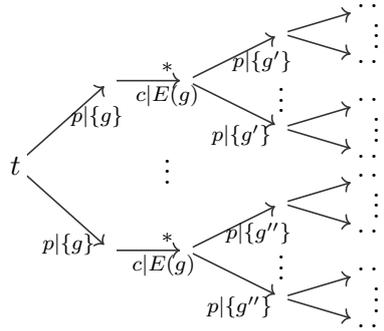
In other words, the result of the rewrite strategy on the term t is the same as the pulse strategy on t restricted to S .

Proof. First, following Remark 6.3.20 we remind that a diffusion rule is equal to a pulse followed by collision.

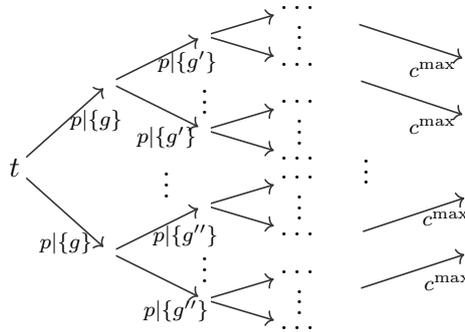
Hence $\rightarrow = \rightarrow_d; \rightarrow_c^{\max} = \rightarrow_{p|g}; \rightarrow_c; \rightarrow_c^{\max}$

We write $c|E(g)$ for the application of the collisions on the edges of a given generator g .

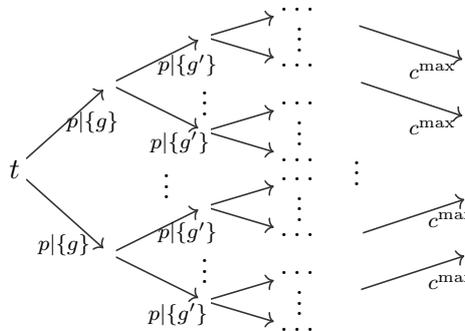
First, consider the following run of the asynchronous token machine:



Then, following Lemma 6.3.22 we can push the collisions rules at the end of the rewriting sequence:



This form a run of the pulse token machine, which by the confluence of the pulse token machine (Theorem 6.3.18) and the fact pulse commutes (Lemma 6.3.14), we can make sure that on each step, all terms pulse on the same generator as in:



□

6.3.4. Discussions

As discussed in Section 6.2.1, some diagrams does not make sense. We will consider two diagrams, both of input type $A \oplus B$, for some A, B and a run of each diagram, using

the asynchronous token machine. For both diagrams, the initial token state consist of a single token entering through the sole input of type $A \oplus B$ in the state $\text{inj}_l t$ for some t an element of the basis of type A .

$$\begin{array}{l}
 \begin{array}{c} e_0 \\ | \\ \oplus \\ | \\ e_1 \quad e_2 \end{array} \quad :: (e_0 \downarrow \text{inj}_l t) \\
 \rightarrow_d (e_1 \downarrow t)(e_2 \downarrow \bullet) \\
 \rightarrow_d (e_2 \uparrow t)(e_2 \downarrow \bullet) \\
 \rightarrow_c 0
 \end{array}
 \qquad
 \begin{array}{c} e_0 \\ | \\ \oplus \\ | \\ e_1 \quad e_2 \\ | \\ \otimes \\ | \\ e_3 \end{array} \quad :: (e_0 \downarrow \text{inj}_l t) \\
 \rightarrow_d (e_1 \downarrow t)(e_2 \downarrow \bullet) \\
 \rightarrow_d (e_1 \downarrow t)(e_1 \uparrow \bullet)(e_3 \downarrow \bullet) \\
 \rightarrow_c 0
 \end{array}$$

In both case, the end result is 0: this can be explained by the fact that in both case, we try to make interact two wires that are in \oplus with one another: on the first diagram through a cup, and the second diagram through a tensor. Once two wires have been split by a \oplus , they cannot interact freely with one another. This requires us to be careful when we will consider the equational theory. In particular, this motivates the introduction of the *worlds labellings* where each wire will be affected a set of worlds, making sure that we can distinguish between diagrams that have a proper semantics and whose semantics will just be 0.

6.4. Worlds labelling

We now want to define the worlds labelling and equational theory of the language. We take inspiration from the Token Machine in order to define both. For instance, take the diagram that splits and merges wires through tensors:



If a token carries the state $\langle t_1, t_2 \rangle$ of type $A \otimes B$, it will be split into the two branches before being remerged again, so it corresponds to just doing nothing, therefore we should have the following equation:

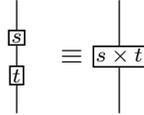
$$\begin{array}{c} | \\ \otimes \\ | \\ \otimes \\ | \end{array} \equiv |$$

and similarly for the plus. Now imagine two scalar one after another on a wire:

Chapter 6. Many Worlds Calculus



then, a token entering this diagram will collect both scalar in, creating their product, therefore we should get the following equations:



Now, consider the diagram of a plus where one of the input wire is bent backward by a cap and put in parallel of the output wire of the plus, together with another wire:



This diagrams cannot be represented in sheet diagrams as the two parallel wire at the bottom left lives in “disjoint worlds” and hence cannot interact, therefore those two wire are in a sort of superposition. It is also possible to consider that the third wire is in all the worlds and hence in superposition with nobody. In sheet diagrams, one cannot have next to each other, two sheets in superposition and one not in superposition with any of those.

One thing that is not inherently clear is how to represent the fact that a part of a diagram can be disabled by an annihilator. This will be done through the *worlds labelling*. We call worlds labelling the attribution of *multiple* worlds to a single wire. The worlds labelling should be thought of as a kind of validity criterion, similarly to what is done in Proof Nets. As mentioned, a wire can be *deactivated*, this was represented in the token machine with the use of the annihilator. When a token enters a co-contraction-node, we take the sum of all the possible outputs where the token can be, in product with annihilators on all the other branches, similarly, the worlds labelling of the output of the co-contraction-node will have to be all disjoint, while its input will be the disjoint union of all the possible worlds labelling of the output. We then define the equational theory on our diagram.

In order to label wires of our diagram with sets of worlds $w \subseteq W$ from a given world set W . For each world $a \in W$, wires labelled by a set containing a are said to be “enabled in a ”, and the others are said “disabled in a ”. This allows us to correlate the enabling of wires. Before making this formal, we illustrate this notion through the following example:

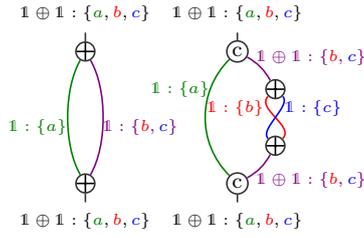
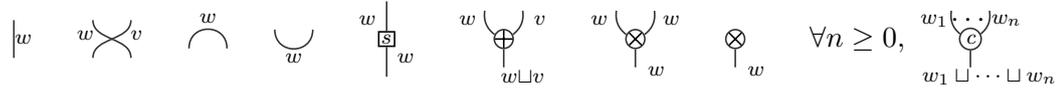


Figure 6.4.: Controlled Not with world set $\{a, b, c, \star\}$

Example 6.4.1. The "controlled not" on quantum bits can be represented by the Figure 6.4. The figure is split into two parts: the control part on the left-hand-side, and the computational part on the right-hand-side. The idea is that the control part, that uses \oplus , will behave as an if-then-else and will bind the world a to the case where the control quantum bit is $|0\rangle$, and the worlds b and c to the case where the control quantum bit is $|1\rangle$. Lastly, the world \star appears nowhere in the labels, and corresponds to "we do not evaluate this circuit at all"¹. On the computational part, we apply the identity within the world a , we negate $|0\rangle$ into $|1\rangle$ within b , and we negate $|1\rangle$ into $|0\rangle$ within c . The "controlled not" above can then be seen as a diagram $\mathcal{D}_{\text{CNOT}}$ of $\mathfrak{A} \rightarrow \mathfrak{A}$ together with a labelling function $\ell_{\mathfrak{A}}$ which labels every wire with a set of worlds.

We now give a formal definition of the concept of worlds:

Definition 6.4.2 (Worlds Labelling). Given a diagram D and a world set W , the worlds labelling consist of a labelling function ℓ_D from the wires of D to the subsets of W , satisfying the following constraints:



where \sqcup denotes disjoint set-theoretic union. The constraints for the mirrored versions are similar. The sequential composition \circ and the parallel composition \square preserve the labels.

We write $f : \mathbb{Z}^n(A_i, w_i) \rightarrow \mathbb{Z}^m(B_j, v_j)$ for a diagram f with n input wires of type A_1, \dots, A_n with worlds labelling w_1, \dots, w_n and m output wires of type B_1, \dots, B_m with worlds labelling v_1, \dots, v_m .

6.5. The Denotational Semantics

We give a few intuitions and how the denotational semantics of the Many-Worlds Calculus works without going into the details and precise definition, as it was mainly developed

¹While not strictly necessary, it is often practical to have a world absent from every wire.

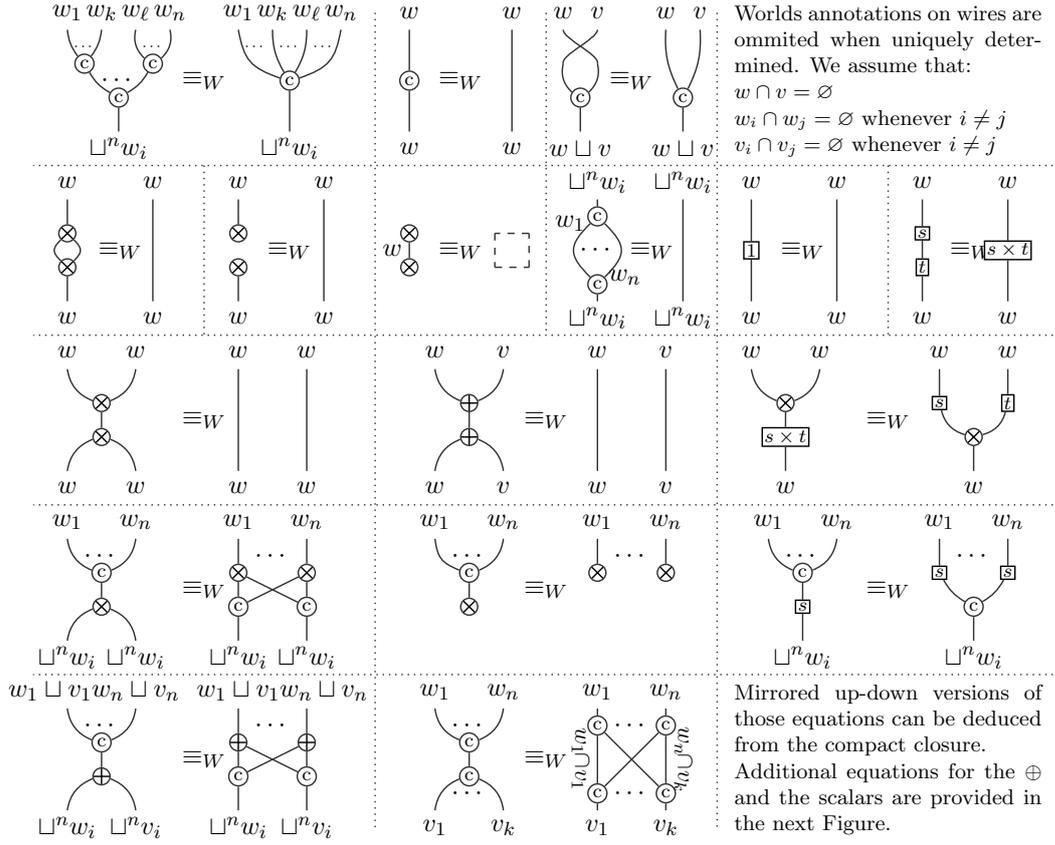


Figure 6.5.: Equations with a Fixed World Set W

The same intuition can be given for the plus: in the case $A \square B$ we get 0, as both data cannot be available, and then we let the token flow either in A or in B in the case $A \square \bullet$ and $\bullet \square B$.

The denotation semantics have a more rigorous definition in [Cha+22] and comes with an universality theorem (i.e. it can represent any matrices) and completeness theorem with regards to the equational theory.

6.6. The Equational Theory

The equational theory is defined in two steps:

1. A set of equations for a fixed set of worlds W . We write \equiv_W for the induced congruence, in other words the smallest equivalence relation satisfying those equations and such that $f \equiv_W f' \implies \forall g, h, l, k. g \circ (f \square h) \circ k \equiv_W g \circ (f' \square h) \circ k$. We list those equations in Figure 6.5.

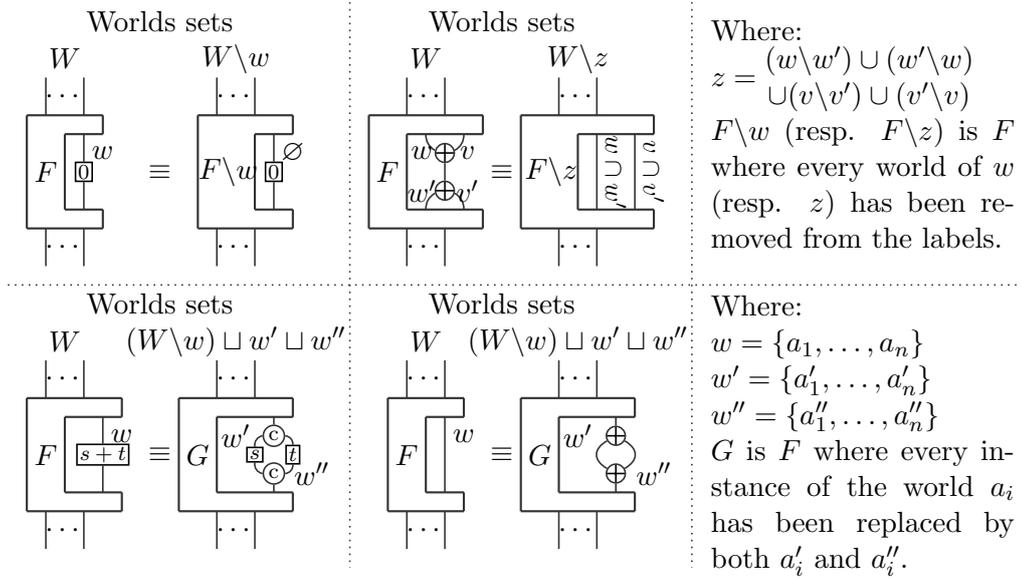


Figure 6.6.: Equations with Side-Effects on World Sets

- Five additional equations with side effects on the set of worlds. We write \equiv for the induced equivalence relation. We have: two equations allowing the annihilation (or creation, when looking at them from right to left) of worlds due to coproducts or scalars (first row of Figure 6.6); Two equations allowing the splitting (or merging, when looking at them from right to left) of worlds due to coproducts or scalars (second row of Figure 6.6).

Example 6.6.1. We consider $R = \mathbb{C}$. We illustrate how our language can encode some basic quantum primitive in it and show how they operate. In Figure 6.7 we show the encoding of a quantum bit $\alpha|0\rangle + \beta|1\rangle$ and the Hadamard unitary. In particular, the Plus allows to "build" a new quantum bit from two scalars in parallel or to "open" a quantum bit to recover its corresponding scalars, the left branch corresponding to $|0\rangle$ and the right branch to $|1\rangle$. The meaning of the Contraction is better seen when applying Hadamard to a quantum bit as we show in Figure 6.8: it allows us to duplicate and sum scalars. The rewriting sequence of Figure 6.8 is made using the equational theory defined in Section 6.6.

6.7. Induced Equations

In Figure 6.5, we presented a set of equations reasonably small, by having equations parametrized by the arity of the contraction, and by omitting a lot of useful equations

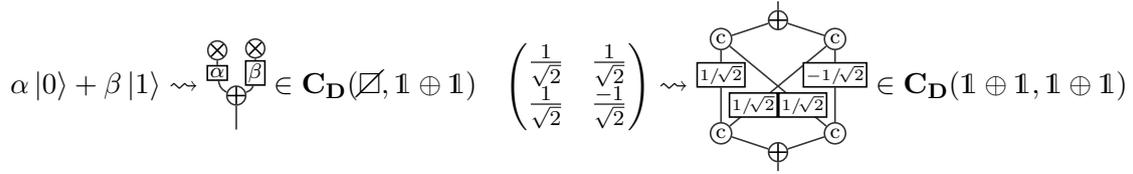


Figure 6.7.: A Quantum Bit and the Hadamard Unitary

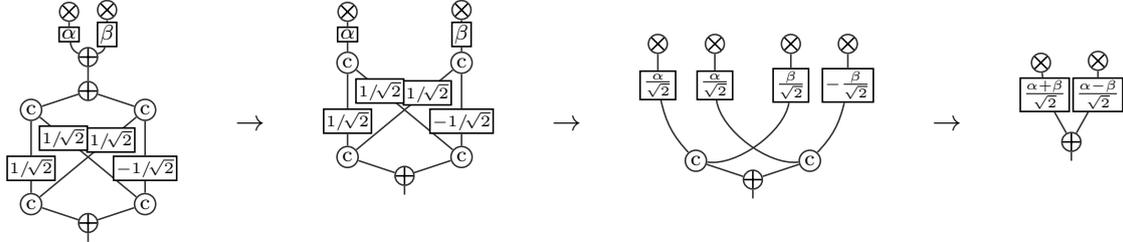
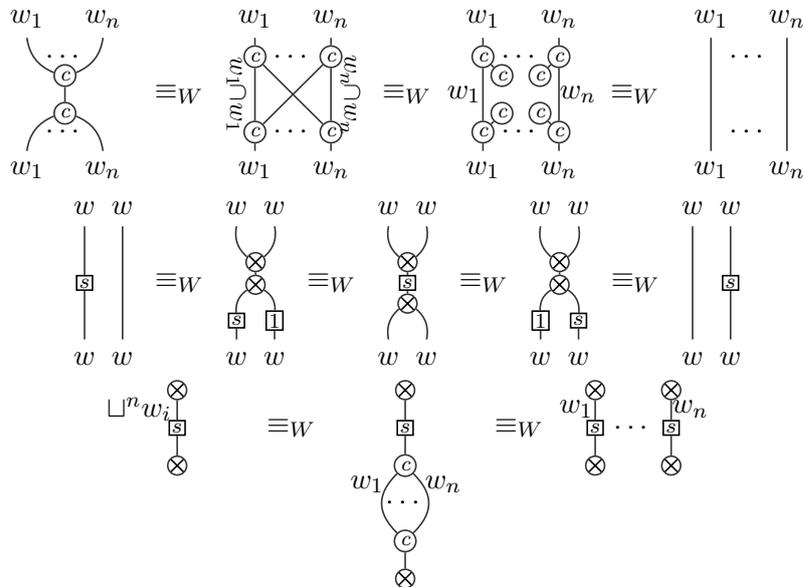


Figure 6.8.: Applying the Hadamard unitary to a quantum bit

that can be deduced from them. In Figure 6.9, we take the opposite approach: we give equations using the contractions of arity zero and two (which are sufficient to generate all the other contractions) and we provide additional axioms that follow from Lemma 6.7.1 and Lemma 6.7.2.

Note that this is only an alternative presentation to the equational theory for \equiv_W , the axioms of Figure 6.6 are still required for \equiv . We can also note the two representations are equivalent by taking the contraction of arity n as a syntactic sugar for multiple composition of the binary contraction.

Lemma 6.7.1. *Whenever w_i are disjoint sets of worlds, we have the following:*



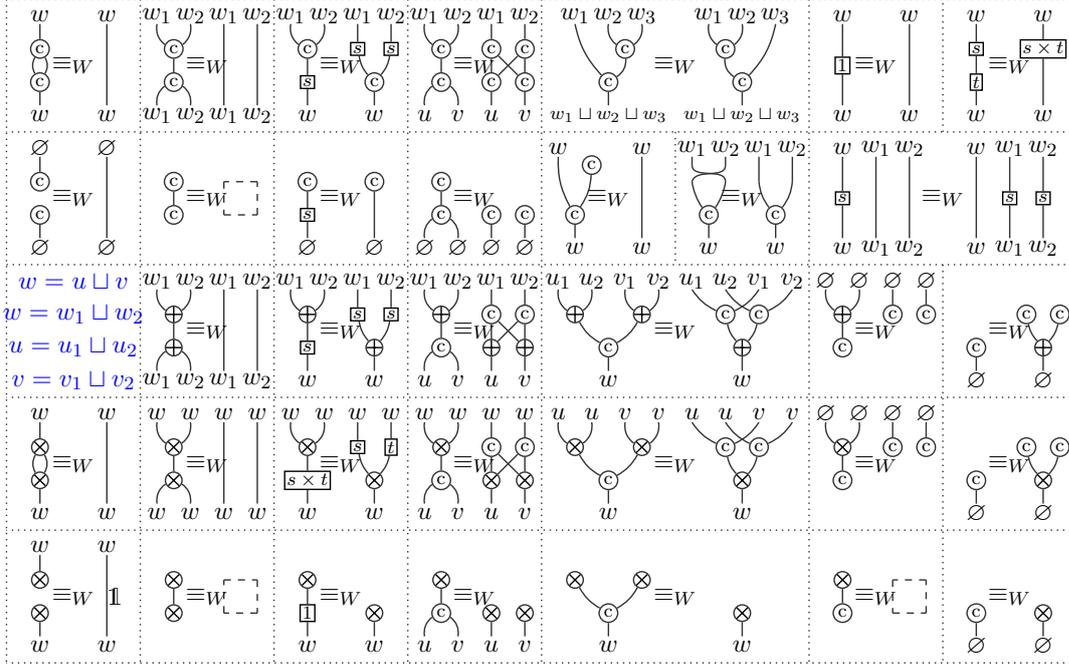
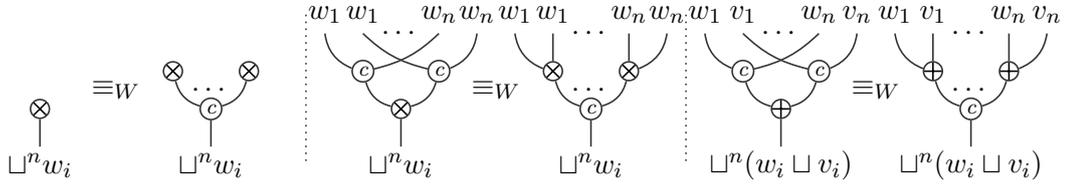


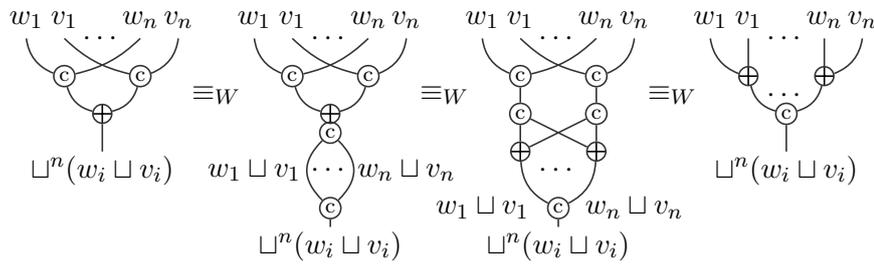
Figure 6.9.: Alternative Presentation of the Equational Theory for a Fixed World Set W

□

Lemma 6.7.2. *Whenever w_i are disjoint sets of worlds, and that the v_i s are disjoint sets of worlds too, we have the following:*

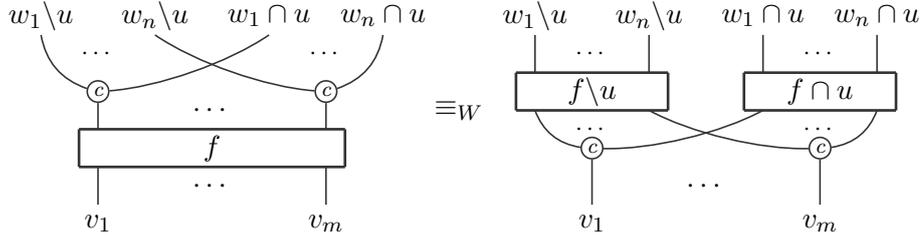


Proof. We provide a proof for the third equation, the first two are proven similarly.



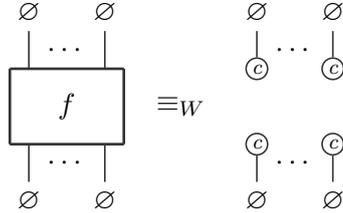
□

Lemma 6.7.3 (Naturality of the Binary Contraction). *For every $f : \square^n(A_i : w_i) \rightarrow \square^m(B_j : v_j)$ with world set W and every $u \subseteq W$, we have*



where $f \setminus u : \square^n(A_i : w_i \setminus u) \rightarrow \square^m(B_j : v_j \setminus u)$ is equal to f where every worlds label w has been replaced by $w \setminus u$, and similarly for $f \cap u$. \square

Lemma 6.7.4 (Empty World). *For every $f : \square^n(A_i : \emptyset) \rightarrow \square^m(B_j : \emptyset)$ with world set W but such that every worlds label of f is \emptyset , we have*



Proof. This is simply proven by replacing every wire by two contractions of arity zero (sixth axiom of Figure 6.5 with $n = 0$), and then using the naturality of the contraction of arity zero (last two lines of Figure 6.5 with $n = 0$) to consume every generator. \square

Example 6.7.5 (The Quantum Switch). *Similarly to the "controlled not" in Example 6.4.1 where a quantum bit controls whether the identity or a negation is applied to some data, one can consider the case where a quantum bit control whether $U \circ V$ or $V \circ U$ is applied to some data (for U and V two quantum operators on the same type A).*

On the rightmost part of Figure 6.10, one can see the most direct implementation of this "quantum switch", but this implementation uses two copies of U and V . On the leftmost part of the figure, we show another implementation which only relies on one copy of each, and both can be rewritten into another using the equational theory.

In those diagrams, the world set is $W = w \sqcup v$ and we rely on violet, blue and red wires to indicate respectively worlds labels $w \sqcup v$, w and v . Each figure has a control side which operates on a quantum bit (type $\mathbb{1} \oplus \mathbb{1}$) and binds the world w to $|0\rangle$ and the world v to $|1\rangle$; and a computational side which operates on some data of an arbitrary type A , on which could be applied U and/or V .

The first rewriting step relies on the two lemmas below, both of which being deducible from the equational theory (see Section 6.7). The second rewriting step is simply using the properties of a compact closed category.

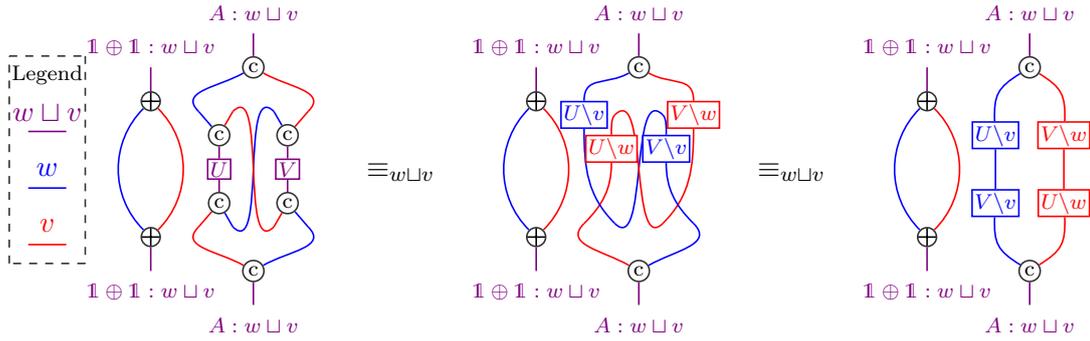
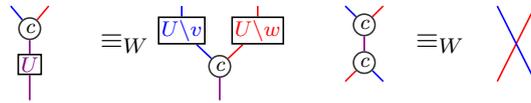


Figure 6.10.: Rewriting the Quantum Switch



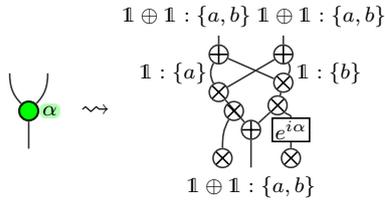
Remark 6.7.6. With Marc de Visme and Renaud Vilmart, a categorical definition of the graphical language has been defined in the setting of compact closed and auto-dual coloured PROP. With it comes a denotational semantics that is sound and complete with regard to the equational theory. Also, a normal form of the diagrams have been developed, which allowed to show the completeness. As most of this work was developed by Marc de Visme and Renaud Vilmart, and was not necessary for this thesis, we did not put it here. The interested reader can refer to [Cha+22].

6.8. Comparison with Other Graphical Languages

The distinctive feature of the Many-Worlds Calculus is that it graphically puts the tensor and the biproduct on an equal footing. By comparison, other graphical language for quantum computing are inherently centred around either one of them. The ZX-calculus [CD11] and cousin languages ZW- and ZH-Calculi [BK19; Had15], as well as Duncan’s Tensor-Sum Logic [Dun09], use the tensor product as the default monoid, while more recent language – particularly for linear optics [CP20; FC22; Clé+22b] – use the biproduct. We have a closer look at each of them in the following and show how – at least part of – each language can be encoded naturally in the Many-Worlds Calculus. Most of them comes equipped with an equational theory. By completeness of our language [Cha+22], all the equations expressible in the fragments we consider can be derived in our framework.

6.8.1. ZX-Calculus

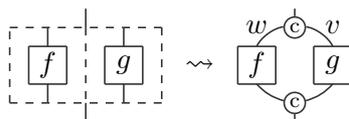
The first difference is the restrictions of the ZX-calculus to computations between qubits, in other words linear map from $\mathbb{C}^{2^n} \mapsto \mathbb{C}^{2^m}$, while our language can encode any linear map from $\mathbb{C}^n \mapsto \mathbb{C}^m$. The Tensor generator allowing the decomposition of \mathbb{C}^{2^n} into instances of \mathbb{C}^2 was already present in the *scalable* extension of the ZX-calculus [CHP19], but the main difference comes from the Plus (and the Contraction).



Additionally, every ZX-diagram can be encoded in our graphical language. The identity, swap, cup and cap of the ZX calculus are encoded by the similar generators over the type $\mathbb{1} \oplus \mathbb{1}$, the Hadamard gate is encoded as in Figure 6.7, and the green spider is encoded as shown above. An encoding for the red spider can then be deduced from those.

6.8.2. Tensor-Sum Logic

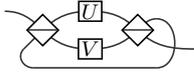
The core difference between the Tensor-Sum Logic [Dun09] and ours is the presence of the contraction in our graphical language. They instead rely on an enrichment of their category by a sum, which they represent graphically with boxes. We show on below how the morphism $f + g$ would be encoded in both their and our language. More generally, their boxes correspond to uses of our contraction generator in a "well-bracketed" way. Another point of difference is their approach to quantum computation, as we do not assign the same semantics to those superpositions of morphisms. In their approach, the superposition is a classical construction and corresponds to the measurement and the classical control flow, while in our approach the superposition is a quantum construction and corresponds to the quantum control.



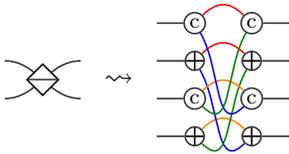
6.8.3. PBS-Calculus

The PBS Calculus [CP20] allow one to represent coherent quantum control by the use of *polarizing beam splitters* (pbs): whenever a qubit enter a pbs node, depending on the polarity of the qubit it will either go through or be reflected. By making implicit the

target system, controlled by the optical system represented by the diagram, the PBS-Calculus allows one to encode the Quantum Switch (depicted below). The pbs generator is related to the \oplus of the Many-Worlds.

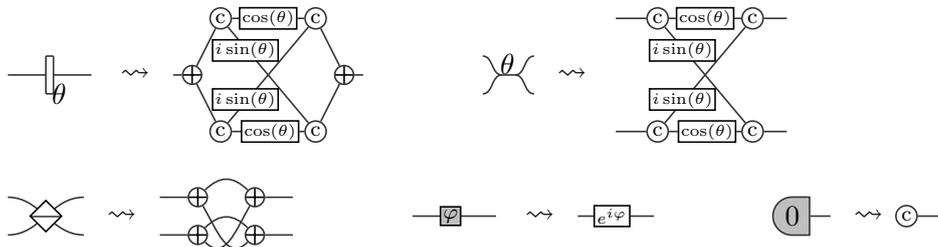


The first main difference with our language is that, since the generators of the PBS-Calculus represent physical components, any PBS-diagram is by construction physical, while our approach is more atomic and decomposes physical components into abstract smaller ones. The second main difference lie in the trace: while they can allow a particle to pass through a wire at most twice, in our system, each wire can be used at most once: more formally, their trace is based on the coproduct while ours is on the tensor product. If we are assured that each wire can only be used once during the computation, any PBS-diagram can be translated to the Many-Worlds calculus directly, with the transformation below, where we distinguish the control system (the part of the diagram connected to \oplus s) from the target system (connected to \odot s) which is implicit in the PBS-Calculus.



6.8.4. LOv-Calculus

In the PBS-Calculus, the qubit in the control system (the one explicitly represented) cannot be put in arbitrary superpositions of $|0\rangle$ and $|1\rangle$ *during* the computation. To allow this feature, we may add some linear optical components to the language's generators and end up with the LOv-Calculus [Cl6+22b]. In this language, there is no trace and there is a unique photon travelling the circuit, which relieves us of the previous constraint. There is also no need for an implicit target system anymore. All wires at the interface between the generators are of type $\mathbb{1} \oplus \mathbb{1}$, and parallel wires have disjoint sets of worlds. Each generator can then be interpreted as follows:



6.8.5. Path-Calculus

The Path-Calculus is another recent graphical language for linear optical circuits [FC22]. Its generators correspond directly to a subset of the Many-Worlds' with $\circ\text{---} \rightsquigarrow \textcircled{\circ}\text{---}$, $\text{---}\triangleright \rightsquigarrow \text{---}\textcircled{\circ}$ and $\text{---}^r\text{---} \rightsquigarrow \text{---}\square\text{---}$; where each wire has type $\mathbb{1}$ and where parallel wires are on disjoint sets of worlds. This language is then used as the core for a more expressive language called QPath, which this time cannot be directly encoded in our language, except when restricting the set of generators (specifically to $n = 1$), in which case $\overset{|1\rangle}{\bullet}\text{---} \rightsquigarrow \textcircled{\otimes}\text{---}$.

6.8.6. Linear Logic's Proof Nets

In a similar spirit to what is done with linear logic proof nets, the Many-Worlds Calculus feature a validity criterion in the form of the world labelling. While in Linear Logic, a non-valid proof structure has no computational content, in our case non-valid diagrams (i.e. diagrams whose only worlds label is \emptyset) can be reduced under the equational theory to a normal form and is captured by the semantics. As in the Many-Worlds Calculus, formulas are self-dual, the connection with MALL validity criterion as in [HVG03] fall short.

6.9. Representing Computation

As a motivational example, we may see how this Many-Worlds diagrams can be used to represent computations expressed in the language of Chapter 4 that explicitly uses the two compositions \otimes (through pairs), and \oplus (through pattern-matching).

6.9.1. Syntax of the Language

The language we present here is adapted from Chapter 4, with the addition of linear combination of terms, as in [SVV18], but without abstraction nor recursion. The syntax that is used in the language is given as follows, with scalars α ranging over the commutative semiring R :

$$\begin{array}{ll}
 \text{(Base types)} & A, B ::= \mathbb{1} \mid A \oplus B \mid A \otimes B \\
 \text{(Isos, first-order)} & \alpha ::= A \leftrightarrow B \\
 \\
 \text{(Values)} & v ::= () \mid x \mid \text{inj}_l v \mid \text{inj}_r v \mid \langle v_1, v_2 \rangle \\
 \text{(Patterns)} & p ::= x \mid \langle p_1, p_2 \rangle \\
 \text{(Expressions)} & e ::= v \mid \text{let } p_1 = \omega p_2 \text{ in } e \mid e + e \mid \alpha e \\
 \text{(Isos)} & \omega ::= \{v_1 \leftrightarrow e_1 \mid \dots \mid v_n \leftrightarrow e_n\}
 \end{array}$$

$$\begin{aligned} \text{(Terms)} \quad t ::= & () \mid x \mid \text{inj}_l t \mid \text{inj}_r t \mid \langle t_1, t_2 \rangle \mid \\ & \alpha t \mid t + t \mid \omega t \mid \text{let } p_1 = t \text{ in } t \end{aligned}$$

As before, the language features branching through the inj_l and inj_r constructors, linear combinations of terms and expressions, and as before, the *isos* comes with the OD_A predicates, used in the typing rule of isos, to ensure exhaustivity and the non-overlapping character of the left-hand and right expressions of the clauses, allowing in particular to define unitaries (in the complex setting). Constraints on the linear combinations may also be used to enforce probabilistic constraints (i.e., that states are normalized in the quantum setting).

Terms and values are considered modulo associativity and commutativity of the addition, and modulo the equational theory of modules.

$$\begin{aligned} \alpha \cdot (e_1 + e_2) &= \alpha \cdot e_1 + \alpha \cdot e_2 & 1 \cdot e &= e \\ \alpha \cdot e + \beta \cdot e &= (\alpha + \beta) \cdot e & \alpha \cdot (\beta \cdot e) &= (\alpha\beta) \cdot e \\ 0 \cdot e_1 + e_2 &= e_2 \end{aligned}$$

We furthermore consider the value and term constructs $\langle -, - \rangle$, $\text{let } - = - \text{ in } -$, $\text{inj}_l(-)$, $\text{inj}_r(-)$ distributive over sum and scalar multiplication.

The typing judgements are the same as in Table 4.2 with the additional rules from [SVV18]:

$$\frac{\Psi; \Delta \vdash_e t_1 : A \quad \Psi; \Delta \vdash_e t_2 : A}{\Psi; \Delta \vdash_e t_1 + t_2 : A} \quad \frac{\Psi; \Delta \vdash_e t : A}{\Psi; \Delta \vdash_e \alpha t : A}$$

For a more complete description of the language, we refer the reader to [SVV18]. In the following, we will use the shorthands $\mathbf{ff} := \text{inj}_l()$ and $\mathbf{tt} := \text{inj}_r()$.

Example 6.9.1. *In the case where $R = \mathbb{C}$, one can encode the Hadamard gate by:*

$$\left\{ \begin{array}{l} \mathbf{ff} \leftrightarrow \frac{1}{\sqrt{2}}(\mathbf{ff} + \mathbf{tt}) \\ \mathbf{tt} \leftrightarrow \frac{1}{\sqrt{2}}(\mathbf{ff} - \mathbf{tt}) \end{array} \right\} : \mathbb{1} \oplus \mathbb{1} \leftrightarrow \mathbb{1} \oplus \mathbb{1}$$

In particular, when $R = \mathbb{C}$, the isos are necessary unitaries [SVV18] and when restricting the type system to only the type of booleans $\mathbb{1} \oplus \mathbb{1}$ it is possible to encode any quantum circuits inside the language.

For the encoding, we will take again the explicit substitution alternative rewriting system from Chapter 4.

We recall Convention 4.4.21:

Convention 6.9.2. *Given a substitution $\sigma = \{x_1 \mapsto v_1, \dots, x_n \mapsto v_n\}$ we will use the shorthand $\text{let } \sigma \text{ in } t$ for $\text{let } x_1 = v_1 \text{ in } \dots \text{let } x_n = v_n \text{ in } t$.*

The results can be adapted in a straightforward way to this new language:

Property 6.9.3. *Given a substitution $\sigma = \{x_i \mapsto v_i\}$ that closes a term t , then $\text{let } \sigma \text{ in } t \rightarrow_{e\beta}^* \sigma(t)$. \square*

Property 6.9.4. *Given two well-typed terms t, t' such that $t \rightarrow t'$ then $t \rightarrow_{e\beta}^* t'$. \square*

6.9.2. Encoding into the Many-Worlds

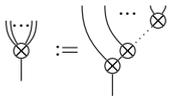
One can encode any term of the language into a Many-Worlds diagram. Given some typing derivation ξ of a term $x_1 : A_1, \dots, x_n : A_n \vdash_e t : B$ we write (ξ, W) for the function that maps ξ to a diagram in the Many-Worlds Calculus with n input wires of type A_1, \dots, A_n and one output wire of type B and where W is the set of worlds needed to encode ξ . For the typing derivation ξ of an iso $\vdash_\omega \omega : A \leftrightarrow B$, (ξ, W) gives a diagram with one input wire of type A and one output wire of type B .

In order to encode a term into the Many-Worlds it is important to know how many worlds are needed. For that, we define $\mathcal{W}(t)$ as the set of worlds needed for t .

Definition 6.9.5. *Let W be an infinite set of worlds, we define $\mathcal{W}(t)$, the set of worlds needed for t by induction over t :*

- $\mathcal{W}(\mathbb{1}) = a \in W$ such that a is fresh.
- $\mathcal{W}(\text{inj}_r t) = \mathcal{W}(\text{inj}_l t) = \mathcal{W}(t)$
- $\mathcal{W}(\langle t_1, t_2 \rangle) = \mathcal{W}(t_1) \uplus \mathcal{W}(t_2)$
- $\mathcal{W}(\alpha t) = \mathcal{W}(t)$
- $\mathcal{W}(t_1 + t_2) = \mathcal{W}(t_1) \uplus \mathcal{W}(t_2)$
- $\mathcal{W}(\omega t) = \mathcal{W}(\omega) \times \mathcal{W}(t)$
- $\mathcal{W}(\{v_1 \leftrightarrow e_1 \mid \dots \mid v_n \leftrightarrow e_n\}) = \biguplus_i \mathcal{W}(e_i)$
- $\mathcal{W}(\text{let } p = t \text{ in } t') = \mathcal{W}(t) \times \mathcal{W}(t')$

$(\xi, \mathcal{W}(t))$ is defined inductively over \vdash_e and \vdash_ω as shown in Figure 6.11. For the encoding,

we use the following syntactic sugar: . In the remaining we will simply write (t) for the encoding of the well-typed term t .

One can show that the input worlds and output worlds of an expression are the same:

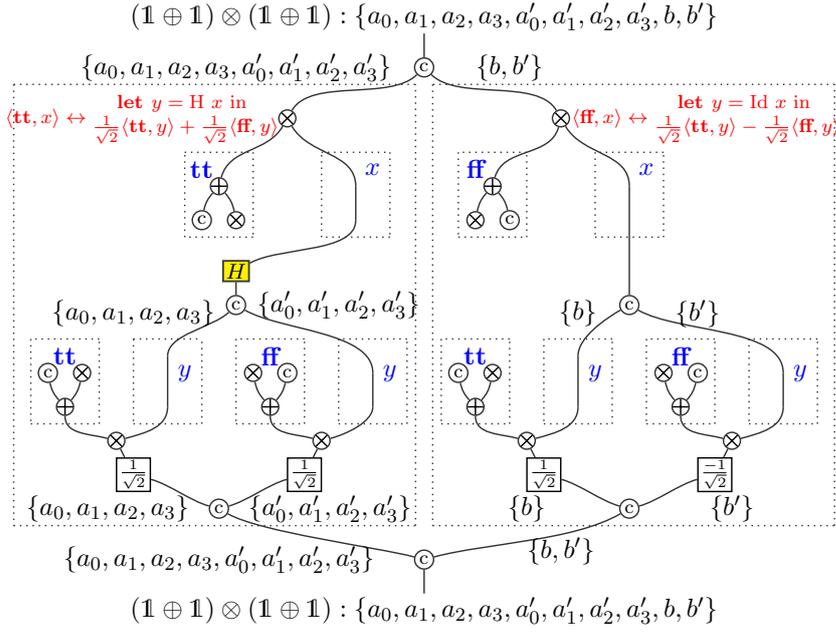
$$\begin{aligned}
 \left(\frac{}{x : A \vdash_e x : A} \right) &= |^A & \left(\frac{}{\vdash_e () : \mathbb{1}} \right) &= \otimes \mathbb{1} & \left(\frac{\xi}{\frac{\Delta \vdash_e t : A}{\Delta \vdash_e \text{ot} : A}} \right) &= \frac{\Delta}{\langle \xi \rangle} \\
 \left(\frac{\xi}{\frac{\Delta \vdash_e t : A}{\Delta \vdash_e \text{inj}_r t : A \oplus B}} \right) &= \frac{\Delta}{\langle \xi \rangle} \oplus & \left(\frac{\xi}{\frac{\Delta \vdash_e t : B}{\Delta \vdash_e \text{inj}_l t : A \oplus B}} \right) &= \langle \xi \rangle \oplus \Delta & \\
 \left(\frac{\frac{\xi_1}{\Delta_1 \vdash_e t_1 : A} \quad \frac{\xi_2}{\Delta_2 \vdash_e t_2 : B}}{\Delta_1, \Delta_2 \vdash_e \langle t_1, t_2 \rangle : A \otimes B} \right) &= \frac{\Delta_1}{\langle \xi_1 \rangle} \otimes \frac{\Delta_2}{\langle \xi_2 \rangle} & \\
 \left(\frac{\frac{\xi_1}{\Delta \vdash_e t_1 : A} \quad \frac{\xi_2}{\Delta \vdash_e t_2 : A}}{\Delta \vdash_e t_1 + t_2 : A} \right) &= \frac{\Delta}{\langle \xi_1 \rangle \oplus \langle \xi_2 \rangle} & \\
 \left(\frac{\frac{\xi_1}{\Delta_1 \vdash_e t_1 : A_1 \otimes \dots \otimes A_n} \quad \frac{\xi_2}{x_1 : A_1, \dots, x_n : A_n, \Delta_2 \vdash_e t_2 : B}}{\Delta_1, \Delta_2 \vdash_e \text{let } \langle x_1, \dots, x_n \rangle = t_1 \text{ in } t_2 : B} \right) &= \frac{\Delta_1}{\langle \xi_1 \rangle} \otimes \frac{\Delta_2}{\langle \xi_2 \rangle} & \\
 \left(\frac{\frac{\xi_i}{\Delta_i \vdash_e v_i : A} \quad \frac{\xi'_i}{\Delta_i \vdash_e e_i : B}}{\vdash_e \{v_1 \leftrightarrow e_1 \mid \dots \mid v_n \leftrightarrow e_n\} : A \leftrightarrow B} \right) &= \frac{\Delta}{\langle \xi_1 \rangle^\dagger \dots \langle \xi_n \rangle^\dagger} & \\
 \left(\frac{\frac{\xi_1}{\vdash_\omega \omega : A \leftrightarrow B} \quad \frac{\xi_2}{\Delta \vdash_e t : A}}{\Delta \vdash_e \omega t : B} \right) &= \frac{\Delta}{\langle \xi_2 \rangle} &
 \end{aligned}$$

Figure 6.11.: Inductive translation of terms from Section 6.9 into Many-Worlds diagrams.

Lemma 6.9.6. *Given a well-typed term $\xi : \Delta \vdash_e t : B$, then $\langle \xi, \mathcal{W}(e) \rangle$ have the same worlds on all of its inputs and all outputs.*

Proof. By a straightforward induction on t :

- Case x : as it is just the identity wire the worlds cannot change.
- Case $()$: direct as there is no input wire.
- All the other cases by direct application of the induction hypothesis on the sub-terms. \square


 Figure 6.12.: Representation of the term t from Example 6.9.7.

Example 6.9.7. We can represent the term

$$t := \left\{ \begin{array}{l} \langle \mathbf{tt}, x \rangle \leftrightarrow \text{let } y = \mathbf{H} x \text{ in } \frac{1}{\sqrt{2}} \langle \mathbf{tt}, y \rangle + \frac{1}{\sqrt{2}} \langle \mathbf{ff}, y \rangle \\ \langle \mathbf{ff}, x \rangle \leftrightarrow \text{let } y = \text{Id } x \text{ in } \frac{1}{\sqrt{2}} \langle \mathbf{tt}, y \rangle - \frac{1}{\sqrt{2}} \langle \mathbf{ff}, y \rangle \end{array} \right\} : (\mathbf{1} \oplus \mathbf{1})^{\otimes 2} \leftrightarrow (\mathbf{1} \oplus \mathbf{1})^{\otimes 2}$$

(already given in [SVV18]) as shown in Figure 6.12. The yellow box stands for the Hadamard gate (which one can build following Example 6.9.1). Each line of this isomorphism corresponds to a column of the figure. Each column start by matching the input as $\langle \mathbf{tt}, x \rangle$ or $\langle \mathbf{ff}, x \rangle$, then compute y from x , and then build the output by following the syntax. The world set is computed by composing each of the blocks of this term, using the world-agnostic composition of \mathbf{MW}_{\forall} . It can be seen as a subset of $\{a, b\} \times \{c, c'\} \times \{0, 1, 2, 3\}$ where $\{a, b\}$ corresponds to being on the first or second line of the matching, $\{c, c'\}$ being on the left or right of the sum, and $\{0, 1, 2, 3\}$ being the world set of the Hadamard gate.

We can then show that the reduction of the language matches the equational theory of the Many-Worlds Calculus. First, we recall the notion of orthogonality between pure values and show that two values are orthogonal when they do not match. Then, we show that when the diagrams of orthogonal values are composed together, it reduces to the diagram with empty worlds.

Definition 6.9.8 (Orthogonality on pure values). Two values v_1, v_2 of the same type are said to be orthogonal, noted $v_1 \perp v_2$ if it can be inferred from the following rules:

$$\frac{}{\text{inj}_l v \perp \text{inj}_r v'} \quad \frac{v \perp v'}{\text{inj}_l v \perp \text{inj}_l v'} \quad \frac{v \perp v'}{\text{inj}_r v \perp \text{inj}_r v'}$$

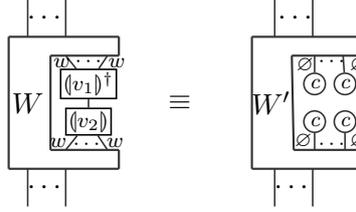
$$\frac{v \perp v'}{\langle v, v_1 \rangle \perp \langle v', v_2 \rangle} \quad \frac{v \perp v'}{\langle v_1, v \rangle \perp \langle v_2, v' \rangle}$$

Lemma 6.9.9. *Given two well-typed values v_1, v_2 and a substitution σ , if $\sigma[v_1] \neq v_2$ then $v_1 \perp v_2$*

Proof. By a straightforward induction on the pattern-matching $\sigma[v_1] = v_2$, the case where $v_1 = x$ or $v_1 = v_2 = ()$ cannot occur otherwise v_1 and v_2 would match. For the other cases we can directly apply the induction hypothesis. \square

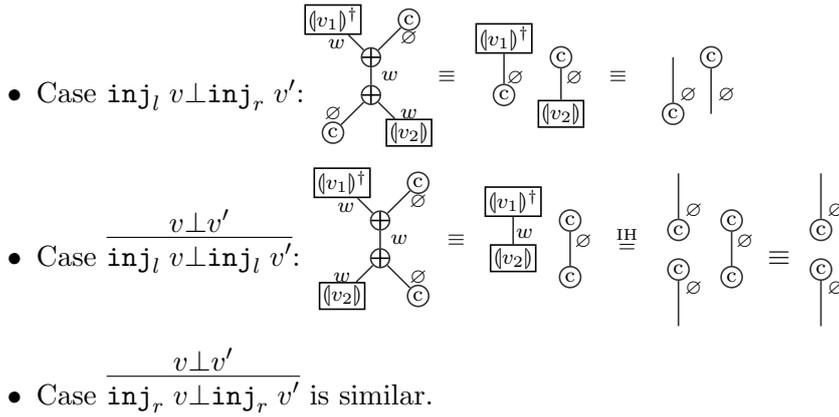
We can then show that orthogonal values, composed together cancel each other out and destroy the worlds on their inputs and outputs:

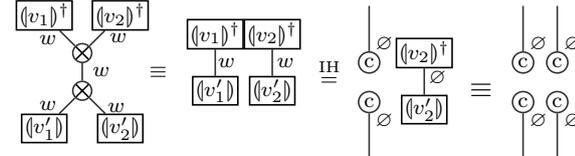
Lemma 6.9.10. *Given two well typed values $\Delta_1 \vdash_e v_1 : A$ and $\Delta_2 \vdash_e v_2 : A$, if $v_1 \perp v_2$ then*



where $W' = W \setminus w$.

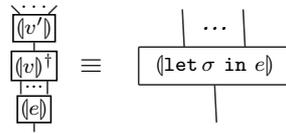
Proof. By induction on $v_1 \perp v_2$, for simplicity we do not draw the context.



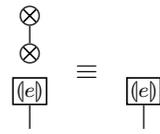
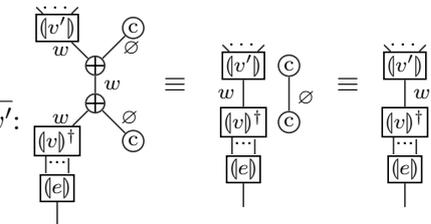
- Case $\frac{v \perp v'}{\langle v, v_1 \rangle \perp \langle v', v_2 \rangle}$: 
- Case $\frac{v \perp v'}{\langle v_1, v \rangle \perp \langle v_2, v' \rangle}$ is similar. □

Conversely, we can show that if two values matches under substitution σ , their composition gives the diagram where σ has been decomposed in a succession of **let**:

Lemma 6.9.11. *Given two well-typed pure values $\Delta \vdash_e v$ and $\Delta \vdash_e v'$ and any expression e such that $\Delta \vdash_e e$, given $\sigma[v] = v'$ such that $\sigma = \{(x_i \mapsto v_i)_{i \in I}\}$, then*

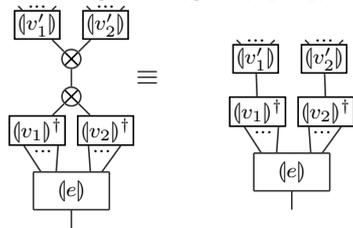


Proof. By induction on $\sigma[v] = v'$:

- Case $\sigma[x] = v'$ then $\sigma = \{x \mapsto v'\}$ and we directly get the desired result: 
- Case $\sigma[()] = ()$: 
- Case $\frac{\sigma[v] = v'}{\sigma[\text{inj}_l v] = \text{inj}_l v'}$: 

and then by induction hypothesis on $\sigma[v] = v'$.

- Case $\sigma[\text{inj}_r v] = \text{inj}_r v'$ is similar.
- Case $\sigma[\langle v_1, v_2 \rangle] = \langle v'_1, v'_2 \rangle$



then by induction hypothesis on $\sigma_1[v_1] = v'_1$ and $\sigma_2[v_2] = v'_2$. This holds due to the fact that the support of σ_1 and σ_2 are disjoint, as imposed by the definition of $\sigma[\langle v_1, v_2 \rangle] = \langle v'_1, v'_2 \rangle$. \square

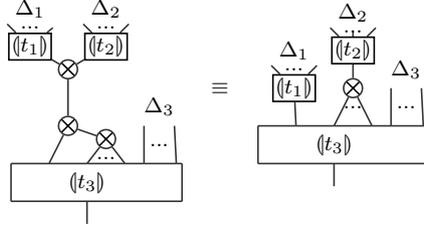
As with μMALL , we consider the explicit substitution reduction defined as in Chapter 4. As with Proof Net and the lambda-calculus with explicit substitution, two terms that reduce with explicit substitution give rise to exactly the same diagram. The only special case is the one of the decomposition of a **let** term into two:

Lemma 6.9.12. *Given a well typed term $\frac{\xi_1}{\Delta \vdash t : A}$ such as $t \rightarrow_{\text{elet}} t'$ and $\frac{\xi_2}{\Delta \vdash t' : A}$ then the interpretation of the two terms are the same: $\llbracket \xi_1 \rrbracket = \llbracket \xi_2 \rrbracket$.*

Proof. By induction on $\rightarrow_{\text{elet}}$:

- $u_1 = \text{let } \langle x, p \rangle = \langle t_1, t_2 \rangle \text{ in } t_3 \rightarrow_{\text{elet}} \text{let } x = t_1 \text{ in let } p = t_2 \text{ in } t_3 = u_2$

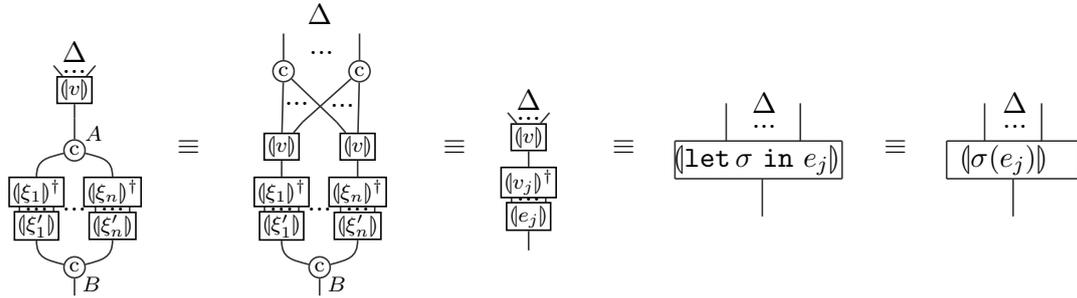
Then



- All the other case are direct as the translation of both ξ_1 and ξ_2 gives exactly the same diagrams. \square

Lemma 6.9.13 (Evaluation of an iso). *Given $\vdash_{\omega} \{v_1 \leftrightarrow e_1 \mid \dots \mid v_n \leftrightarrow e_n\} : A \leftrightarrow B$ and $\Delta \vdash_e v : A$ and $v' : B$ such that $\omega v \rightarrow \sigma(e_j)$ with $\sigma[v_j] = v$ then $\llbracket \omega v \rrbracket = \llbracket \sigma(e_j) \rrbracket$.*

Proof.



Where:

- The first equation is due to the naturality of the contraction.
- The second equality is due to Lemma 6.9.10 and by the first axiom of Figure 6.5 for removing the left-over unary contraction and co-contraction from Lemma 6.9.10.
- The third equation is due to Lemma 6.9.11.
- The fourth equation is due to Property 6.9.3. □

Theorem 6.9.14 (Soundness). *Given two well-typed terms t_1, t_2 , if $t_1 \rightarrow t_2$ then their interpretation is the same: $\llbracket t_1 \rrbracket = \llbracket t_2 \rrbracket$.*

Proof. By direct application of Lemma 6.9.13, Lemma 6.9.12 and Property 6.9.4 and Property 6.9.3. □

We could also define an operational equivalence between terms and show adequacy, but this is left as future work.

6.10. Conclusion

We introduced a new graphical language with a token-based semantics, along with a worlds labelling system and an equational theory.

This language is a first step towards the unification of languages based on the tensor \otimes and those based on the biproduct \oplus . This allows us to reason about both systems in parallel, and superposition of executions, as shown by the encoding of the Quantum Switch in Example 6.7.5 and the translation from term of the language in Section 6.9.

Following this translation, a natural development of the Many-Worlds calculus consists in accommodating recursion in the language. The question of a complete equational theory for the language on mixed states (e.g. via the discard construction [Car+19]) is also left open. With recursion, one would be able to encode the whole language of [SVV18]. As mentioned in Chapter 4, the language of [SVV18] only have the type of lists as recursive data-type and is very restrictive in the way recursion is handled. Extending the language of Chapter 4 to the quantum case would be more befitting.

Finally, while our language allows for quantum control, it does not entirely capture another language that aims at formalizing quantum control, namely the PBS-Calculus [CP20]. How and in which context could we capture the PBS-Calculus is left for future work.

Conclusion

Summary. In this thesis, we studied the question of *reversible and quantum control* along with *pure quantum types* and its *Curry-Howard correspondence*. We gave a first linear and reversible programming language with a typing system based on the logic μ MALL. The language is expressive enough to represent any primitive recursive function and comes with its Curry-Howard correspondence with the logic μ MALL.

We then looked at token-based semantics for quantum graphical languages, first in the case of the ZX-Calculus, which gave us strong intuition on how to adapt it for a new graphical language featuring both a pairing and branching: the Many-Worlds Calculus. We showed how the Many-Worlds can be used to represent pure quantum computation with quantum tests, by encoding both a pure quantum programming language and the quantum switch.

Future Work. The iso language and the Many-Worlds are strongly linked by linear logic. While the iso-language features recursion and inductive types but no quantum computation, the Many-Worlds-Calculus features quantum computation but no recursion and inductive types. A clear extension is then to extend the language of Chapter 4 to the quantum version, as done in [SVV18], while on the other hand the Many-Worlds could be extended in order to handle recursive types and recursion. Then, the Many-Worlds-Calculus would serve as a direct denotational model of the language, as shown in Section 6.9. Then could come the question of coinductive types. In [CDVP21], the authors explore a variant of the ZX-Calculus with *delayed traces*, allowing to consider *streams* of qubits to flow in the diagram indefinitely, while in [DS19], the authors explore proof nets for μ MALL. Those two approaches could be used for the extension of the Many-Worlds-Calculus to the case of inductive and coinductive types.

The Many-Worlds Calculus could also serve as a denotational model for other similar logic or programming language with effects, as [DD22]. A collaboration is currently planned on this subject.

As the Many-Worlds-Calculus is parametrized by a commutative semiring, it is not, in essence, necessary quantum. It could therefore also be used to represent any kind of algebraic branching, be it non-deterministic or probabilistic. On this note, seeing how the Many-Worlds-Calculus adapt to probabilistic programming and their denotational semantics could be interesting.

Bibliography

- [Abb+20] Alastair A Abbott et al. “Communication through coherent control of quantum channels”. In: *Quantum* 4 (2020), p. 333.
- [AC04] Samson Abramsky and Bob Coecke. “A categorical semantics of quantum protocols”. In: *Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science, 2004*. IEEE. 2004, pp. 415–425.
- [AC09] Samson Abramsky and Bob Coecke. “Categorical quantum mechanics”. In: *Handbook of quantum logic and quantum structures 2* (2009), pp. 261–325.
- [AD06] Samson Abramsky and Ross Duncan. “A categorical quantum logic”. In: *Mathematical Structures in Computer Science* 16.3 (2006), pp. 469–489.
- [AG05] Thorsten Altenkirch and Jonathan Grattage. “A Functional Quantum Programming Language”. In: *Proceedings of the 20th Symposium on Logic in Computer Science, LICS’05* (Chicago, Illinois, US.). Ed. by Prakash Panangaden. IEEE. IEEE Computer Society Press, 2005, pp. 249–258. DOI: [10.1109/LICS.2005.1](https://doi.org/10.1109/LICS.2005.1).
- [AG11a] Holger Bock Axelsen and Robert Glück. “A simple and efficient universal reversible Turing machine”. In: *International Conference on Language and Automata Theory and Applications*. Springer. 2011, pp. 117–128.
- [AG11b] Holger Bock Axelsen and Robert Glück. “What do reversible programs compute?” In: *International Conference on Foundations of Software Science and Computational Structures*. Springer. 2011, pp. 42–56.
- [AK16] Holger Bock Axelsen and Robin Kaarsgaard. “Join Inverse Categories as Models of Reversible Recursion”. In: *Proceedings of the 19th International Conference on Foundations of Software Science and Computation Structures (FOSSACS’16)*. Ed. by Bart Jacobs and Christof Löding. Vol. 9634. Lecture Notes in Computer Science. Eindhoven, The Netherlands: Springer, 2016, pp. 73–90. DOI: [10.1007/978-3-662-49630-5_5](https://doi.org/10.1007/978-3-662-49630-5_5).
- [AL95] Andrea Asperti and Cosimo Laneve. “Paths, computations and labels in the λ -calculus”. In: *Theoretical Computer Science* 142.2 (1995), pp. 277–297.
- [Ama+20] Bogdan Aman et al. “Foundations of Reversible Computation”. In: *Reversible Computation: Extending Horizons of Computing - Selected Results of the COST Action IC1405*. Ed. by Irek Ulidowski et al. Vol. 12070. Lecture Notes in Computer Science. Springer, 2020, pp. 1–40. ISBN: 978-3-030-47360-0. DOI: [10.1007/978-3-030-47361-7_1](https://doi.org/10.1007/978-3-030-47361-7_1).

Bibliography

- [Amy18] Matthew Amy. “Towards Large-scale Functional Verification of Universal Quantum Circuits”. In: *Proceedings 15th International Conference on Quantum Physics and Logic, QPL 2018, Halifax, Canada, 3-7th June 2018*. Ed. by Peter Selinger and Giulio Chiribella. Vol. 287. EPTCS. 2018, pp. 1–21. URL: <https://doi.org/10.4204/EPTCS.287.1>.
- [Bac14] Miriam Backens. “The ZX-Calculus is Complete for Stabilizer Quantum Mechanics”. In: *New Journal of Physics* 16.9 (2014), p. 093021. DOI: [10.1088/1367-2630/16/9/093021](https://doi.org/10.1088/1367-2630/16/9/093021).
- [Bac+20] Miriam Backens et al. *There and back again: A circuit extraction tale*. 2020. arXiv: [2003.01664](https://arxiv.org/abs/2003.01664) [quant-ph].
- [Bae08] David Baelde. “A Linear Approach to the Proof-Theory of Least and Greatest Fixed Points”. Theses. École Polytechnique, 2008.
- [Bae12] David Baelde. “Least and greatest fixed points in linear logic”. In: *ACM Transactions on Computational Logic (TOCL)* 13.1 (2012), pp. 1–44.
- [Bae+20] David Baelde et al. “Bouncing threads for infinitary and circular proofs”. In: *arXiv preprint arXiv:2005.08257* (2020).
- [Bar84] Henk Barendregt. “The lambda calculus: its syntax and semantics”. In: *Studies in logic and the foundations of Mathematics* (1984).
- [BC13] Yves Bertot and Pierre Castéran. *Interactive theorem proving and program development: Coq’Art: the calculus of inductive constructions*. Springer Science & Business Media, 2013.
- [BDS16] David Baelde, Amina Doumane, and Alexis Saurin. “Infinitary Proof Theory: the Multiplicative Additive Case”. In: *25th EACSL Annual Conference on Computer Science Logic (CSL 2016)*. Ed. by Jean-Marc Talbot and Laurent Regnier. Vol. 62. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2016, 42:1–42:17. ISBN: 978-3-95977-022-4. URL: <http://drops.dagstuhl.de/opus/volltexte/2016/6582>.
- [Bea+19] Niel de Beaudrap et al. “Pauli Fusion: a computational model to realise quantum transformations from ZX terms”. In: *QPL’19 : International Conference on Quantum Physics and Logic*. 12 pages + appendices. Los Angeles, United States, June 2019. URL: <https://hal.archives-ouvertes.fr/hal-02413388>.
- [Ben00] Charles H. Bennett. “Notes on the history of reversible computation”. In: *IBM Journal of Research and Development* 44.1 (2000), pp. 270–278. DOI: [10.1147/rd.441.0270](https://doi.org/10.1147/rd.441.0270).
- [Ben73] Charles H Bennett. “Logical reversibility of computation”. In: *IBM journal of Research and Development* 17.6 (1973), pp. 525–532.

Bibliography

- [Bér+12] Antoine Bérut et al. “Experimental verification of Landauer’s principle linking information and thermodynamics”. In: *Nature* 483.7388 (2012), pp. 187–189.
- [BH20] Niel de Beaudrap and Dominic Horsman. “The ZX calculus is a language for surface code lattice surgery”. In: *Quantum* 4 (Jan. 2020), p. 218. ISSN: 2521-327X. URL: <https://doi.org/10.22331/q-2020-01-09-218>.
- [BK19] Miriam Backens and Aleks Kissinger. “ZH: A Complete Graphical Calculus for Quantum Computations Involving Classical Non-linearity”. In: *Proceedings of the 15th International Conference on Quantum Physics and Logic, Halifax, Canada, 3-7th June 2018*. Ed. by Peter Selinger and Giulio Chiribella. Vol. 287. Electronic Proceedings in Theoretical Computer Science. 2019, pp. 23–42. URL: <https://doi.org/10.4204/EPTCS.287.2>.
- [Car12] Michael Kirkedal Carøe. “Design of Reversible Computing Systems”. PhD thesis. University of Copenhagen, Denmark, 2012.
- [Car+19] Titouan Carrette et al. “Completeness of Graphical Languages for Mixed States Quantum Mechanics”. In: *46th International Colloquium on Automata, Languages, and Programming (ICALP 2019)*. Ed. by Christel Baier et al. Vol. 132. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019, 108:1–108:15. ISBN: 978-3-95977-109-2. URL: <http://drops.dagstuhl.de/opus/volltexte/2019/10684>.
- [CD11] Bob Coecke and Ross Duncan. “Interacting quantum observables: categorical algebra and diagrammatics”. In: *New Journal of Physics* 13.4 (2011), p. 043016.
- [CDH20] Cole Comfort, Antonin Delpuch, and Jules Hedges. *Sheet diagrams for bimonoidal categories*. 2020. URL: <https://arxiv.org/abs/2010.13361>.
- [CDP08] G. Chiribella, G. M. D’Ariano, and P. Perinotti. “Transforming Quantum Operations: Quantum Supermaps”. In: *EPL (Europhysics Letters)* 83.3 (2008), p. 30004. DOI: [10.1209/0295-5075/83/30004](https://doi.org/10.1209/0295-5075/83/30004).
- [CDVP21] Titouan Carrette, Marc De Visme, and Simon Perdrix. “Graphical language with delayed trace: Picturing quantum computing with finite memory”. In: *2021 36th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. IEEE. 2021, pp. 1–13.
- [Cha+] Christophe Chareton et al. “A Deductive Verification Framework for Circuit-building Quantum Programs”. [arXiv:2003.05841](https://arxiv.org/abs/2003.05841). To appear in *Proceedings of ESOP’21*.
- [Cha+22] Kostia Chardonnet et al. “The Many-Worlds Calculus”. working paper or preprint. Aug. 2022. URL: <https://hal.archives-ouvertes.fr/hal-03654190>.
- [Chi+13] Giulio Chiribella et al. “Quantum computations without definite causal structure”. In: *Physical Review A* 88.2 (2013), p. 022318.

Bibliography

- [CHP19] Titouan Carette, Dominic Horsman, and Simon Perdrix. “SZX-Calculus: Scalable Graphical Quantum Reasoning”. In: *44th International Symposium on Mathematical Foundations of Computer Science (MFCS 2019)*. Ed. by Peter Rossmanith, Pinar Heggernes, and Joost-Pieter Katoen. Vol. 138. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019, 55:1–55:15. ISBN: 978-3-95977-117-7. URL: <http://drops.dagstuhl.de/opus/volltexte/2019/10999>.
- [Chu36] Alonzo Church. “An unsolvable problem of elementary number theory”. In: *American journal of mathematics* 58.2 (1936), pp. 345–363.
- [Chu40] Alonzo Church. “A formulation of the simple theory of types”. In: *The journal of symbolic logic* 5.2 (1940), pp. 56–68.
- [Chu41] Alonzo Church. *The calculi of lambda-conversion*. 6. Princeton University Press, 1941.
- [CK17] Bob Coecke and Aleks Kissinger. *Picturing Quantum Processes: A First Course in Quantum Theory and Diagrammatic Reasoning*. Cambridge University Press, 2017. ISBN: 9781316219317. DOI: [10.1017/9781316219317](https://doi.org/10.1017/9781316219317).
- [CL02] J. Robin B. Cockett and Stephen Lack. “Restriction Categories I: Categories of Partial Maps”. In: *Theoretical Computer Science* 270.1 (2002), pp. 223–259. DOI: [10.1016/S0304-3975\(00\)00382-0](https://doi.org/10.1016/S0304-3975(00)00382-0).
- [CL03] J. Robin B. Cockett and Stephen Lack. “Restriction categories II: partial map classification”. In: *Theoretical Computer Science* 294.1 (2003), pp. 61–102. DOI: [10.1016/S0304-3975\(01\)00245-6](https://doi.org/10.1016/S0304-3975(01)00245-6).
- [CL07] Robin Cockett and Stephen Lack. “Restriction Categories III: Colimits, Partial Limits and Extensivity”. In: *Mathematical Structures in Computer Science* 17.4 (2007), pp. 775–817. DOI: [10.1017/S0960129507006056](https://doi.org/10.1017/S0960129507006056).
- [Clé+22a] Alexandre Clément et al. “A Complete Equational Theory for Quantum Circuits”. In: *arXiv preprint arXiv:2206.10577* (2022).
- [Clé+22b] Alexandre Clément et al. *LOv-Calculus: A Graphical Language for Linear Optical Quantum Circuits*. 2022. URL: <https://arxiv.org/abs/2204.11787>.
- [CLV21] Kostia Chardonnet, Louis Lemonnier, and Benoît Valiron. “Categorical Semantics of Reversible Pattern-Matching”. In: *Proceedings 37th Conference on Mathematical Foundations of Programming Semantics, Hybrid: Salzburg, Austria and Online, 30th August - 2nd September, 2021*. Ed. by Ana Sokolova. Vol. 351. Electronic Proceedings in Theoretical Computer Science. Open Publishing Association, 2021, pp. 18–33. DOI: [10.4204/EPTCS.351.2](https://doi.org/10.4204/EPTCS.351.2).
- [CP12] Bob Coecke and Simon Perdrix. “Environment and Classical Channels in Categorical Quantum Mechanics”. In: *Logical Methods in Computer Science* Volume 8, Issue 4 (2012). URL: <https://lmcs.episciences.org/719>.

Bibliography

- [CP20] Alexandre Clément and Simon Perdrix. “PBS-calculus: A graphical language for coherent control of quantum computations”. In: *arXiv preprint arXiv:2002.09387* (2020).
- [CSV20] Kostia Chardonnet, Alexis Saurin, and Benoît Valiron. “Toward a Curry-Howard Equivalence for Linear, Reversible Computation - Work-in-Progress”. In: *Proceedings of the 12th International Conference on Reversible Computation (RC 2020)*. Ed. by Ivan Lanese and Mariusz Rawski. Vol. 12227. Lecture Notes in Computer Science. Springer, 2020, pp. 144–152. ISBN: 978-3-030-52481-4. DOI: [10.1007/978-3-030-52482-1_8](https://doi.org/10.1007/978-3-030-52482-1_8).
- [Cur34] Haskell B Curry. “Functionality in combinatory logic”. In: *Proceedings of the National Academy of Sciences* 20.11 (1934), pp. 584–590.
- [CVV21] Kostia Chardonnet, Benoît Valiron, and Renaud Vilmart. “Geometry of Interaction for ZX-Diagrams”. In: *Proceedings of the 46th International Symposium on Mathematical Foundations of Computer Science, MFCS 2021* (Tallinn, Estonia). Ed. by Filippo Bonchi and Simon J. Puglisi. Vol. 202. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2021, 30:1–30:16. ISBN: 978-3-95977-201-3. DOI: [10.4230/LIPIcs.MFCS.2021.30](https://doi.org/10.4230/LIPIcs.MFCS.2021.30).
- [Dal17] Dal Lago, Ugo and Faggian, Claudia and Valiron, Benoît and Yoshimizu, Akira. “The Geometry of Parallelism: Classical, Probabilistic, and Quantum Effects”. In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. POPL 2017. New York, NY, USA: Association for Computing Machinery, 2017, 833–845. ISBN: 9781450346603. URL: <https://doi.org/10.1145/3009837.3009859>.
- [DCM22] Alejandro Díaz-Caro and Octavio Malherbe. *Semimodules and the (syntactically-)linear lambda calculus*. 2022. DOI: [10.48550/ARXIV.2205.02142](https://doi.org/10.48550/ARXIV.2205.02142). URL: <https://arxiv.org/abs/2205.02142>.
- [DD22] Alejandro Díaz-Caro and Gilles Dowek. “Linear lambda-calculus is linear”. In: *CoRR* abs/2201.11221 (2022). arXiv: [2201.11221](https://arxiv.org/abs/2201.11221). URL: <https://arxiv.org/abs/2201.11221>.
- [DG18] Ross Duncan and Liam Garvie. “Verifying the Smallest Interesting Colour Code with Quantomatic”. In: *Proceedings 14th International Conference on Quantum Physics and Logic, Nijmegen, The Netherlands, 3-7 July 2017*. Ed. by Bob Coecke and Aleks Kissinger. Vol. 266. Electronic Proceedings in Theoretical Computer Science. 2018, pp. 147–163. URL: <https://doi.org/10.4204/EPTCS.266.10>.
- [DL14] Ross Duncan and Maxime Lucas. “Verifying the Steane code with Quantomatic”. In: *Proceedings of the 10th International Workshop on Quantum Physics and Logic, Castelldefels (Barcelona), Spain, 17th to 19th July 2013*. Ed. by Bob Coecke and Matty Hoban. Vol. 171. Electronic Proceedings in Theoretical Computer Science. Open Publishing Association, 2014, pp. 33–49. DOI: [10.4204/EPTCS.171.4](https://doi.org/10.4204/EPTCS.171.4).

Bibliography

- [DLF11] Ugo Dal Lago and Claudia Faggian. “On multiplicative linear logic, modality and quantum circuits”. In: *QPL* 95 (2011), pp. 55–66.
- [Dou17] Amina Doumane. “On the infinitary proof theory of logics with fixed points”. Theses. Université Sorbonne Paris Cité, June 2017. URL: <https://hal.archives-ouvertes.fr/tel-01676953>.
- [DP10] Ross Duncan and Simon Perdrix. “Rewriting Measurement-Based Quantum Computations with Generalised Flow”. In: *Lecture Notes in Computer Science* 6199 (2010), pp. 285–296. DOI: [10.1007/978-3-642-14162-1_24](https://doi.org/10.1007/978-3-642-14162-1_24).
- [DR79] Willem-Paul De Roeper. “Recursive program schemes: semantics and proof theory”. In: *Journal of Symbolic Logic* 44.4 (1979).
- [DR99] Vincent Danos and Laurent Regnier. “Reversible, irreversible and optimal λ -machines”. In: *Theoretical Computer Science* 227.1-2 (1999), pp. 79–97.
- [DS19] Abhishek De and Alexis Saurin. “Infinites: The parallel syntax for non-wellfounded proof-theory”. In: *TABLEAUX 2019 - 28th International Conference on Automated Reasoning with Analytic Tableaux and Related Methods*. TABLEAUX 2019 - 28th International Conference on Automated Reasoning with Analytic Tableaux and Related Methods. London, United Kingdom, Sept. 2019. URL: <https://hal.archives-ouvertes.fr/hal-02337286>.
- [Dun04] Ross Duncan. *Believe it or not, Bell states are a model of multiplicative linear logic*. Tech. rep. 2004.
- [Dun06] Ross Duncan. “Types for quantum computing”. In: (2006).
- [Dun09] Ross Duncan. “Generalized Proof-Nets for Compact Categories with Biproducts”. In: *Semantic Techniques in Quantum Computation*. Ed. by Simon Gay and Ian Mackie. Cambridge University Press, 2009. Chap. 3, pp. 70–134. ISBN: 978-0-521-51374-6.
- [Dun+20] Ross Duncan et al. “Graph-theoretic Simplification of Quantum Circuits with the ZX-calculus”. In: *Quantum* 4 (2020), p. 279.
- [FBW18] Mark Fingerhuth, Tomáš Babej, and Peter Wittek. “Open source software in quantum computing”. In: *PLOS ONE* 13.12 (Dec. 2018), pp. 1–28. URL: <https://doi.org/10.1371/journal.pone.0208561>.
- [FC22] Giovanni de Felice and Bob Coecke. *Quantum Linear Optics via String Diagrams*. 2022. URL: <https://arxiv.org/abs/2204.12985>.
- [FH65] Richard P. Feynman and A. R. Hibbs. *Quantum Mechanics and Path Integrals*. McGraw-Hill Publishing Company, 1965. ISBN: 0-07-020650-3.

Bibliography

- [FS13] Jérôme Fortier and Luigi Santocanale. “Cuts for circular proofs: semantics and cut-elimination”. In: *Computer Science Logic 2013 (CSL 2013)*. Ed. by Simona Ronchi Della Rocca. Vol. 23. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2013, pp. 248–262. ISBN: 978-3-939897-60-6. URL: <http://drops.dagstuhl.de/opus/volltexte/2013/4201>.
- [FT82] Edward Fredkin and Tommaso Toffoli. “Conservative logic”. In: *International Journal of theoretical physics* 21.3 (1982), pp. 219–253.
- [Gab+13] Marco Gaboardi et al. “Linear dependent types for differential privacy”. In: *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 2013, pp. 357–370.
- [Gen35] Gerhard Gentzen. “Untersuchungen über das logische schließen. I.” In: *Mathematische zeitschrift* 35 (1935).
- [Gir06] Jean-Yves Girard. “Geometry of interaction IV: the feedback equation”. In: *Logic Colloquium*. Vol. 3. 2006, pp. 76–117.
- [Gir11] Jean-Yves Girard. “Geometry of interaction V: logic in the hyperfinite factor”. In: *Theoretical Computer Science* 412.20 (2011), pp. 1860–1883.
- [Gir13] Jean-Yves Girard. “Geometry of interaction VI: a blueprint for transcendental syntax”. In: *preprint* (2013).
- [Gir87] Jean-Yves Girard. “Linear logic”. In: *Theoretical computer science* 50.1 (1987), pp. 1–101.
- [Gir88] Jean-Yves Girard. “Geometry of interaction II: deadlock-free algorithms”. In: *International Conference on Computer Logic*. Springer. 1988, pp. 76–93.
- [Gir89a] Jean-Yves Girard. “Geometry of interaction I: interpretation of System F”. In: *Studies in Logic and the Foundations of Mathematics*. Vol. 127. Elsevier, 1989, pp. 221–260.
- [Gir89b] Jean-Yves Girard. “Towards a geometry of interaction”. In: *Contemporary Mathematics* 92.69-108 (1989), p. 6.
- [Gir95] Jean-Yves Girard. “Geometry of interaction III: accommodating the additives”. In: *London Mathematical Society Lecture Note Series* (1995), pp. 329–389.
- [Gir96] Jean-Yves Girard. “Proof-nets: the parallel syntax for proof-theory”. In: *Lecture Notes in Pure and Applied Mathematics* (1996), pp. 97–124.
- [GK18] Robert Glück and Robin Kaarsgaard. “A categorical foundation for structured reversible flowchart languages: Soundness and adequacy”. In: *Log. Methods Comput. Sci.* 14.3 (2018). DOI: [10.23638/LMCS-14\(3:16\)2018](https://doi.org/10.23638/LMCS-14(3:16)2018).

Bibliography

- [GKY19] Robert Glück, Robin Kaarsgaard, and Tetsuo Yokoyama. “Reversible Programs Have Reversible Semantics”. In: *Formal Methods. FM 2019 International Workshops - Porto, Portugal, October 7-11, 2019, Revised Selected Papers, Part II*. Ed. by Emil Sekerinski et al. Vol. 12233. Lecture Notes in Computer Science. Springer, 2019, pp. 413–427. ISBN: 978-3-030-54996-1. DOI: [10.1007/978-3-030-54997-8_26](https://doi.org/10.1007/978-3-030-54997-8_26).
- [Gre+13] Alexander S. Green et al. “Quipper: A Scalable Quantum Programming Language”. In: *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI’13* (Seattle, WA, USA). Ed. by Hans-Juergen Boehm and Cormac Flanagan. ACM, 2013, pp. 333–342. ISBN: 978-1-4503-2014-6. DOI: [10.1145/2491956.2462177](https://doi.org/10.1145/2491956.2462177).
- [Gri89] Timothy G Griffin. “A formulae-as-type notion of control”. In: *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 1989, pp. 47–58.
- [Gro96] Lov K Grover. “A fast quantum mechanical algorithm for database search”. In: *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*. 1996, pp. 212–219.
- [Had15] Amar Hadzihasanovic. “A Diagrammatic Axiomatisation for Qubit Entanglement”. In: *2015 30th Annual ACM/IEEE Symposium on Logic in Computer Science*. 2015, pp. 573–584. URL: <http://dx.doi.org/10.1109/LICS.2015.59>.
- [HH16] Ichiro Hasuo and Naohiko Hoshino. “Semantics of Higher-Order Quantum Computation via Geometry of Interaction”. In: *CoRR* abs/1605.05079 (2016). arXiv: [1605.05079](https://arxiv.org/abs/1605.05079). URL: <http://arxiv.org/abs/1605.05079>.
- [Hil11] Anne Hillebrand. “Quantum Protocols involving Multiparticle Entanglement and their Representations”. MA thesis. University of Oxford, 2011. URL: <https://www.cs.ox.ac.uk/people/bob.coecke/Anne.pdf>.
- [HK15] Chris Heunen and Martti Karvonen. “Reversible Monadic Computing”. In: *Proceedings of the 31st Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXI)*. Ed. by Dan Ghica. Vol. 319. Electronic Notes in Theoretical Computer Science. Nijmegen, The Netherlands, 2015, pp. 217–237. DOI: [10.1016/j.entcs.2015.12.014](https://doi.org/10.1016/j.entcs.2015.12.014).
- [HKK18] Chris Heunen, Robin Kaarsgaard, and Martti Karvonen. “Reversible Effects as Inverse Arrows”. In: *Proceedings of the 34th Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXIV)*. Ed. by Sam Staton. Vol. 341. Electronic Notes in Theoretical Computer Science. Dalhousie University, Halifax, Canada: Elsevier, 2018, pp. 179–199. DOI: [10.1016/j.entcs.2018.11.009](https://doi.org/10.1016/j.entcs.2018.11.009).

Bibliography

- [HNW18] Amar Hadzihasanovic, Kang Feng Ng, and Quanlong Wang. “Two Complete Axiomatisations of Pure-state Qubit Quantum Computing”. In: *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*. LICS ’18. Oxford, United Kingdom: ACM, 2018, pp. 502–511. ISBN: 978-1-4503-5583-4. URL: <http://doi.acm.org/10.1145/3209108.3209128>.
- [How80] William A Howard. “The formulae-as-types notion of construction”. In: *To HB Curry: essays on combinatory logic, lambda calculus and formalism* 44 (1980), pp. 479–490.
- [HV19] Chris Heunen and Jamie Vicary. *Categories for Quantum Theory*. Oxford University Press, 2019. URL: <https://doi.org/10.1093%2Foso%2F9780198739623.001.0001>.
- [HVG03] D. Hughes and R. Van Glabbeek. “Proof nets for unit-free multiplicative-additive linear logic (extended abstract)”. In: *18th Annual IEEE Symposium of Logic in Computer Science, 2003. Proceedings*. Ottawa, Ont., Canada: IEEE Comput. Soc, 2003. ISBN: 978-0-7695-1884-8. DOI: [10.1109/LICS.2003.1210039](https://doi.org/10.1109/LICS.2003.1210039). URL: <http://ieeexplore.ieee.org/document/1210039/> (visited on 09/29/2022).
- [JKT18] Petur Andrias Højgaard Jacobsen, Robin Kaarsgaard, and Michael Kirkedal Thomsen. “CoreFun : A Typed Functional Reversible Core Language”. In: *Reversible Computation - 10th International Conference, RC 2018, Leicester, UK, September 12-14, 2018, Proceedings*. Ed. by Jarkko Kari and Irek Ulidowski. Vol. 11106. Lecture Notes in Computer Science. Springer, 2018, pp. 304–321. DOI: [10.1007/978-3-319-99498-7_21](https://doi.org/10.1007/978-3-319-99498-7_21).
- [JPV18a] Emmanuel Jeandel, Simon Perdrix, and Renaud Vilmart. “A Complete Axiomatisation of the ZX-Calculus for Clifford+T Quantum Mechanics”. In: *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*. LICS ’18. Oxford, United Kingdom: ACM, 2018, pp. 559–568. ISBN: 978-1-4503-5583-4. URL: <https://doi.acm.org/10.1145/3209108.3209131>.
- [JPV18b] Emmanuel Jeandel, Simon Perdrix, and Renaud Vilmart. “Diagrammatic Reasoning Beyond Clifford+T Quantum Mechanics”. In: *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*. LICS ’18. Oxford, United Kingdom: ACM, 2018, pp. 569–578. ISBN: 978-1-4503-5583-4. URL: <http://doi.acm.org/10.1145/3209108.3209139>.
- [JPV19] Emmanuel Jeandel, Simon Perdrix, and Renaud Vilmart. “A Generic Normal Form for ZX-Diagrams and Application to the Rational Angle Completeness”. In: *2019 34th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. 2019, pp. 1–10. DOI: [10.1109/LICS.2019.8785754](https://doi.org/10.1109/LICS.2019.8785754).

Bibliography

- [JS12] Roshan P. James and Amr Sabry. “Information effects”. In: *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL’12)*. Ed. by John Field and Michael Hicks. Philadelphia, Pennsylvania, USA: ACM, 2012, pp. 73–84. ISBN: 978-1-4503-1083-3. DOI: [10.1145/2103656.2103667](https://doi.org/10.1145/2103656.2103667).
- [JS14] Roshan P. James and Amr Sabry. “Theseus: A High-Level Language for Reversible Computing”. Draft, available at <https://legacy.cs.indiana.edu/~sabry/papers/theseus.pdf>. 2014.
- [JS91] André Joyal and Ross Street. “The geometry of tensor calculus, I”. In: *Advances in mathematics* 88.1 (1991), pp. 55–112.
- [Jun+17] Ralf Jung et al. “RustBelt: Securing the foundations of the Rust programming language”. In: *Proceedings of the ACM on Programming Languages* 2.POPL (2017), pp. 1–34.
- [Kaa19a] Robin Kaarsgaard. “Condition/Decision Duality and the Internal Logic of Extensive Restriction Categories”. In: *Proceedings of the 35th Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXV)*. Ed. by Barbara König. Vol. 347. Electronic Notes in Theoretical Computer Science. London, UK, 2019, pp. 179–202. DOI: [10.1016/j.entcs.2019.09.010](https://doi.org/10.1016/j.entcs.2019.09.010).
- [Kaa19b] Robin Kaarsgaard. “Inversion, Iteration, and the Art of Dual Wielding”. In: *Proceedings of the 11th International Conference on Reversible Computation (RC 2019)*. Ed. by Michael Kirkedal Thomsen and Mathias Soeken. Vol. 11497. Lecture Notes in Computer Science. Lausanne, Switzerland: Springer, 2019, pp. 34–50. ISBN: 978-3-030-21499-9. DOI: [10.1007/978-3-030-21500-2_3](https://doi.org/10.1007/978-3-030-21500-2_3).
- [KAG17] Robin Kaarsgaard, Holger Bock Axelsen, and Robert Glück. “Join inverse categories and reversible recursion”. In: *Journal of Logical and Algebraic Methods in Programming* 87 (2017), pp. 33–50. ISSN: 2352-2208. DOI: [10.1016/j.jlamp.2016.08.003](https://doi.org/10.1016/j.jlamp.2016.08.003).
- [Kas79] J. Kastl. “Inverse Categories”. In: *Algebraische Modelle, Kategorien und Gruppoide*. Studien zur Algebra und ihre Anwendungen, Band 7. Berlin, Akademie-Verlag, 1979, pp. 51–60.
- [Koz83] Dexter Kozen. “Results on the propositional μ -calculus”. In: *Theoretical computer science* 27.3 (1983), pp. 333–354.
- [KR21] Robin Kaarsgaard and Mathys Rennela. *Join inverse rig categories for reversible functional programming, and beyond*. Draft, available at [arXiv:2105.09929](https://arxiv.org/abs/2105.09929). 2021.

Bibliography

- [KV19] Robin Kaarsgaard and Niccolò Veltri. “En Garde! Unguarded Iteration for Reversible Computation in the Delay Monad”. In: *Proceedings of the 13th International Conference on Mathematics of Program Construction (MPC 2019)*. Ed. by Graham Hutton. Vol. 11825. Lecture Notes in Computer Science. Porto, Portugal: Springer Verlag, 2019, pp. 366–384. ISBN: 978-3-030-33635-6. DOI: [10.1007/978-3-030-33636-3_13](https://doi.org/10.1007/978-3-030-33636-3_13).
- [Lac04] Stephen Lack. “Composing PROPs”. In: *Theory and Applications of Categories*. Vol. 13. 9. 2004, pp. 147–163. URL: <http://www.tac.mta.ca/tac/volumes/13/9/13-09abs.html>.
- [Lan61] Rolf Landauer. “Irreversibility and Heat Generation in the Computing Process”. In: *IBM Journal of Research and Development*. 5.3 (1961), pp. 183–191. DOI: [10.1147/rd.53.0183](https://doi.org/10.1147/rd.53.0183).
- [Lau11] Olivier Laurent. “Théorie de la démonstration”. In: *Cours du Master MPRI* (2011). URL: <https://perso.ens-lyon.fr/olivier.laurent/thdem11.pdf>.
- [Lee+21] Dongho Lee et al. “Concrete categorical model of a quantum circuit description language with measurement”. In: *arXiv preprint arXiv:2110.02691* (2021).
- [Ler09] Xavier Leroy. “Formal verification of a realistic compiler”. In: *Communications of the ACM* 52.7 (2009), pp. 107–115.
- [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.
- [LWK20] Louis Lemonnier, John van de Wetering, and Aleks Kissinger. *Hypergraph simplification: Linking the path-sum approach to the ZH-calculus*. [arXiv:2003.13564](https://arxiv.org/abs/2003.13564). 2020. arXiv: [2003.13564](https://arxiv.org/abs/2003.13564) [quant-ph].
- [LZ15] Ugo Dal Lago and Margherita Zorzi. “Wave-style token machines and quantum lambda calculi”. In: *arXiv preprint arXiv:1502.04774* (2015).
- [Mai+19] Kenji Maillard et al. “The next 700 relational program logics”. In: *Proceedings of the ACM on Programming Languages* 4.POPL (2019), pp. 1–33.
- [Mel09] Paul-André Mellies. “Categorical semantics of linear logic”. In: *Panoramas et synthèses* 27 (2009), pp. 15–215.
- [Mel14] Paul-André Mellies. “Local states in string diagrams”. In: *Rewriting and Typed Lambda Calculi*. Springer, 2014, pp. 334–348.
- [MY07] Kenichi Morita and Yoshikazu Yamaguchi. “A universal reversible Turing machine”. In: *International Conference on Machines, Computations, and Universality*. Springer. 2007, pp. 90–98.
- [NC02] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information*. Cambridge University Press, 2002.

Bibliography

- [New42] Maxwell Herman Alexander Newman. “On theories with a combinatorial definition of” equivalence””. In: *Annals of mathematics* (1942), pp. 223–243.
- [Nor07] Ulf Norell. “Towards a practical programming language based on dependent type theory”. PhD thesis. SE-412 96 Göteborg, Sweden: Department of Computer Science and Engineering, Chalmers University of Technology, 2007.
- [NST18] Rémi Nollet, Alexis Saurin, and Christine Tasson. “Local validity for circular proofs in linear logic with fixed points”. In: *27th EACSL Annual Conference on Computer Science Logic (CSL 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2018.
- [OSG08] Bryan O’Sullivan, Don Stewart, and John Goerzen. “Real World Haskell”. In: *Available online <http://book.realworldhaskell.org/>* (2008).
- [Par69] David Park. “Fixpoint induction and proofs of program properties”. In: *Machine intelligence* 5 (1969).
- [Por+17] Christopher Portmann et al. “Causal Boxes: Quantum Information-Processing Systems Closed Under Composition”. In: *IEEE Transactions on Information Theory* 63.5 (2017), pp. 3277–3305. DOI: [10.1109/TIT.2017.2676805](https://doi.org/10.1109/TIT.2017.2676805).
- [PPR20] Luca Paolini, Mauro Piccolo, and Luca Roversi. “A class of recursive permutations which is primitive recursive complete”. In: *Theoretical Computer Science* 813 (2020), pp. 218–233.
- [PRZ17] Jennifer Paykin, Robert Rand, and Steve Zdancewic. “QWIRE: a core language for quantum circuits”. In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL’17* (Paris, France). Ed. by Giuseppe Castagna and Andrew D. Gordon. ACM, 2017, pp. 846–858. ISBN: 978-1-4503-4660-3. DOI: [10.1145/3009837.3009894](https://doi.org/10.1145/3009837.3009894).
- [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* (San Diego, California, USA). Ed. by Suresh Jagannathan and Peter Sewell. ACM, 2014, pp. 647–658. ISBN: 978-1-4503-2544-8. DOI: [10.1145/2535838.2535879](https://doi.org/10.1145/2535838.2535879).
- [Rey02] John C Reynolds. “Separation logic: A logic for shared mutable data structures”. In: *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*. IEEE. 2002, pp. 55–74.
- [RJ87] Hartley Rogers Jr. *Theory of recursive functions and effective computability*. MIT press, 1987.
- [RS17] Francisco Rios and Peter Selinger. “A categorical model for a quantum circuit description language”. In: *arXiv preprint [arXiv:1706.02630](https://arxiv.org/abs/1706.02630)* (2017).

Bibliography

- [San02] Luigi Santocanale. “A calculus of circular proofs and its categorical semantics”. In: *FOSSACS 2002*. Vol. 2303. FOSSACS 2002, proceedings of the 5th International Conference on Foundations of Software Science and Computation Structures. Grenoble, France: Springer, Apr. 2002, pp. 357–371. URL: <https://hal.archives-ouvertes.fr/hal-01261170>.
- [Sel07] Peter Selinger. “Dagger compact closed categories and completely positive maps”. In: *Electronic Notes in Theoretical computer science* 170 (2007), pp. 139–163.
- [Sel10] Peter Selinger. “A survey of graphical languages for monoidal categories”. In: *New structures for physics*. Springer, 2010, pp. 289–355.
- [Sho99] Peter W Shor. “Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer”. In: *SIAM review* 41.2 (1999), pp. 303–332.
- [Sin+22] Kartik Singhal et al. “Q# as a Quantum Algorithmic Language”. In: *arXiv preprint arXiv:2206.03532* (2022).
- [SM13] Mehdi Saeedi and Igor L. Markov. “Synthesis and Optimization of Reversible Circuits - a Survey”. In: *ACM Computing Surveys* 45.2 (2013), 21:1–21:34. DOI: [10.1145/2431211.2431220](https://doi.org/10.1145/2431211.2431220).
- [Sta15] Sam Staton. “Algebraic Effects, Linearity, and Quantum Programming Languages”. In: *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL’15*. Ed. by Sriram K. Rajamani and David Walker. ACM, 2015, pp. 395–406. DOI: [10.1145/2676726.2676999](https://doi.org/10.1145/2676726.2676999).
- [SU06] Morten Heine Sørensen and Pawel Urzyczyn. *Lectures on the Curry-Howard isomorphism*. Elsevier, 2006.
- [SV06] Peter Selinger and Benoît Valiron. “A lambda calculus for quantum computation with classical control”. In: *Mathematical Structures in Computer Science* 16.3 (2006), pp. 527–552.
- [SV+09] Peter Selinger, Benoit Valiron, et al. “Quantum lambda calculus”. In: *Semantic techniques in quantum computation* (2009), pp. 135–172.
- [SVV18] Amr Sabry, Benoit Valiron, and Juliana Kaizer Vizzotto. “From Symmetric Pattern-Matching to Quantum Control”. In: *Proceedings of the 21st International Conference on Foundations of Software Science and Computation Structures (FOSSACS’18)*. Ed. by Christel Baier and Ugo Dal Lago. Vol. 10803. Lecture Notes in Computer Science. Thessaloniki, Greece: Springer, 2018, pp. 348–364. DOI: [10.1007/978-3-319-89366-2_19](https://doi.org/10.1007/978-3-319-89366-2_19).
- [Swa+16] Nikhil Swamy et al. “Dependent types and multi-monadic effects in F”. In: *Proceedings of the 43rd annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 2016, pp. 256–270.

Bibliography

- [TA15] Michael Kirkedal Thomsen and Holger Bock Axelsen. “Interpretation and programming of the reversible functional language RFUN”. In: *Proceedings of the 27th Symposium on the Implementation and Application of Functional Programming Languages, IFL 2015, Koblenz, Germany, September 14-16, 2015*. Ed. by Ralf Lämmel. ACM, 2015, 8:1–8:13. ISBN: 978-1-4503-4273-5. DOI: [10.1145/2897336.2897345](https://doi.org/10.1145/2897336.2897345).
- [Tof80] Tommaso Toffoli. “Reversible computing”. In: *Automata, Languages and Programming*. Ed. by Jaco de Bakker and Jan van Leeuwen. Berlin, Heidelberg: Springer Berlin Heidelberg, 1980, pp. 632–644. ISBN: 978-3-540-39346-7.
- [Vil19] Renaud Vilmart. “A Near-Minimal Axiomatisation of ZX-Calculus for Pure Qubit Quantum Mechanics”. In: *2019 34th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. 2019, pp. 1–10. DOI: [10.1109/LICS.2019.8785765](https://doi.org/10.1109/LICS.2019.8785765).
- [Vil19] Renaud Vilmart. “ZX-Calculi for Quantum Computing and their Completeness”. Theses. Université de Lorraine, Sept. 2019. URL: <https://hal.archives-ouvertes.fr/tel-02395443>.
- [Vil20] Renaud Vilmart. *The Structure of Sum-Over-Paths, its Consequences, and Completeness for Clifford*. [arXiv:2003.05678](https://arxiv.org/abs/2003.05678). Mar. 2020. arXiv: [2003.05678](https://arxiv.org/abs/2003.05678) [quant-ph].
- [VKB21] Augustin Vanrietvelde, Hlér Kristjánsson, and Jonathan Barrett. “Routed quantum circuits”. In: *Quantum* 5 (2021), p. 503. DOI: [10.22331/q-2021-07-13-503](https://doi.org/10.22331/q-2021-07-13-503).
- [VRH22] Finn Voichick, Robert Rand, and Michael Hicks. “Qunity: A Unified Language for Quantum and Classical Computing”. In: *arXiv preprint arXiv:2204.12384* (2022).
- [Wec+21] Julian Wechs et al. “Quantum Circuits with Classical Versus Quantum Control of Causal Order”. In: *PRX Quantum* 2 (3 2021), p. 030335. DOI: [10.1103/PRXQuantum.2.030335](https://doi.org/10.1103/PRXQuantum.2.030335).
- [Wil+16] Robert Wille et al. “SyReC: A hardware description language for the specification and synthesis of reversible circuits”. In: *Integration, the VLSI Journal* 53 (2016), pp. 39–53. DOI: [10.1016/j.vlsi.2015.10.001](https://doi.org/10.1016/j.vlsi.2015.10.001).
- [YAG12] Tetsuo Yokoyama, Holger Bock Axelsen, and Robert Glück. “Towards a Reversible Functional Language”. In: *Revised Papers of the Third International Workshop on Reversible Computation (RC’11)*. Ed. by Alexis De Vos and Robert Wille. Vol. 7165. Lecture Notes in Computer Science. Gent, Belgium: Springer, 2012, pp. 14–29. DOI: [10.1007/978-3-642-29517-1_2](https://doi.org/10.1007/978-3-642-29517-1_2).
- [YAG16] Tetsuo Yokoyama, Holger Bock Axelsen, and Robert Glück. “Fundamentals of reversible flowchart languages”. In: *Theoretical Computer Science* 611 (2016), pp. 87–115. DOI: [10.1016/j.tcs.2015.07.046](https://doi.org/10.1016/j.tcs.2015.07.046).

Bibliography

- [YG07] Tetsuo Yokoyama and Robert Glück. “A reversible programming language and its invertible self-interpreter”. In: *Proceedings of the 2007 ACM SIG-PLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation, PEPM 2007, Nice, France, January 15-16, 2007*. Ed. by G. Ramalingam and Eelco Visser. 2007, pp. 144–153. DOI: [10.1145/1244381.1244404](https://doi.org/10.1145/1244381.1244404).
- [Yok10] Tetsuo Yokoyama. “Reversible Computation and Reversible Programming Languages”. In: *Proceedings of the Workshop on Reversible Computation (RC'09)*. Ed. by Irek Ulidowski. Vol. 253(6). Electronic Notes in Theoretical Computer Science. York, UK: Elsevier, 2010, pp. 71–81. DOI: [10.1016/j.entcs.2010.02.007](https://doi.org/10.1016/j.entcs.2010.02.007).