Timing Constraints in Quantum Programming Languages

Vitor Fernandes

September 29, 2020

Contents

1 Introduction

1.1 Motivation

Motivated by quantum advantage, *i.e.* the fact that some tasks run faster on quantum computers than on classical, the interest in quantum computation is rapidly increasing. Applications of quantum computing to real-world problems are numerous, ranging from chemistry, to cryptography, to artificial intelligence, etc. Thus, companies like Google, IBM, and Microsoft are trying to developed viable quantum computers. The first two are developing quantum computers based on superconductors while the last one is based on topology. However, we are still far from having a topological quantum computer. On the other hand, superconductor quantum computers are a reality, although faulty.

Like classical computation, in which hardware progressed from faulty to highly reliable, it is predictable that the same can occur to quantum computers. This implies that qubits are going to be stable/coherent enough time for reliable computation. However, current quantum computers are faulty, particularly the qubits have a very short lifetime. Therefore, a strategy to prevent the execution of algorithms working with faulty qubits passes through the development of a quantum programming language with some form of timing constraints.

Actually, aside from a few exceptions, quantum programming languages do *not* possess well-established formal semantics. This makes difficult to engineer quantum programs and mathematically verify that they behave as expected. Worse than this, the few languages that possess a formal semantics assume an ideal quantum computer, which even though reasonable in the classical case, is and will be too strong for quantum computers in the foreseeable future.

1.2 Goal

We aim to join two different areas: quantum programming languages and timing constraints. The development of quantum programming languages started in the nineties with the seminal work of Knill [Kni96] and Ömer [Öme98]. These languages (albeit with no formal semantics) were influential for the subsequent development of a quantum programming language with a formal *operational semantics* by Sanders and Zuliani [SZ00]. The latter is based on Dijkstra's guarded command language and, among other things, includes program constructs for state preparation and measurement; this provides basic tools for the formal development and reasoning of quantum programs. After this result, P. Selinger and B. Valiron made remarkable progress on the area via a series of works that lift the usual tools of programming, such as typing systems, higher-order functions and recursion, into the quantum setting [PSV14, SV08]. The end result is a λ-calculus for quantum computation with both an operational and a denotational semantics. Technically, standard λ-calculus is a well-established foundation for classical programming languages. The quantum extension proposed by P. Selinger and B. Valiron takes an analogous role in the quantum setting. Note, however, that quantum λ-calculus assumes an *ideal* quantum computer.

Regarding timing constraints, programming languages with a notion of time were already introduced in the past (e.g. [Pla10, Höf09]), but they missed crucial, almost mandatory features of programming, such as typing systems, algebraic datatypes and recursion. This problem was addressed in [NBHM16, GJN18] by taking advantage of the famous category-theoretic notion of a monad, intuitively a generic denotational model to study the different features of a given programming paradigm. The monadic nature of this work made possible to establish a bridge between timed computation and modern programming language theory, allowing one to obtain, among other things, typed and higher-order programming languages with timing constraints. This was attested by the publications (e.g. [NBHM16, GJN18]) where, among other things, the authors introduced while loops with Zeno features (a primitive form of recursion) and program operations with time delays.

In sum, the goal of this project is to establish *formal foundations* for quantum programming languages that do not consider an ideal quantum computer.

1.3 The two possible approaches for reaching the goal

For reaching our goal we have two possible approaches: static and dynamic evaluation. The former is associated with the typing system, while the latter is related to semantics.

Let us detail. In sum, a typing system verifies if a term is well-typed. If it is well-typed then the term can be executed according to the rules of the semantics. Otherwise, if the term is not well-typed it is considered invalid and, consequently, cannot be executed. Since typing systems do not execute a term, we say that they perform a static evaluation. On the other hand, the semantics is used to reduce a term to its output, *i.e.* to execute a term and produce its output. Since in the semantics the term is executed, we say that we are doing a dynamic evaluation.

Quantum computation uses qubits to perform operations. However, quantum computers that exist are far from being ideal, *i.e.* the qubits quickly become susceptible to noise. Intuitively, this means that qubits have

a lifetime in which its manipulation is secure. Thus, until an ideal quantum computers exist, we have to deal with this restriction.

To deal with it we can use a static or a dynamic evaluation. The approach for static evaluation is to introduce time on types. That way, when verifying if a term is well-typed we can also check if the qubits have exceeded its lifetime. Three situations arise from here: if a term is well-typed and the lifetime of qubits is not exceeded then the term is valid and is ready to be executed; if a term is not well-typed then is automatically discarded; if a term is well-typed but the lifetime of qubits is not respected then we can discard the term or think on another strategy so that the term can be evaluated (*e.g.* apply noise to the qubits when the term is executed; use the teleportation protocol to transmit the information of the almost decoherent qubit to a new one). The approach for dynamic evaluation is to apply a noise function to qubits after their lifetime exceed. To do that we could use the operational semantics. However dealing with noise in the operational semantics may become to difficult. For that reason, we will introduce noise in the denotational semantics. Sine it gives a mathematical meaning to terms, and noise can be simulated mathematically, then it is easier and more practical to use denotational semantics instead of operational semantics.

1.4 Why λ-calculus?

Developed in the 1930s by Alonzo Church, λ-calculus has as prominent features higher-order functions and recursion. The former allows functions to be interpreted as data, *i.e.* a function can be an input or an output of a program. The latter allows us to describe programs recursively and thus possibly divergent, leading to a famous problem known has the *non-halting problem* [Tur36]. Furthermore, untyped λ-calculus is equivalent Turing machines.

Since its creation, the λ-calculus was improved in different ways: types were introduced to solve the non-halting problem, resulting in what is now called the simply typed λ-calculus [Hin97, Sel08]; formal semantics were developed to reason about programs – more concretely, small-step operational semantics [Plo75], big-step operational semantics [Kah87]; denotational semantics (via *e.g.* category theory [SS71]), the latter allow us the use of mathematical tools to reason about programs; the extension of λ-calculus with features for describing quantum phenomena, through the development of a quantum λ-calculus [SV08].

In sum, λ-calculus is famous due to its versatility and simplicity. Let us detail the former. The λ-calculus supports higher-order functions and recursion. It can be typed, meaning that we can verify if a λ-term is valid or not (static evaluation). It can have formal semantics, which means that we can develop a small-step and a big-step operational semantics (dynamic evaluation) and also a denotational semantics. The first one allows a step-by-step verification of the program's execution. The second one establishes a relation between expressions and its output. The last one uses category theory to interpret λ-calculus, which is of great advantage because we can use sophisticated mathematical tools to reason about programs. λ-calculus harbours a surprisingly simple set of basic primitives that encode the basic features of programming, such as higher-order computation, conditionals, and recursion. Due to the reasons above, we will adopt in this PhD project λ-calculus as our basic programming language on which to study timing constraints in quantum computing.

1.5 Document structure

The document is structured as follows: In Section 2 we outline the untyped λ-calculus developed by Alonzo Church [Chu36] and simply typed λ-calculus as described *e.g.* in [Sel08]. In Section 3 we detail quantum λ-calculus, developed by Peter Selinger and Benoit Valiron [SV08]. In Section 4 we discuss about the two possible approaches to reach the goal and present the work that we developed so far. In Section 5 we present the work plan for this project. In Section 6 we make some conclusions regarding the first year of the PhD. Finally, in appendix we give an introduction of category theory: specifically in Appendix A we focus on simple categorical notions; in Appendix B we focus on two special cases of categories.

2 Simply typed λ-calculus

In this section detail the origins of the λ-calculus and one problem associated to it that lead to the introduction of types. Then, we present a simply typed λ-calculus, as described *e.g.* in [Sel08] (for the interested reader we also recommend [Hin97]). After that, we show a call-by-value operational semantics in its small-step and big-step forms. At last, we lean over denotational semantics, firstly by giving a general categorical 'receipt', and later by using this receipt to provide a simple denotational semantics for the simply type λ-calculus with roots on equational logic.

2.1 λ-calculus

In the 1930s two equivalent theories were developed independently of each other to answer the question 'what is a computable function?': λ-calculus [Chu36] and Turing machines [Tur36]. The former was developed by Alonzo Church, while the latter was developed by Alan Turing.

To form λ-terms, or terms for simplicity, the syntax of the λ-calculus is composed of variables, 'applications', and 'abstractions', respectively denoted as follows:

$$M, N ::= x \mid MN \mid \lambda x.M$$

We assume an infinite set of variables $\mathcal{X}$; the application is decomposed in the *function* and *argument* part, with the former represented by M and the latter by N; the abstraction is also decomposed in the binder (λx) and the scope of the binder or the body of the abstraction (M).

From the abstraction appears the notions of free or bound occurrence and free or bound variables. A variable x is bounded when it appears in the scope of the binder λx. Otherwise it is a free variable. A variable x has a bounded occurrence if it appears in a term $\lambda x.M$. Otherwise it is a free occurrence.

Furthermore, there are a few more notions that come in hand with the λ-calculus: α-equivalence, substitution, and β-reduction. α-equivalence tells us that two terms are α-equivalent if the terms are the same after renaming the bound variables. α-equivalent terms are usually related by the symbol $=_\alpha$. Substitution, differently from α-equivalence, aims to replace free variable occurrences of x by a term N. However, when doing that, we need to be cautious to not bound a variable that was previously free. We denote as $M[N/x]$ the substitution of the free occurrences of x, in the term M, by N. At last, β-reduction allows us to obtain simple terms from complex ones, specifically represents the computational process of reducing terms to their outputs. Formally, according to [Sel08], β-reduction is defined as follows:

Definition 2.1. The single-step β-reduction is the smallest relation $\to_\beta$ on terms satisfying:

$$\begin{array}{lc|lc} (\beta) & \dfrac{}{(\lambda x.M)N \to_\beta M[N/x]} & (\mathbf{cong_1}) & \dfrac{M \to_\beta M'}{MN \to_\beta M'N} \\ (\mathbf{cong_2}) & \dfrac{N \to_\beta N'}{MN \to_\beta MN'} & (\zeta) & \dfrac{M \to_\beta M'}{\lambda x.M \to_\beta \lambda x.M'} \end{array}$$

From the previous definition one can define a 'multi-step' β-reduction relation:

Definition 2.2. The multi-step β-reduction, $\twoheadrightarrow_\beta$, is the reflexive transitive closure of $\to_\beta$. Informally, $M \twoheadrightarrow_\beta M'$ if M reduces to M' in zero or more steps.

From the concept of β-reduction, it arises a new one known as *evaluation strategy*. Since applications are decomposed in functions and arguments, one can decide to evaluate first the arguments and later the function or first the function and later the arguments. The first approach is known as *call-by-value*, and the second as *call-by-name*. If no evaluation strategy is chosen, then the same term has two ways to be evaluated. Nonetheless, a famous theorem *Church-Rosser Theorem* tells us that terms have **unique** outputs even if the respective reduction is non-deterministic.

Theorem 2.3. Let M, N, and P be terms, such that $M \twoheadrightarrow_\beta N$ and $N \twoheadrightarrow_\beta P$. Then there exists a term Z such that $N \twoheadrightarrow_\beta Z$ and $P \twoheadrightarrow_\beta Z$.

The λ-calculus developed by Church **did not contemplate any restriction** *w.r.t.* term formation, which lead to the existence of functions that never stop. This problem is related to the *Halting problem*, and served as motivation to introduce types on λ-calculus.

2.2 Typing the λ-calculus

There are two ways for typing terms: one explored by Church and the other by Curry. The approach taken by Alonzo Church is more strict, in the sense that each term receives an unique type. This is kind of strict because the same term, for example $\lambda x.x$, has different meaning according the type of x. Furthermore, by this approach, terms that cannot be typed, like $\lambda x.xx$, are not even considered in the language. On the other hand, Haskell Curry presented a more relaxed approach. Instead of restricting term formation with certain typing rules, he developed a set of rules that decide after forming a term it can be typed or not. As a consequence, the term $\lambda x.x$ has the same meaning independently of the type of x. Moreover, terms that cannot be typed are still considered in the language.

Recall the syntax of the λ-calculus presented earlier:

$$M, N ::= x \mid MN \mid \lambda x.M$$

The types that are used on terms are as follows:

$$\sigma, \tau ::= a \mid \sigma \to \tau$$

where a is an *atom*, and $\sigma \to \tau$ is a *composite type*.

To give some intuition about how types work, one can see them as sets: a is a set and $\sigma \to \tau$ is the set of functions from σ to τ. Consider a term $M : \sigma \to \tau$. Intuitively this term represents a function, that maps elements in σ to elements in τ; a constant can be represented by the typed term $M : a$, if M does not have free variables.

With the introduction of types emerges the concept of *context* or *environment*: informally, it stores all free variables of a term. It is represented as follows:

$$\Gamma = x_1 : \sigma_1, \ldots, x_n : \sigma_n$$

Furthermore, it is possible to join a context and a typed term together originating a *typing judgment*, as follows:

$$\Gamma \vdash M : \sigma$$

Inference rules use typing judgments to verify if a term can be typed or not. The format of an inference rules is as follows:

$$\frac{\text{premises}}{\text{conclusion}} \quad [\text{conditions}]$$

An inference rule can be seen as an *if-then* statement, *i.e.* if the requirements (premises plus condition) hold then the conclusion is true. Henceforth an inference rule is read from top to bottom. Furthermore, conditions tend to only concern the context.

Table 1 has the rules that are used to verify if a term is well-typed or not.

$$\textbf{(AX)} \quad \frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma}$$

$$\textbf{(}\to \textbf{E)} \quad \frac{\Gamma \vdash P : \sigma \to \tau \quad \Gamma \vdash Q : \sigma}{\Gamma \vdash PQ : \tau}$$

$$\textbf{(}\to \textbf{I)} \quad \frac{\Gamma, x : \sigma \vdash P : \tau}{\Gamma \vdash \lambda x.P : \sigma \to \tau}$$

Table 1: Term formation rules for the simply typed λ-calculus

The rule **(AX)** tells us that a variable to be valid needs to have the same type in the environment and in the term. The rule **(→ E)** says that P must be a a function and Q its a general term (it could be an atom or a composite type). At last, the rule **(→ I)** states the existence of a variable x in the environment and that the environment should be consistent with x, *i.e.* $x : \sigma \notin \Gamma$. If the requirements are valid, then we can create an abstraction, which is a function and therefore its type is a composite one.

The readers familiar with logic probably have noticed that the typing rules have a similar form to some rules presented in propositional logic. For example, the rule **(→ E)** can be seen as the logic rule *modus ponens*. This similarity criystallises into an important result known as the *Curry-Howard isomorphism* [Hin97], which broadly speaking establishes a relation between computer programs and logical proofs.

With the introduction of types in the λ-calculus, we are performing a static analysis, *i.e.* we are looking for malformations on the programs, instead of evaluating them. To evaluate a program we need to reduce it to a value and, by doing this we are performing a dynamic analysis. It is here that enters the semantics on λ-calculus.

2.3 Semantics

We can choose between an operational, denotational and/or axiomatic semantics. The operational semantics specifies steps of the computation. To do that, we can use a *small-step* or a *big-step*. The former reduces the λ-terms by single steps, describing with detail what happens to an expression. The latter directly relates a λ-term to its value. By doing this, we are abstracting away the details of the evaluation while retaining the syntactic nature of the result. The denotational semantics gives a mathematical meaning to programs via an interpretation function. By doing this, we abstract from syntax aspects in expression evaluation and focus mathematical properties of the computation. The axiomatic semantics defines the properties of the computation. In other words, the axiomatic semantics focus on the definition of axioms and inference rules to construct formal proofs of program properties.

To define the operational semantics we use terms without free variables, *i.e.* $FV(M) = \varnothing$. Henceforth, the values are those that can be built from the grammar:

$$V, U ::= x \mid \lambda x.M$$

Our operational semantics, small-step and big-step, follows a call-by-value strategy because in the presence of effects (recall that our goal is to introduce temporal constraints in the quantum λ-calculus) this strategy is more natural.

2.3.1 Small-Step Operational Semantics

Small-step operational semantics [Plo75] aims to give a detailed description of all steps of a computation, which is ideal for debugging. However, it is also difficult to establish notions of equivalence via such a semantics precisely due to this fine-grained view.

An example of a call-by-value small-step operational semantics is given as follows:

Definition 2.4. The single-step β-reduction is the smallest relation $\to$ on terms satisfying

(APP1)	$\dfrac{N \to N'}{MN \to MN'}$	**(APP2)**	$\dfrac{M \to M'}{MV \to M'V}$
(RED)	$\dfrac{}{(\lambda x.M)V \to M[V/x]}$		

Table 2: Rules of a small-step, call-by-value semantics

From the relation $\to$ we now build a 'multiple-step' reduction relation $\twoheadrightarrow$ by induction.

Definition 2.5. The relation $\twoheadrightarrow$ is built by the following rules:

$$\frac{}{M \twoheadrightarrow M} \ (1) \qquad \frac{M \to M' \quad M' \twoheadrightarrow M''}{M \twoheadrightarrow M''} \ (2)$$

What we have done is very similar to what was done in β-reduction (Subsection 2.1). The main difference is that in this case we committed to an evaluation strategy. Due to this fact, the non-determinacy that existed earlier now disappears.

It is possible to define a big-step operational semantics directly, instead of firstly defining a small-step semantics and then extend it by induction.

2.3.2 Big-Step Operational Semantics

Big-step operational semantics or natural semantics [Kah87] relates an expression to its value, *i.e.* a program to its result. In contrast with the small-step, the big-step semantics abstracts intermediate steps of the evaluation. This abstraction makes it easier to define notions of equivalence.

Next we provide a call-by-value big-step operational semantics (Table 3):

The expression $M \Downarrow V$ denotes that a term M reduces to a value V. Attention on **(APP-RED)** rule. In the premises, we have $N \Downarrow U$, *i.e.* we obligate the argument to reduce to a value and then we feed the value (not the term) to the function. This big-step operational semantics is call-by-value because of that detail.

Now that the operational semantics was presented, it remains to present the denotational semantics.

$$\textbf{(VAL1)} \qquad \frac{}{x \Downarrow x}$$

$$\textbf{(VAL2)} \qquad \frac{}{\lambda x.M \Downarrow \lambda x.M}$$

$$\textbf{(APP-RED)} \qquad \frac{M \Downarrow \lambda x.M' \quad N \Downarrow U \quad M'[U/x] \Downarrow V}{MN \Downarrow V}$$

Table 3: Rules of a big-step call-by-value semantics

2.3.3 Denotational Semantics

Denotational semantics gives mathematical meaning to the language and, contrarily to the operational semantics, it totally abstracts from computational steps and of syntactic details. Consequently, it provides easier ways of defining notions of equivalence. Furthermore, according to [AC98], the denotational semantics provide:

- rigorous definitions that abstract away from implementation details, and that can serve as an implementation independent reference
- mathematical tools for proving properties of programs: as in logic, semantic models are guides in designing sound proof rules, that can then be used in automated proof-checkers

To developed a denotational semantics one should follow the following 'receipt' (the order in which the steps appear is not mandatory, but is a recommendation for standardization):

1. Develop a variant typed λ-calculus
2. Find a category that is compatible with the typed λ-calculus developed
3. Derive a set of inference rules based on the type system according to the valuation function

The first point is related with what we have done until now. We began by describing its terms and types (Section 2.1 and Section 2.2). Then we presented the typing rules and its operational semantics (Section 2.2 and Section 2.3).

For the second point, one needs to find a category that is compatible in some way, with the typed λ-calculus developed. From the many that exist, Cartesian closed or monoidal closed categories are commonly used because they support a notion of abstraction. Now, we are going to use Cartesian closed categories to proceed with the illustration of the 'receipt', while monoidal closed categories will be used for later.

At this point is necessary to have some knowledge about category theory, which we already assume. However, in Appendix A there is an introduction for this topic for the readers who are not familiar with it.

We are going to use the category **Set** (of sets and functions), which is Cartesian closed, to interpret the typed λ-calculus. Recall that the typed λ-calculus is composed by terms and types and a category is composed by objects and arrows. To establish a relation between the syntactic elements (terms and types) and the semantics elements (objects and arrows), we use a function, known as *meaning/interpretation function*, usually denoted by $[\![-]\!]$. For example, let M be a term. Then $[\![M]\!]$ reads as the meaning of M under a given interpretation.

Denotational semantics should be *compositional*, *i.e.* a term interpretation should be given by the subterms interpretation. For example, the interpretation of the application term $[\![MN]\!]$, should be given by the interpretation of $[\![M]\!]$ and $[\![N]\!]$. With this, we ticked the second point of our recipe, and we are ready to address the third, *i.e.* derive the rules of the denotational semantics having in mind the typing rules.

Let us recall the terms

$$M, N ::= x \mid MN \mid \lambda x.M$$

and the types

$$\sigma, \tau ::= a \mid \sigma \to \tau$$

of the typed λ-calculus from Section 2.2.

The meaning function maps types to sets and typing judgments to functions. So, each atom typed a is mapped to a set $[\![a]\!]$, and each composite type $\sigma \to \tau$ is mapped to $[\![\sigma \to \tau]\!] = [\![\tau]\!]^{[\![\sigma]\!]}$.

Recall that typing judgments have the form $\Gamma \vdash M : \sigma$ and $\Gamma = x_1 : \sigma_1, \ldots, x_n : \sigma_n$. Henceforth, a typing judgment of the form $x_1 : \sigma_1, \ldots, x_n : \sigma_n \vdash M : \sigma$ will be a function $[\![\sigma_1]\!] \times \cdots \times [\![\sigma_n]\!] \to [\![\sigma]\!]$.

Now, we are ready to define the rules of the denotational semantics (Table 4), having in mind the typing rules in Table 1:

$$\textbf{(AX)}\quad \frac{x:\sigma \text{ in the ith position of } \Gamma}{[\![\Gamma \vdash x:\sigma]\!] = \pi_i : [\![\Gamma]\!] \to [\![\sigma]\!]}$$

$$\textbf{(APP)}\quad \frac{[\![\Gamma \vdash M:\sigma \to \tau]\!] = f : [\![\Gamma]\!] \to [\![\tau]\!]^{[\![\sigma]\!]} \qquad [\![\Gamma \vdash N:\sigma]\!] = g : [\![\Gamma]\!] \to [\![\sigma]\!]}{[\![\Gamma \vdash MN:\tau]\!] = app \circ \langle f, g\rangle : [\![\Gamma]\!] \to [\![\tau]\!]}$$

$$\textbf{(ABS)}\quad \frac{[\![\Gamma, x:\sigma \vdash M:\tau]\!] = f : [\![\Gamma]\!] \times [\![\sigma]\!] \to [\![\tau]\!]}{[\![\Gamma \vdash \lambda x.M:\sigma \to \tau]\!] = \lambda f : [\![\Gamma]\!] \to [\![\tau]\!]^{[\![\sigma]\!]}}$$

Table 4: Denotational semantics rules

where π_x, app, $\langle f, g\rangle$, and λf are introduced in Appendix A.

The rule **(AX)** corresponds to a function that takes an element $[\![\Gamma]\!]$ and an element $x \in [\![\sigma]\!]$ and returns $x \in [\![\sigma]\!]$. In this case, it is perceptible that we are doing a projection of the variable x. The rule **(APP)** corresponds to a function that takes an element $[\![\Gamma]\!]$ and returns $MN \in [\![\tau]\!]$. At last and following the same thought, the rule **(ABS)** corresponds to a function that takes an element in $[\![\Gamma]\!]$ and returns f with the first argument fixed.

With the presentation of the denotational semantics rules, the 'receipt' has come to the end.

There are two important properties relating the denotational and operational semantics that need to be mentioned, *soundness* and *completeness*. Regarding the former, we say that the denotational semantics is sound with respect to the big-step operational semantics if

$$M \Downarrow V \text{ entails } [\![M]\!] = [\![V]\!]$$

Regarding the latter, we say that the denotational semantics is complete with respect to the big-step operational semantics if

$$[\![M]\!] = [\![V]\!] \text{ entails } M \Downarrow V$$

Here, we used the big-step operational semantics, however we could also have used the small-step or, if existent, axioms.

3 Quantum λ-calculus

In this section we briefly explain the importance of quantum computation and present a quantum λ-calculus developed by Peter Selinger and Benoit Valiron in [SV08]. Regarding the latter, we begin describing the main features. Then, we go into detail by presenting the syntax, typing system, semantics (operational and denotational), and a notion of equivalence between terms. The explanation has the format of an overview (some concepts and results of [SV08] are not presented here) because the goal of the document is primarily to give a general picture of our research plan.

3.1 Motivation and basic concepts

Quantum computation is based on a field of physics known as *quantum mechanics* and dedicated to the study of the world in an atomic scale. From this area, new and interesting concepts arose such as: *qubit*, *superposition*, *entanglement*, and the *no-cloning theorem*.

The qubit is the unit of information adopted in quantum computation. Unlike the bit, used in classical computation, which can only assume values 0 and 1, qubits can assume not only those values but also a linear combination of them ($\alpha|0\rangle + \beta|1\rangle$, where α and β are probability amplitudes). By measuring this state, the probability of obtain 0 is $|\alpha|^2$, while to obtain 1 is $|\beta|^2$. From this, one can conclude that the maximal information possible to extract from a qubit is a bit.

An entangled stated is composed by two or more qubits that possess correlated states. To better understand this concept, consider the following simple example with two entangled qubits, where one of them is measured and the other is not.

Example 3.1. Let q_1 and q_2 be two qubits that are entangled, and very distant from each other, *e.g.* q_1 in Portugal and q_2 is in Australia. Now, the question that arises is the following: how many measurements are necessary to discover the values of the two qubits? A person with no knowledge of quantum mechanics will likely say two. An expected answer, because we have two qubits, so one measurement for each. However, we only need to perform one measurement! The key for this mind blowing result is the qubits being entangled.

At last, the no-cloning theorem simply tells that qubits cannot be cloned, unlike classical computation, where we can copy, at will, bits.

Focusing on computer science, the main motivation for quantum computing is related to the concept of *quantum advantage*, *i.e.* the capacity of quantum computers to solve some problems much faster than classical computers. Examples of this are the algorithms of Grover [Gro96] (search algorithm) and Shor [Sho94] (factorization algorithm).

3.2 Quantum λ-calculus and its linear type system

Such like the λ-calculus is the basis for functional programming, it is hoped that the same can occur with the quantum λ-calculus. The latter developed by P. Selinger and B. Valiron [SV08] is based on linear logic. It has classical control, an operational and denotational semantics, and an equivalence method for terms. Regarding the operational semantics, it is probabilistic (due to measurement) and its evaluation strategy is call-by-value, because of effects, such like measurement and unitary operators. With respect to the denotational semantics, it is based on a symmetric monoidal closed category, more concretely the category of vector spaces with completely positive linear maps.

A theory based on pure linear logic forbids the copying and discarding both classical and quantum variables. In terms of category theory, this means that we cannot use products to model the language, implying that Cartesian categories are not suitable for the task. However, it is possible to use tensors. More categorically, we can use monoidal categories to interpret the language.

Regarding term equivalence [SV08], introduces axiomatic, operational, and denotational definitions. It also proves that the axiomatic equivalence is sound with respect to the operational equivalence and the denotational semantics is fully abstract with respect to the operational semantics. Being sound means that if terms are axiomatically equivalent then they are also operationally equivalent. Being fully abstract means that the denotational and operational semantics are equivalent, *i.e.* if something is true in the denotational semantics then it is also true in the operational semantics.

Let us be more concrete and start with a fragment of the syntax of the quantum λ-calculus (the syntax is not completely presented here, since we want to emphasize some terms).

$$M, N, P ::= x \mid MN \mid \lambda x.M \mid \text{if } M \text{ then } N \text{ else } P \mid new \mid meas \mid U \mid 0 \mid 1$$

The first four cases are easily identified. The terms *new*, *meas*, and U denote, respectively, the creation and measurement of qubits, and the application of unitary operators on qubits. At last, 0 and 1 are boolean terms. To simplify the notation, c is often used for denoting the five previous terms.

3.2.1 Types

The next grammar shows the types available for typing terms:

$$A, B ::= bit \mid qbit \mid A \multimap B \mid \top \mid A \otimes B$$

The constant c has associated to it a fixed type A_c: $meas : qbit \multimap bit$; $0, 1 : bit$; $new : bit \multimap qbit$; $U : qbit^{\otimes n} \multimap qbit^{\otimes n}$. The typing judgement is of the form:

$$\Delta \triangleright M : A$$

where Δ is a typing context, M is a term, and A is a type. The typing context is a list of distinct typed variables. The set of the typing rules, having in mind the syntax fragment, is presented in Table 5.

$$\textbf{(ax}_1\textbf{)} \quad \frac{}{x : A \triangleright x : A} \qquad \textbf{(ax}_2\textbf{)} \quad \frac{}{\triangleright c : A_c}$$

$$\textbf{(app)} \quad \frac{\Gamma_1 \triangleright M : A \multimap B \quad \Gamma_2 \triangleright N : A}{\Gamma_1, \Gamma_2 \triangleright MN : B} \qquad (\lambda) \quad \frac{\Delta, x : A \triangleright M : B}{\Delta \triangleright \lambda x.M : A \multimap B}$$

$$\textbf{(if)} \quad \frac{\Gamma_1 \triangleright P : bit \quad \Gamma_2 \triangleright M : A \quad \Gamma_2 \triangleright N : A}{\Gamma_1, \Gamma_2 \triangleright \text{ if } P \text{ then } M \text{ else } N \ : A}$$

Table 5: Fragment of the typing rules

Just like in the classical case, a typing judgment is valid iff it is possible to construct it using the set of rules in Table 5.

Note that the syntax of this language does not allow the construction of quantum states like $\alpha|0\rangle + \beta|1\rangle$. The justification, according to [SV08], is that this notation 'would not lend itself to expressing entangled states'. Thus, the authors present the notion of quantum closure.

Definition 3.2. A *quantum closure* is a triple $[Q, L, M]$ where:

- Q is a normalized vector in $\otimes_{i=1}^{n} \mathbb{C}^2$, for some $n \geq 0$, and it is called a *quantum array*
- L is a bijective function from a set $|L|$ of variables to $\{0, \ldots, n-1\}$, called *linking function*
- M is a term

The dimension of Q is written as $|Q| = n$. When $L(x_i) = i$ for all variables x_i then L can be written as $|x_1 \ldots x_n\rangle$. The meaning of $L(x_i) = i$ is that the variable x_i is bound in the term M to qubit number $L(x_i)$ in the ith-position of the state Q. At last, the pair (Q, L) is called *quantum context*.

α-equivalence can also be extended to quantum closures as follows:

$$[Q, | x \ldots y \ldots z\rangle, M] =_\alpha [Q, | x \ldots y' \ldots z, M[y'/y]] \quad \text{ if } y' \notin FV(M) \cup \{x, \ldots, y, \ldots, z\}.$$

As we will see from the next definition, quantum closures can be typed.

Definition 3.3. A quantum closure $[Q, L, M]$ has type A in the typing context Γ, written $\Gamma \vDash [Q, L, M] : A$, if $| L | \cap | \Gamma | = \varnothing$, $FV(M) \backslash | \Gamma | \subseteq | L |$, and $\Gamma, x_1 : qbit, \ldots, x_k : qbit \triangleright M : A$ is a valid typing judgment, where $\{x_1, \ldots, x_k\} = FV(M) \backslash | \Gamma |$.

A *closed* well-typed quantum closure, also known as *program*, has $|\Gamma| = \varnothing$.

3.3 Semantics

In this section we are going to present the semantics of [SV08]. Concretely, a call-by-value operational semantics and a denotational semantics. Furthermore, we also present some classes of terms equivalence.

3.3.1 Operational Semantics

Since the operational semantics is based on a call-by-value reduction strategy, it is necessary a notion of value (also presented as a fragment):

$$V, W ::= x \mid c \mid \lambda x.M$$

A program of the form $[Q, L, V]$, where V is a value, is a *value program*.

$$[Q, L, (\lambda x.M)V] \to_1^{\beta} [Q, L, M[V/x]]$$

$$\frac{[Q, L, N] \to_p^k [Q', L', N']}{[Q, L, MN] \to_p^k [Q', L', MN']} \qquad \frac{[Q, L, M] \to_p^k [Q', L', M']}{[Q, L, MV] \to_p^k [Q', L', M'V]}$$

$$[Q, L, \text{if } 0 \text{ then } M \text{ else } N] \to_p^{if_0} [Q, L, N]$$

$$[Q, L, \text{if } 1 \text{ then } M \text{ else } N] \to_p^{if_1} [Q, L, M]$$

$$\frac{[Q, L, M] \to_p^k [Q', L', P']}{[Q, L, \text{if } P \text{ then } M \text{ else } N] \to_p^k [Q', L', \text{if } P' \text{ then } M \text{ else } N]}$$

Table 6: Fragment of the operational semantics ('classical rules')

Recall that in the introduction of this subsection, we stated that the operational semantics is probabilistic. In essence, probabilities emerge from measurements.

In the rules, the transition is represented by an arrow $\to_p^{act}$, where p denotes the probability associated with the transition, and *act* identifies the action made. A fragment of the 'classical' rules for the reduction are on Table 6.

Note that in [SV08], the application of unitary operators is defined for more than one qubit. However, since we decided to present a fragment of the syntax, we adopted the action of unitary operators to our setting. The 'quantum' rules are as follows. To simplify notation, for the first two assume that $Q = \sum_j Q_j^0 \otimes \alpha_j \mid 0\rangle \otimes \tilde{Q}_j^0 + \sum_j Q_j^1 \otimes \beta_j \mid 1\rangle \otimes \tilde{Q}_j^1$, where $Q_j^b \in \mathbb{C}^{2^{i-1}}$ and $\tilde{Q}_j^b \in \mathbb{C}^{2^{n-i}}$. Informally, Q is seen as the separation of the qubits whose value is 0 from the ones with value 1. Furthermore, α and β are the probabilities amplitude of the qubit that is going to be measured.

$$[Q, \mid x_1 \dots x_n\rangle, meas(x_i)] \to_{|\alpha|^2}^{m_0} [\sum_j Q_j^0 \otimes \tilde{Q}_j^0, \mid x_1 \dots x_{i-1} x_{i+1} \dots x_n\rangle, 0]$$

$$[Q, \mid x_1 \dots x_n\rangle, meas(x_i)] \to_{|\alpha|^2}^{m_1} [\sum_j Q_j^1 \otimes \tilde{Q}_j^1, \mid x_1 \dots x_{i-1} x_{i+1} \dots x_n\rangle, 1]$$

For the following, let w be a fresh variable:

$$[Q, \mid x_1 \dots x_n\rangle, new(0)] \to_1^{n_0} [Q \otimes \mid 0\rangle, \mid x_1 \dots x_n w\rangle, w]$$
$$[Q, \mid x_1 \dots x_n\rangle, new(1)] \to_1^{n_1} [Q \otimes \mid 1\rangle, \mid x_1 \dots x_n w\rangle, w]$$

At last, let Q' be the result of applying U to a qubit $L(x)$ in Q:

$$[Q, L, U(x)] \to_1^{U} [Q', L, x]$$

Consider now the substitution lemma:

Lemma 3.4. If $\Delta, x : A \triangleright M : B$ and if $\Gamma \triangleright N : A$ then $\Delta, \Gamma \triangleright M[N/x] : B$ is a valid typing judgment.

Differently from the non-linear case, where a substitution lemma only holds when a variable is substituted by a value, in the linear case we can substitute a variable by any term.

When developing a theory with operational semantics and type system, it is important to assure that the transitions respective to the operational semantics preserve types. That way, it is guarantee that the semantics does not generate ill-typed terms. This property is known as *safety property* and it is formally defined as follows:

Theorem 3.5. If P is a valid program of type A, either it is a value or $P \to_p P'$, with P' a valid program of type A.

Recall the notion of quantum array in the definition of quantum closure. The qubits are ordered, and one question reasonable to be asked is: does the order matters? In this case, one can say that the order of qubits in the quantum array is not relevant. Because, by permuting qubits we can change its order without affecting any result.

Definition 3.6. If σ is a permutation of $\{1, \dots, n\}$, σ is extended to $\mathbb{N}$ with $\sigma(j) = j$ for $j > n$, and $\bar{\sigma}$ is defined to be the corresponding permutation of qubits $\bar{\sigma} \mid x_1 \dots x_n \dots x_{n+k}\rangle = \mid x_{\sigma(1)} \dots x_{\sigma(n)} x_{n+1} \dots x_{n+k}\rangle$. It is said that (Q_1, L_1) is σ-equivalent to (Q_2, L_2), written $(Q_1, L_1) =_\alpha^\sigma (Q_2, L_2)$, if Q_1 and Q_2 have the same size, $Q_2 = \bar{\sigma}(Q_1)$ and $L_2 = \sigma^{-1} \circ L_1$. It is defined an equivalence relation called *alpha-equivalence on quantum contexts* by $(Q_1, L_1) =_\alpha (Q_2, L_2)$ if there exists a σ such that $(Q_1, L_1) =_\alpha^\sigma (Q_2, L_2)$.

An important result that is related with permutation, obtained by the authors, is that alpha-equivalence is sound with respect to the semantics:

Lemma 3.7. If $[Q, L, M] \to_p [Q', L', M']$ then $[\bar{\sigma}Q, \sigma L, M] \to_p [\bar{\sigma}Q', \sigma L', M']$

Since L can be written as $|x_1 \dots x_n\rangle$, then instead of $\sigma^{-1} \circ L$ we write σL.

To obtain a direct relation between a program and its result, *i.e.* its value, it was developed a big-step reduction relation.

Definition 3.8. If X is the set of closed valid programs and U the set of values, let $prob_U : X \times U \to [0,1]$ be the map $prob_U(P,V)$ that returns the total probability for a program P to end up on the value V in zero or more steps. This function is called the *big-step reduction* relation.

In the case of a *small-step reduction*, the operation $prob : X \times X \to [0,1]$ for closed programs P, P' is defined as follows:

- if there is a single-step probabilistic reduction $P \to_p P'$, concretely a reduction by a measurement

$$prob(P, P') = p$$

- if V is a value program

$$prob(V, V) = 1$$

- in all other cases

$$prob(P, P') = 0$$

Since we are dealing with probabilities, the sum of all the probabilities from P to its value must be one, *i.e.* for all well-typed P, $\sum_{P' \in X} prob(P, P') = 1$.

The following definition relates the probability of a program, with type *bit*, to its value.

Definition 3.9. If P is a closed well-typed program of type *bit*, and $b \in \{0,1\}$, it is defined $(P \Downarrow b) = \sum_{V \in U_b} prob(P,V)$, where U_b is the set of valid programs with value b. It is said that P evaluates to b with probability $P \Downarrow b$ (note that $P \Downarrow b$ is a number).

The notion of probability is also presented in quantum closures.

Definition 3.10. A *formal probability distribution* of quantum closures is defined to be $\Gamma \vDash \sum_i \rho_i [Q_i, L_i, M_i] : A$, where each $\Gamma \vDash [Q_i, L_i, M_i] : A$ is valid and $\sum \rho_i \leq 1$. The distribution is said to be closed if $|\Gamma| = \varnothing$.

The following lemma shows how the small-step and big-step reduction relations can be curried.

Lemma 3.11. Let $\mathcal{C}Z$ be the set of probability distributions over a given set Z. The small-step reduction relation ($prob : X \times X \to [0,1]$) can be curried to a map

$$prob' : \mathcal{C}X \to \mathcal{C}X$$
$$prob'(\sum_i \alpha_i P_i) = \sum_i \alpha_i \sum_{P' \in X} prob(P_i, P')P'$$

In the case of big-step reduction ($prob_U : X \times U \to [0,1]$), the currying is

$$prob'_U : \mathcal{C}X \to \mathcal{C}U$$
$$prob'_U(\sum_i \alpha_i P_i) = \sum_i \alpha_i prob_U(P_i, V)V$$

3.3.2 Denotational Semantics

The denotational semantics was developed based on a symmetric monoidal closed category (details in Appendix B) called **CPM** (CPM stands for Completely Positive Maps). However, we will follow the authors' approach and thus first introduce the category V, which is the basis for the development of **CPM**. The category V is defined as follows:

- the objects are signatures $\sigma = n_1, \dots, n_k$, *i.e.* finite tuples of positive integers that indicate the dimension of a vector space
- the arrows $\sigma \to \sigma'$ are linear maps $V_\sigma \to V_{\sigma'}$, where $V_{n_1,\dots,n_k} = \mathbb{C}^{n_1 \times n_1} \times \dots \times \mathbb{C}^{n_k \times n_k}$

Since we are working with vector spaces, there exists a tensor product,

$$(n_1, \ldots, n_k) \otimes (m_1, \ldots, m_l) = n_1 m_1, \ldots, n_l m_1, n_2 m_1, \ldots, n_k m_l$$

and a canonical isomorphism $V_{\sigma\otimes\tau} \simeq V_\sigma \otimes V_\tau$. At last, 1 to denote the unit element.

Note that for all vector spaces σ, τ, σ' the property $\mathbf{V}(\sigma \otimes \tau, \sigma') =_\Phi \mathbf{V}(\sigma, \tau \otimes \sigma')$ holds. Consider that $B(\tau)$ is a basis for a vector space V_τ and that $f \in \mathbf{V}(\sigma \otimes \tau, \sigma')$. From f one constructs $g = \Phi(f) \in \mathbf{V}(\sigma, \tau \otimes \sigma')$ by $g(s) = \sum_{b\in B(\tau)} b \otimes f(s \otimes b)$. On the other hand, if $g(s) = \sum_{b\in B(\tau), u\in B(\sigma')} \alpha_{b,u} b \otimes u$, one constructs $f = \Phi^{-1}(g)$ by $f(s \otimes t) = \sum_{u\in B(\sigma')} \alpha_{t,u} u$. Because of that, V is monoidal closed.

The objects of **CPM** are the same as **V**, while the arrows in **CPM** are completely positive maps. The latter are defined as follows [Sel04]:

Definition 3.12. Let $F : V_\sigma \to V_{\sigma'}$ be a linear function. We say that F is:

- *positive* if $F(A)$ is positive for all positive $A \in V_\sigma$
- *completely positive* if $id_\tau \otimes F : V_{\tau\otimes\sigma'} \to V_{\tau\otimes\sigma}$ is positive for all signatures τ

After presenting the category **CPM**, let us go to the definition of the meaning function over the syntax. Starting with types:

$$\begin{aligned}
[\![bit]\!] &= (1, 1) \\
[\![A \otimes B]\!] &= [\![A]\!] \otimes [\![B]\!] \\
[\![qbit]\!] &= 2 \\
[\![A \multimap B]\!] &= [\![A]\!] \otimes [\![B]\!]
\end{aligned}$$

Then we have the interpretation of contexts:

$$\begin{aligned}
[\![x_1 : A_1, \ldots, x_n : A_n]\!] &= [\![A_1]\!] \otimes \cdots \otimes [\![A_n]\!] \\
[\![\varnothing]\!] &= 1
\end{aligned}$$

At last, we have typing judgments which are interpreted as linear maps:

$$[\![\Delta \triangleright M : A]\!] = [\![\Delta]\!] \to [\![A]\!]$$

According to the syntax fragment, the set of rules that compose the denotational semantics are presented in Table 7.

$$[\![x : A \triangleright x : A]\!](v) = v \qquad [\![\triangleright * : \top]\!](x) = x$$

$$[\![\triangleright 0 : bit]\!](x) = (x, 0) \qquad [\![\triangleright 1 : bit]\!](x) = (0, x)$$

$$[\![\triangleright new : bit \multimap qbit]\!] = \Phi(\iota) : 1 \to [\![bit]\!] \otimes [\![qbit]\!]$$

$$[\![\triangleright meas : qbit \multimap bit]\!] = \Phi(p) : 1 \to [\![qbit]\!] \otimes [\![bit]\!]$$

$$[\![\triangleright U : qbit^{\otimes n} \multimap qbit^{\otimes n}]\!] = \Phi(U) : 1 \to 2^{2n}$$

$$\frac{[\![\Delta, x : A \triangleright M : B]\!] = f : [\![\Delta]\!] \otimes [\![A]\!] \to [\![B]\!]}{[\![\Delta \triangleright \lambda x.M : A \multimap B]\!] = \Phi(f) : [\![\Delta]\!] \to [\![A]\!] \otimes [\![B]\!]}$$

$$\frac{[\![\Delta \triangleright M : A \multimap B]\!] = \Phi(g) : [\![\Delta]\!] \to [\![A]\!] \otimes [\![B]\!] \qquad [\![\Gamma \triangleright N : A]\!] = f : [\![\Gamma]\!] \to [\![A]\!]}{[\![\Delta, \Gamma \triangleright MN : B]\!] = x \otimes y \mapsto g(x \otimes (fy)) : [\![\Delta]\!] \otimes [\![\Gamma]\!] \to [\![B]\!]}$$

$$\frac{[\![\Delta \triangleright P : bit]\!] : x \mapsto (px, qx) : [\![\Delta]\!] \to [\![bit]\!] \qquad [\![\Gamma \triangleright M : A]\!] = f : [\![\Gamma]\!] \to [\![A]\!] \qquad [\![\Gamma \triangleright N : A]\!] = g : [\![\Gamma]\!] \to [\![A]\!]}{\Delta, \Gamma \triangleright \text{if } P \text{ then } M \text{ else } N : \mathrm{A} : x \otimes y \mapsto (px)(fy) + (qx)(gy)}$$

Table 7: Set of rules of the denotational semantics

where, $\Phi : hom(A \otimes B, C) \to hom(A, B \otimes C)$ is the bijection from the closed structure, ι denotes de creation of qubits and p the measurement operation:

$$\begin{aligned}
\iota : 1, 1 &\to 2 \\
(a, b) &\mapsto \begin{pmatrix} a & 0 \\ 0 & b \end{pmatrix}
\end{aligned}
\qquad\qquad
\begin{aligned}
p : 2 &\to 1, 1 \\
\begin{pmatrix} a & b \\ c & d \end{pmatrix} &\mapsto (a, d)
\end{aligned}$$

The meaning function also acts on quantum closures and on probabilistic distributions of quantum closures. Consider the former first. If $\Delta \vDash [Q, L, M] : A$ is a quantum closure where $L = |x_1 \dots x_n y : 1 \dots y_m\rangle$ and $|Q| = n + m$, then the quantum context (Q, L) can be seen as a map $g : 1 \to 2^{\otimes n} \otimes 2^{\otimes m}$ such that $g(1) = Q$. Furthermore, if we have

$$[\![\Delta, x_1 : qbit, \dots, x_n : qbit \triangleright M : A]\!] = f : [\![\Delta]\!] \otimes 2^{\otimes n} \to [\![A]\!]$$

then, $[\![\Delta \vDash [Q, L, M] : A]\!]$ is defined through the following composition:

$$[\![\Delta]\!] \xrightarrow{id_{[\![\Delta]\!]} \otimes g} [\![\Delta]\!] \otimes 2^{\otimes n} \otimes 2^{\otimes m} \xrightarrow{f \otimes 2^{\otimes m}} [\![A]\!] \otimes 2^{\otimes m} \xrightarrow{[\![A]\!] \otimes Tr} [\![A]\!]$$

where Tr is the trace operation.

The definition of probabilistic distribution of quantum closures is obtained by using linearity on the definition of quantum closures.

$$[\![\Delta \vDash \sum_i \rho_i P_i : A]\!] = \sum_i \rho_i [\![\Delta \vDash P_i : A]\!]$$

The substitution lemma for the denotational semantics is the following:

Lemma 3.13. If $|\Gamma| \cap |\Delta| = \varnothing$ and $[\![\Delta, x : A \triangleright M : B]\!] = G : [\![\Delta]\!] \otimes [\![A]\!] \to [\![B]\!]$, $[\![\Gamma \triangleright N : A]\!] = F : [\![\Gamma]\!] \to [\![A]\!]$, then $[\![\Delta, \Gamma \triangleright M[N/x] : B]\!] = H : [\![\Delta]\!] \otimes [\![\Gamma]\!] \to [\![B]\!]$, with $H(d \otimes g) = G(d \otimes (Fg))$.

The following lemma tells that the three styles of the previous semantics (small-step, big-step, and denotational) agree with each other.

Lemma 3.14. Given a program $P : A$ with $prob' P = \sum_i \rho_i P_i$ then $[\![P : A]\!] = [\![prob' P : A]\!] = [\![prob'_U P : A]\!]$.

3.3.3 Equivalence classes of terms

Since it is possible to build terms, one may want to know if whether two terms are equivalent in some way. For that, one can use an *axiomatic equivalence*, based on a set of syntactic rules, an *operational equivalence*, based on the behaviour of terms, and/or a *denotational equivalence*, based on the meaning function.

Let us begin with the first notion of equivalence. According to the fragment syntax considered before, the axiomatic equivalence is defined as follows:

Definition 3.15. An equivalence relation on a typing judgment, denoted by $\approx_{ax}$, is the smallest relation obeying the following rules:

(β)	$\Gamma \triangleright (\lambda x.M)N \approx_{ax} M[N/X] : A$
(η)	$\Gamma \triangleright \lambda x.Mx \approx_{ax} M : A \multimap B$
(β^1_{if})	$\Gamma \triangleright \text{if } 1 \text{ then } M \text{ else } N \approx_{ax} M : A$
(β^0_{if})	$\Gamma \triangleright \text{if } 0 \text{ then } M \text{ else } N \approx_{ax} N : A$
(η_{if})	$\Gamma \triangleright \text{if } B \text{ then } M[1/x] \text{ else } M[0/x] \approx_{ax} M[B/x] : A$
(id)	$\Gamma \triangleright meas(new\ M) \approx_{ax} M : qbit$

Table 8: Fragment of the rules of axiomatic equivalence

The operational equivalence depends on the behaviour of terms, *i.e.* the set of transitions made from the beginning of the term until its value, if existent. Intuitively, two terms are operationally equivalent if their behaviour is the same; to say that, however, one needs to observe the behaviour of terms. However, $\top$ and *bit* are the only types observable. To verify the behaviour of a term, it is used a so-called *operational context*. Informally, an operational context is a term $C[-]$, where $[-]$ is known as *hole*, in which two other terms M and M' can be inserted, *i.e.* $C[M]$ and $C[M']$. Now, consider that $C[-] : bit$ and let M and M' be two other terms. For $C[M]$ and $C[M']$ to have the same behaviour they must reduce to 0 and 1 with the same probability. Formally:

Definition 3.16. An operational context is a formula defined by:

$$C[-] ::= [-] \mid (C[-]M) \mid (MC[-]) \mid \lambda x.C[-] \mid \text{if } C[-] \text{ then } M \text{ else } N \mid \text{if } M \text{ then } C[-] \text{ else } C[-]$$

With the introduction of this concept, emerges the notion of *captured variable*, *i.e.* a variable that is inside the hole. Attention on the definition of typed operational contexts for a better comprehension of captured variables.

Definition 3.17. A *typed operational context* is a typing tree with root $\Gamma' \triangleright C[-] : B$, considering the additional axiom $\Gamma \triangleright [-] : A$, *i.e.* a typing tree of the form

$$\dfrac{\dfrac{\Gamma \triangleright [-] : A}{\vdots}}{\Gamma' \triangleright C[-] : B}$$

The interpretation of the typing tree is as follows: the root has type B, with free variables Γ', a hole of type A, and captured variables Γ. Another notation used is $\Gamma' \triangleright C[\Gamma \triangleright - : A] : B$.

The definition of substitution on operational contexts is only going to consider the type *bit*.

Definition 3.18. Let $\triangleright C[\Gamma \triangleright - : A] : bit$ be a closed typed operational context of type *bit*, and let $R = [Q, L, M]$ be a well-typed quantum closure with typing judgement $\Gamma \vDash [Q, L, M] : A$. In this case, the substitution $C[R]$ is defined by $[Q, L, C[M]]$, where M is syntactically replacing $[-]$ in $C[-]$. This definition is linearly extended to probabilistic distributions of quantum closures of the form $\Gamma \vDash \sum_i \rho_i R_i : A$ by setting $C[\sum_i \rho_i R_i] = \sum_i \rho_i C[R_i]$.

At last, the definition of operational equivalence is as follows:

Definition 3.19. Given two well-typed quantum closures $\Gamma \triangleright R : A$ and $\Gamma \triangleright R' : A$, it is said that R is operationally equivalent to R' with respect to Γ if for all closed typed operational contexts $\triangleright C[\Gamma \triangleright - : A] : bit$, $C[R] \Downarrow 0 = C[R'] \Downarrow 0$ and $C[R] \Downarrow 1 = C[R'] \Downarrow 1$. In this case, it is written $\Gamma \triangleright R \approx_{op} R' : A$. If M, M' are terms, it is said that $\Gamma \triangleright M \approx_{op} M' : A$ if $\Gamma \vDash [\|\rangle, |\rangle, M] \approx_{op} [|\rangle, |\rangle, M'] : A$.

Finally, denotational equivalence. Informally, two terms are denotationally equivalent if their meaning function are the same map. Formally:

Definition 3.20. It is said that two typing judgments $\Gamma \triangleright M : A$ and $\Gamma \triangleright M' : A$ are denotationally equivalent if $[\![\Gamma \triangleright M : A]\!]$ an $[\![\Gamma \triangleright M' : A]\!]$ are the same map in **CPM**. In that case it is written $\Gamma \triangleright M \approx_{den} M' : A$.

This definition can be extended to quantum closures as follows:

$$\Gamma \vDash R \approx_{den} R' : A \text{ is true if } [\![\Gamma \vDash R : A]\!] = [\![\Gamma \vDash R' : A]\!]$$

Regarding the soundness and full abstraction of the quantum λ-calculus, we will limit ourselves to show the respective theorems and make a remark on the latter. For the soundness of the axiomatic equivalence the theorem is:

Theorem 3.21. If $\Gamma \vDash M \approx_{ax} M' : A$ then $\Gamma \vDash M \approx_{op} M' : A$.

For the full abstraction of the denotational semantics the theorem is:

Theorem 3.22. The denotational semantics is fully abstract with respect to the operational equivalence of typing judgements, *i.e.*

$$[\![\Gamma \triangleright M : A]\!] = [\![\Gamma \triangleright M' : A]\!] \text{ if and only if } \Gamma \triangleright M \approx_{op} M' : A.$$

In the fragment of the syntax presented there is not a non-terminating term Ω, which is necessary for full abstraction to hold. We decided to not include it, because in our context it is only necessary in this last result.

4 Developed work

In this subsection we present the work that was developed throughout the beginning of this research until now.

4.1 Introduction

We study an extension of λ-calculus with *wait* construct. Consider an infinite set of variables $\mathcal{V}$. Terms of the language are then constructed via the following grammar:

$$M, N ::= x \mid MN \mid \lambda x.M \mid wait_n(M)$$

where $x \in \mathcal{V}$ and n is a natural number, with 0 included. We slightly abuse nomenclature by calling these terms 'λ-terms'. As usual, we call *values* terms that are built via the grammar:

$$V ::= x \mid \lambda x.M$$

To have a friendlier view of λ-terms, parentheses are sometimes omitted. The convention to restore the parentheses is known as *association to the left* is applied as follows: $MNPQ \equiv (((MN)P)Q$.

Definition 4.1. We define α-equivalence as the smallest congruence relation $=$ on λ-terms such that for all terms M and all variables y that do not occur in M,

$$\lambda x.M = \lambda y.(M\{y/x\}).$$

The term $M\{y/x\}$ denotes the replacement of x by y in M. More explicitly, it is the smallest relation that satisfies the following rules:

$$(refl)\ \frac{}{M = M} \qquad (symm)\ \frac{M = N}{N = M} \qquad (trans)\ \frac{M = N \quad N = P}{M = P}$$

$$(app)\ \frac{M = M' \quad N = N'}{MN = M'N'} \qquad (lam)\ \frac{M = M'}{\lambda x.M = \lambda x.M'} \qquad (wait)\ \frac{M = M'}{wait_n(M) = wait_n(M')}$$

$$(\alpha)\ \frac{y \notin M}{\lambda x.M = \lambda y.(M\{y/x\})}$$

Now, let us define the substitution of variables that occur free in a term.

Definition 4.2. Let N and M be λ-terms and x a variable. The substitution of N for free occurrences of x in M (denoted by $M[N/x]$), is defined as follows:

1. $x[N/x] \equiv N$
2. $y[N/x] \equiv y$, if $x \notin y$
3. $(MP)[N/x] \equiv (M[N/x])(P[N/x])$
4. $(\lambda x.M)[N/x] \equiv \lambda x.M$
5. $(\lambda y.M)[N/x] \equiv \lambda y.(M[N/x])$, if $x \notin y$ and $y \notin FV(N)$
6. $(\lambda y.M)[N/x] \equiv \lambda y'.(M\{y'/y\}[N/x])$, if $x \notin y$, $y \in FV(N)$, and y' fresh
7. $wait_n(M)[N/x] \equiv wait_n(M[N/x])$

Let us now define a small-step operational semantics for the language regarding terms without free variables. We adopt the *call-by-value* strategy, because it is usually more natural in the presence of effects (in our setting, the effect at hand is time). We write $M \xrightarrow{n} M'$ to denote that M reduces to M' in a single step that takes n units of time. Then,

Definition 4.3. The single-step β-reduction is the smallest relation $\xrightarrow{n}$ on terms satisfying the rules,

$$(APP1)\ \frac{N \xrightarrow{n} N'}{MN \xrightarrow{n} MN'} \qquad (APP2)\ \frac{M \xrightarrow{n} M'}{MV \xrightarrow{n} M'V}$$

$$(RED)\ \frac{}{(\lambda x.M)V \xrightarrow{0} M[V/x]} \qquad (WAIT)\ \frac{}{wait_n(M) \xrightarrow{n} M}$$

Theorem 4.4 (Deterministic reduction)**.** Let M, M', and M'' be λ-terms. If $M \xrightarrow{n} M'$ and $M \xrightarrow{m} M''$ then $M' = M''$ and $n = m$, modulo α-conversion.

Proof. The proof follows straightforwardly by noticing that the premisses of the rules are mutually disjoint, *i.e.* at most one rule is applicable for reducing M: first, notice that a λ-term M can be reduced *only* if it is an application $M'N$ or a term of type $wait_n(M')$. Clearly, if M is of the latter type then there is only one applicable rule. So we can focus just on the case in which $M = M'N$. For this case there are three applicable rules, but if M' and N are values then we can only use (RED) and if M' is not a value then we can only use the other two. So, to finish the proof, we assume that M' is not a value. For this case, as already mentioned, there are two applicable rules, but only one can be used if N is a value and only the other can be used if N is not a value. □

From the relation $\xrightarrow{n}$ we now build a 'multiple-step' reduction relation $\xrightarrow{n}\!\!\!\!\rightarrow$ by induction.

Definition 4.5. Define $\xrightarrow{n}\!\!\!\!\rightarrow$ as the smallest relation over λ-terms such that,

- $M \xrightarrow{0}\!\!\!\!\rightarrow M$;
- if $M \xrightarrow{n} M'$ and $M' \xrightarrow{m}\!\!\!\!\rightarrow M''$ then $M \xrightarrow{n+m}\!\!\!\!\rightarrow M''$.

The above definition can also be interpreted as follows.

Definition 4.6. The relation $\xrightarrow{n}\!\!\!\!\rightarrow$ can be built by the following rules:

$$\frac{}{M \xrightarrow{0}\!\!\!\!\rightarrow M}\ (1) \qquad \frac{M \xrightarrow{n} M' \quad M' \xrightarrow{m}\!\!\!\!\rightarrow M''}{M \xrightarrow{n+m}\!\!\!\!\rightarrow M''}\ (2)$$

Theorem 4.7 (Deterministic reduction II)**.** Let M, M', and M'' be λ-terms. If $M \xrightarrow{n}\!\!\!\!\rightarrow M'$ and $M \xrightarrow{m}\!\!\!\!\rightarrow M''$ then $M' = M''$ and $n = m$, modulo α-conversion.

Proof. The proof will be done by structural induction over $\xrightarrow{}\!\!\!\!\rightarrow$.

1. If the size of $\xrightarrow{}\!\!\!\!\rightarrow$ is zero or one then we are done by Theorem 4.4;
2. If the size of $\xrightarrow{}\!\!\!\!\rightarrow$ is greater than one then we proceed as follows: by assumption $M \xrightarrow{n}\!\!\!\!\rightarrow M'$ and $M \xrightarrow{m}\!\!\!\!\rightarrow M''$, meaning that $M \xrightarrow{n_1} M_1$ and $M_1 \xrightarrow{n_2}\!\!\!\!\rightarrow M'$ where $n = n_1 + n_2$, and $M \xrightarrow{m_1} M_2$ and $M_2 \xrightarrow{m_2}\!\!\!\!\rightarrow M''$ where $m = m_1 + m_2$. According to Theorem 4.4, $M_1 = M_2$ and $n_1 = m_1$. By the induction hypothesis, we conclude that $M' = M''$ and $n_2 = m_2$. We now apply the definition of $\xrightarrow{}\!\!\!\!\rightarrow$ and since $n_2 = m_2$ and $n_1 = m_1$ we also conclude that $n = m$.

□

4.2 Models for linear and non-linear λ-calculus with delays

In this subsection we begin by presenting a variant of typed λ-calculus. From it, we developed a linear and a non-linear model first without considering delays and later by considering it.

4.2.1 Typed λ-calculus

Let us start by presenting the typed λ-calculus, which is going to be the basis for developing the denotational semantics.

Similarly to the previous section, consider an infinite set of variables $\mathcal{V}$. Terms of the the language are constructed via the following grammar:

$$M, N ::= x \mid MN \mid \lambda x.M \mid wait_n(M) \mid \langle M, N\rangle \mid x \leftarrow M; N \mid *$$

Comparatively to the syntax of Subsection 4.1, we added pairing, sequential, and unit, respectively. The sequential rules reads as follows: execute M, bind x to the value it returns, then execute N.

In its turn, the values are:

$$V, U ::= x \mid \lambda x.M \mid \langle V, U\rangle \mid *$$

Since we are in the beginning, the types will not afford any notion of time. Therefore, we adopt a propositional type system:

$$\sigma, \tau ::= a \mid \sigma \to \tau \mid \sigma \times \tau \mid 1$$

where a is a base/atom type and 1 is a 'unit' type. From this, one can infer that $*$ has only type 1.

We use typing judgments of the form

$$\Gamma \vdash M : \sigma$$

to relate the set of free variables (Γ) and the typed λ-term ($M : \sigma$). Furthermore, if M has no free variables and it is well-typed then it is called program.

With timing constraints we are introducing the notion of effects on the λ-calculus. As a consequence, terms can now be:

- *producers*, *i.e.* terms that cause effects
- *values*, *i.e.* terms that do not cause effects

For that reason we divided typing judgements in two:

$$\begin{array}{rl} \Gamma \vdash^w M : \sigma & M \text{ is a producer of type } \sigma \\ \Gamma \vdash^v V : \sigma & M \text{ is a value of type } \sigma \end{array}$$

For simplicity, we shall use $\Gamma \vdash M : \sigma$ in theorems to denote both.

The term formation rules are presented on Table 9.

$$\begin{array}{lll|ll} (\text{AX1}) & \Gamma, x : a \vdash^v x : a & & (*) & \Gamma \vdash^v * : 1 \\ (\to E) & \dfrac{\Gamma \vdash^v M : \sigma \to \tau \quad \Gamma \vdash^v N : \sigma}{\Gamma \vdash^w MN : \tau} & & (\to I) & \dfrac{\Gamma, x : \sigma \vdash^w M : \tau}{\Gamma \vdash^v (\lambda x.M) : (\sigma \to \tau)} \\ (\to W) & \dfrac{\Gamma \vdash^w M : \tau}{\Gamma \vdash^w wait_n(M) : \tau} & & (P) & \dfrac{\Gamma \vdash^v V : \sigma \qquad \Gamma \vdash^v U : \tau}{\Gamma \vdash^v \langle V, U \rangle : \sigma \times \tau} \\ (S) & \dfrac{\Gamma \vdash^w M : \sigma \qquad \Gamma, x : \sigma \vdash^w N : \tau}{\Gamma \vdash^w x \leftarrow M; N : \tau} & & & \end{array}$$

Table 9: Term formation rules

With the addition of more terms, the definition of α-equivalence must be updated.

Definition 4.8. We define α-equivalence as the smallest congruence relation $=_\alpha$ on λ-terms such that for all terms M and all variables y that do not occur in M,

$$\lambda x.M =_\alpha \lambda y.(M\{y/x\}).$$

The term $M\{y/x\}$ denotes the replacement of x by y in M. More explicitly, it is the smallest relation that satisfies the rules on Table 10.

$$\begin{array}{lll|lll} (refl) & \dfrac{}{M =_\alpha M} & & (symm) & \dfrac{M =_\alpha N}{N =_\alpha M} & & (trans) & \dfrac{M =_\alpha N \quad N =_\alpha P}{M =_\alpha P} \\ (app) & \dfrac{M =_\alpha M' \quad N =_\alpha N'}{MN =_\alpha M'N'} & & (lam) & \dfrac{M =_\alpha M'}{\lambda x.M =_\alpha \lambda x.M'} & & (wait) & \dfrac{M =_\alpha M'}{wait_n(M) =_\alpha wait_n(M')} \\ (seq) & \dfrac{M =_\alpha M'}{M \text{ to } x.N =_\alpha M' \text{ to } x.N} & & (pair) & \dfrac{V =_\alpha V' \qquad U =_\alpha U'}{\langle V, U \rangle =_\alpha \langle V', U' \rangle} & & & \\ (\alpha) & \dfrac{y \notin M}{\lambda x.M =_\alpha \lambda y.(M\{y/x\})} & & (lam1) & \dfrac{}{(\lambda x.M)V =_\alpha M[V/x]} & & (seq1) & \dfrac{}{x \leftarrow V; N =_\alpha N[V/x]} \end{array}$$

Table 10: Axioms for α-equivalence

In the presence of timing constraints, we can define a notion of w-equivalence, denoted as $=_w$, which is the congruence closure of α-equivalence. The axioms that we need to add in Table 10 are presented in Table 11.

We also need to update the definition of substitution (Definition 4.9):

$$(wait1) \quad \overline{wait_0(M) = M} \;\Big|\; (wait2) \quad \overline{wait_n(wait_m(M)) = wait_{n+m}(M)}$$

Table 11: Remaining axioms for w-equivalence

Definition 4.9. Let N and M be terms and x a variable. The substitution of N for free occurrences of x in M, is defined as follows:

1. $x[N/x] \equiv N$
2. $y[N/x] \equiv y$, if $x \neq y$
3. $(MP)[N/x] \equiv (M[N/x])(P[N/x])$
4. $(\lambda x.M)[N/x] \equiv \lambda x.M$
5. $(\lambda y.M)[N/x] \equiv \lambda y.(M[N/x])$, if $x \neq y$ and $y \notin FV(N)$
6. $(\lambda y.M)[N/x] \equiv \lambda y'.(M\{y'/y\}[N/x])$, if $x \neq y$, $y \in FV(N)$, and y' fresh
7. $wait_n(M)[N/x] \equiv wait_n(M[N/x])$
8. $\langle M, P\rangle[N/x] \equiv \langle M[N/x], P[N/x]\rangle$
9. $(x \leftarrow M; P)[N/x] \equiv x \leftarrow M[N/x]; P$
10. $(y \leftarrow M; P)[N/x] \equiv y \leftarrow M[N/x]; P[N/x]$
11. $*[N/x] = *$

Now we are prepared to present the semantics. Let us begin with the operational semantics, which is made under terms without free variables. A formula of the form $M \to^n M'$ reads as: M reduces to M' after n units of time. Note that this interpretation is valid for small-step and big-step operational semantics. We start by presenting the rules of the small-step operational semantics.

$$\begin{array}{lc|lc}
(\text{WAIT}) & \overline{wait_n(M) \xrightarrow{n} M} & (\text{AX1}) & \overline{(\lambda x.M)V \xrightarrow{0} M[V/x]} \\
(\text{APP1}) & \dfrac{N \xrightarrow{n} N'}{MN \xrightarrow{n} MN'} & (\text{APP2}) & \dfrac{M \xrightarrow{n} M'}{MV \xrightarrow{n} M'V} \\
(\text{PAIR1}) & \dfrac{U \xrightarrow{0} U'}{\langle V, U\rangle \xrightarrow{0} \langle V, U'\rangle} & (\text{PAIR2}) & \dfrac{V \xrightarrow{0} V'}{\langle V, u\rangle \xrightarrow{0} \langle V', U'\rangle} \\
(\text{SEQ1}) & \dfrac{M \xrightarrow{n} M'}{x \leftarrow M; N \xrightarrow{n} x \leftarrow M'; N} & (\text{SEQ2}) & \dfrac{}{x \leftarrow V; N \xrightarrow{0} N[V/x]}
\end{array}$$

Table 12: Small-step operational semantics

β-equivalence is defined as being the reflexive symmetric transitive closure of $\xrightarrow{n}$.

From this small-step definition, we define a 'multi-step' notion $\xrightarrow{n}\!\!\!\!\rightarrow$ on Definition 4.10.

Definition 4.10. The relation $\xrightarrow{n}\!\!\!\!\rightarrow$ can be built by the following rules:

$$\frac{}{M \xrightarrow{0}\!\!\!\!\rightarrow M} \; (1) \qquad \frac{M \xrightarrow{n} M' \quad M' \xrightarrow{m}\!\!\!\!\rightarrow M''}{M \xrightarrow{n+m}\!\!\!\!\rightarrow M''} \; (2)$$

Note that in Table 12 there is no rule for x and $*$. This is due the formulation of $\xrightarrow{n}\!\!\!\!\rightarrow$, because if we allowed, for example, $x \xrightarrow{0} x$, then we would have an infinite sequence of steps, which is not desirable.

With the introduction of a big-step operational semantics we aim to represent the relation between terms and values. The rules that compose the big-step operational semantics are on Table 13.

An expected result is that $\Downarrow^n$ and $\xrightarrow{n}\!\!\!\!\rightarrow$ are equivalent, expressed in Theorem 4.11.

Theorem 4.11. Let M be a λ-term and V, V' values. Then $M \Downarrow^n V$ iff $M \xrightarrow{m}\!\!\!\!\rightarrow V'$, where $V \equiv V'$ and $n = m$.

$$\text{(AX1)}\quad \frac{}{x \Downarrow^0 x} \qquad \text{(AX2)}\quad \frac{}{\lambda x.M \Downarrow^0 \lambda x.M}$$

$$\text{(APP)}\quad \frac{M \Downarrow^n \lambda x.M' \quad N \Downarrow^m U \quad M'[U/x] \Downarrow^p V}{MN \Downarrow^{n+m+p} V} \qquad \text{(WAIT)}\quad \frac{M \Downarrow^m V}{wait_n(M) \Downarrow^{n+m} V}$$

$$\text{(SEQ)}\quad \frac{M \Downarrow^n V \quad N[V/x] \Downarrow^m U}{x \leftarrow M; N \Downarrow^{n+m} U} \qquad \text{(PAIR)}\quad \frac{V \Downarrow^0 V' \quad U \Downarrow^0 U'}{\langle V, U\rangle \Downarrow^0 \langle V', U'\rangle}$$

$$(*)\quad \frac{}{* \Downarrow^0 *}$$

Table 13: Big-step operational semantics

Proof. Let us begin with the left-to-right implication. The proof is made under induction over $\Downarrow$, and cases are distinguished based on the form of M. Here we present the cases of application and sequential, once the remaining are straightforward.

1. Case $M \equiv M'N$. Then it is assumed that $M'N \Downarrow^n V$, which entails $M' \Downarrow^{n_1} \lambda x.P$, $N \Downarrow^{n_2} U$, and $P[U/x] \Downarrow^{n_3} V$. By i.h. $M' \xrightarrow{m_1}\!\!\!\!\rightarrow \lambda x.P'$, $N \xrightarrow{m_2}\!\!\!\!\rightarrow U'$, and $P'[U'/x] \xrightarrow{m_3}\!\!\!\!\rightarrow V'$. Through several applications of the property of transitivity on the definition of $\longrightarrow\!\!\!\!\rightarrow$ and $(APP1)$ we have $M'N \xrightarrow{m_2}\!\!\!\!\rightarrow M'U'$. Through several applications of the property of transitivity on the definition of $\longrightarrow\!\!\!\!\rightarrow$ and $(APP2)$ we have $M'U' \xrightarrow{m_1}\!\!\!\!\rightarrow (\lambda x.P')U'$. By an application of (RED) we have $(\lambda x.P')U' \xrightarrow{0}\!\!\!\!\rightarrow P'[U'/x]$. Through the i.h. we have $P'[U'/x] \xrightarrow{m_3}\!\!\!\!\rightarrow V'$. Henceforth $M'N \xrightarrow{m_1+m_2+m_3}\!\!\!\!\rightarrow V'$, where $V \equiv V'$ and $n = n_1+n_2+n_3 = m_1+m_2+m_3 = m$.

2. Case $M \equiv x \leftarrow P; Q$. Then we have $x \leftarrow P; Q \Downarrow^n U$ that entails $P \Downarrow^{n_1} V$ and $Q[V/x] \Downarrow^{n_2} U$. By i.h. $P \xrightarrow{m_1}\!\!\!\!\rightarrow V$ and $Q[V/x] \xrightarrow{m_2}\!\!\!\!\rightarrow U$. Through several applications of the property of transitivity on the definition of $\longrightarrow\!\!\!\!\rightarrow$ and (SEQ1) we have $x \leftarrow P; Q \xrightarrow{m_1}\!\!\!\!\rightarrow x \leftarrow V; Q$. By an application of (SEQ2) we have $x \leftarrow V; Q \xrightarrow{0}\!\!\!\!\rightarrow Q[V/x]$, which by the i.h. $Q[U/x] \xrightarrow{m_2}\!\!\!\!\rightarrow U$. Furthermore, $n = n_1 + n_2 = m_1 + m_2 = m$.

Now, let us proof the right-to-left implication. The proof is now made under induction over $\longrightarrow\!\!\!\!\rightarrow$, and cases are still distinguished based on the form of M. We will present only the cases of application and sequential, for the previous reasons.

1. Case $M \equiv M'N$. Then the only possible transition is $M'N \xrightarrow{m}\!\!\!\!\rightarrow V'$. Since MN' reduces to a value, then M' must reduce to an abstraction and N to a value, *i.e.* $M' \xrightarrow{m_1}\!\!\!\!\rightarrow \lambda x.P'$ and $N \xrightarrow{m_2}\!\!\!\!\rightarrow V''$. We now have $(\lambda x.P')V''$, which by an application of the rule (RED) gives us $P'[V''/x]$, *i.e.* $(\lambda x.P')V'' \xrightarrow{0}\!\!\!\!\rightarrow P'[V''/x]$. $P'[V''/x]$ is a value, so $P'[V''/x] \xrightarrow{m_3}\!\!\!\!\rightarrow V'$. By i.h. $N \Downarrow^{n_1} V_1$, $M' \Downarrow^{n_2} V_2 \equiv \lambda x.P'$, and $P'[U/x] \Downarrow^{n_3} V_3$. By applying the rule $(APP-RED)$ we have $M'N \Downarrow^{n_1+n_2+n_3} V_3$. Therefore $V \equiv V_3 \equiv V'$ and $n = n_1+n_2+n_3 = m_1 + m_2 = m$.

2. Case $M \equiv x \leftarrow P; Q$. Then, by applying several times the property of reflexivity on the definition of $\longrightarrow\!\!\!\!\rightarrow$ and the rule (SEQ1) we have $x \leftarrow P; Q \xrightarrow{m_1}\!\!\!\!\rightarrow x \leftarrow V; Q$ that entails $P \xrightarrow{m_1}\!\!\!\!\rightarrow V$. By an application of (SEQ2) we have $x \leftarrow V; Q \xrightarrow{0}\!\!\!\!\rightarrow Q[V/x]$. Independently of the form of Q we will have $Q[V/x] \xrightarrow{m_2}\!\!\!\!\rightarrow U$. By i.h. $P \Downarrow^{n_1} V$ and $Q[V/x] \Downarrow^{n_2} U$. By an application of (SEQ) we have $x \leftarrow P; Q \Downarrow^n U$. Furthermore, $m = m_1 + m_2 = n_1 + n_2 = n$.

□

To prove that $\Downarrow$ is sound to $=_w$, *i.e.* that $=_w$ and $\Downarrow$ are equivalent, we need a substitution lemma for the big-step operational semantics.

Lemma 4.12 (Substitution)**.** If $M \Downarrow^n V$ and $N \Downarrow^m U$, then $M[N/y] \Downarrow^{n+m} V[U/y]$. Furthermore $V[U/y] \Downarrow^p X$, where X is a value.

Proof. The proof is made under induction on the derivation of $M \Downarrow^n V$. The cases are distinguished by the last rule used in the derivation. □

Now, we are ready to state and proof an important result.

Theorem 4.13. If $M =_w N$ then $M \Downarrow^n V$ and $N \Downarrow^m V$. [1]

Proof. The proof is made by induction on $=$ rules, *i.e.* over the axioms. The cases are distinguished by the different axioms. □

[1] FiXme Note: Depois eu mostro-te como provar isto automaticamente usando mónadas.

4.2.2 Classical λ-calculus

In the classical λ-calculus, variables can be copied or discorded. In terms of categories, this means that we can use projection maps and pair maps via 'product constructions'. Having this in consideration, we opted to interpret the λ-calculus through the category of **Set**, which is Cartesian closed.

In Subsection 4.2.1, a typed λ-calculus with timing constraints was presented. However, in this subsection we are not going to take into consideration effects, so the term $wait_n(M)$ is going to be forgotten. This implies that the operational semantics loses its temporal notion and transitions are now simply denoted by $\to$, $\twoheadrightarrow$, and $\Downarrow$. In addition, it also means that typing judgments are all of the type value. To simplify, we write $\vdash$ instead of $\vdash^v$.

When developing a denotational semantics, we need to define a *meaning function* that maps syntactic elements into semantics elements. In our case, the meaning function will map **types** to **sets** and **typing judgments** to **functions**.

The majority of functions that appear in denotational semantics rules are all explained on Appendix A. However, the function that appears in the rule (VOID) is not. Thus, its definition is as follows:

$$\begin{aligned} f : C &\to \{*\} \\ c &\mapsto * \end{aligned}$$

This function maps every object of C into the unit. In terms of sets, we can say that any set is send to the unique-element set, known as *singleton*.

The set of rules for the denotational semantics are on Table 14.

$$\textbf{(AX)} \quad [\![\Gamma, x : \sigma \vdash x : \sigma]\!] = \pi_x : [\![\Gamma]\!] \times [\![\sigma]\!] \to [\![\sigma]\!]$$

$$\textbf{(APP)} \quad \frac{[\![\Gamma \vdash M : \sigma \to \tau]\!] = f : [\![\Gamma]\!] \to [\![\tau]\!]^{[\![\sigma]\!]} \qquad [\![\Gamma \vdash N : \sigma]\!] = g : [\![\Gamma]\!] \to [\![\sigma]\!]}{[\![\Gamma \vdash MN : \tau]\!] = app \circ \langle f, g \rangle : [\![\Gamma]\!] \to [\![\tau]\!]}$$

$$\textbf{(ABS)} \quad \frac{[\![\Gamma, x : \sigma \vdash M : \tau]\!] = f : [\![\Gamma]\!] \times [\![\sigma]\!] \to [\![\tau]\!]}{[\![\Gamma \vdash \lambda x.M : \sigma \to \tau]\!] = \lambda f : [\![\Gamma]\!] \to [\![\tau]\!]^{[\![\sigma]\!]}}$$

$$\textbf{(SEQ)} \quad \frac{[\![\Gamma \vdash M : \sigma]\!] = f : [\![\Gamma]\!] \to [\![\sigma]\!] \qquad [\![\Gamma, x : \sigma \vdash N : \tau]\!] = g : [\![\Gamma]\!] \times [\![\sigma]\!] \to [\![\tau]\!]}{[\![\Gamma \vdash x \leftarrow M; N : \tau]\!] = g \circ \langle id, f \rangle : [\![\Gamma]\!] \to [\![\tau]\!]}$$

$$\textbf{(PAIR)} \quad \frac{[\![\Gamma \vdash V : \sigma]\!] = f : [\![\Gamma]\!] \to [\![\sigma]\!] \qquad [\![\Gamma \vdash U : \tau]\!] = g : [\![\Gamma]\!] \to [\![\tau]\!]}{[\![\Gamma \vdash \langle V, U \rangle : \sigma \times \tau]\!] = \langle f, g \rangle : [\![\Gamma]\!] \to [\![\sigma]\!] \times [\![\tau]\!]}$$

$$\textbf{(UNIT)} \quad \frac{}{[\![\Gamma \vdash * : 1]\!] = f : [\![\Gamma]\!] \to \{*\}}$$

Table 14: Denotational semantics for classical λ-calculus

Notice the rule **(SEQ)**. According to the intuitively description of this rule, the variable x is going to be used in N. So, the presence of x in the environment of the premise $[\![\Gamma, x : \sigma \vdash N : \tau]\!] = g : [\![\Gamma]\!] \times [\![\sigma]\!] \to [\![\tau]\!]$ makes sense.

When developing different semantics, such like denotational and operational, it is expected that they are, in some way, equivalent. This equivalence is called *soundness*. To prove it we need the following lemmas: exchange, weakening, and substitution .

Lemma 4.14 (Exchange)**.** Consider a judgment of the form $\Gamma, x : \sigma, y : \tau, \Delta \vdash M : \gamma$. The the following conditions hold:

- $\Gamma, y : \tau, x : \sigma, \Delta \vdash M : \gamma$
- $[\![\Gamma, x : \sigma, y : \tau, \Delta \vdash M : \gamma]\!] = [\![\Gamma, y : \tau, x : \sigma, \Delta \vdash M : \gamma]\!] \circ \phi$

Proof. The proof is made using induction over $[\![\Gamma, x : \sigma, y : \tau, \Delta \vdash M : \gamma]\!]$ rules. The cases are distinguished by the form of M. □

Lemma 4.15 (Weakening)**.** Consider a judgment of the form $\Gamma \vdash M : \sigma$ and a variable $x : \gamma \notin \Gamma$. Then the following conditions hold:

- $\Gamma, x : \gamma \vdash M : \sigma$
- $[\![\Gamma, x : \gamma \vdash M : \sigma]\!] = [\![\Gamma \vdash M : \sigma]\!] \circ \pi_1$

Proof. The proof is made under induction over $[\![\Gamma, x : \gamma \vdash M : \sigma]\!]$. The cases are distinguished by the form of M. □

Lemma 4.16 (Substitution)**.** If $\Gamma, x : \sigma \vdash M : \tau$ and $\Gamma \vdash N : \sigma$ then the following conditions hold:

- $\Gamma \vdash M[N/x] : \tau$
- $[\![\Gamma \vdash M[N/x] : \tau]\!] = [\![\Gamma, x : \sigma \vdash M : \tau]\!] \circ \langle id, [\![\Gamma \vdash N : \sigma]\!] \rangle$

Proof. The proof is made under induction on the derivation of $[\![\Gamma, x : \gamma \vdash M : \tau]\!]$. Cases are distinguished by the last rule used in the derivation. □

The soundness theorem is formulated as follows:

Theorem 4.17 (Soundness)**.** Let be V a value and M a term. Then the following condition hold:

$$\text{for all } \vdash M : \sigma \text{ if } M \Downarrow V \text{ then, } \vdash V : \sigma \text{ and } [\![\vdash M : \sigma]\!] = [\![\vdash V : \sigma]\!]$$

Proof. The proof is made under induction over $\Downarrow$ rules and cases are distinguished by the form of M. □

4.2.3 Linear λ-calculus

The key difference between linear λ-calculus and classical λ-calculus is that in the linear setting variables cannot be discarded nor copied. Categorically, this means that we cannot use projection maps nor can we pair maps via 'product constructions'. These restrictions lead to a shift from Cartesian closed categories to monoidal-closed ones. An important example of the latter type of category is the category of *finite-dimensional vector spaces* and linear maps, which is particularly relevant for quantum computation. We present in Figure 1 the term formation rules of linear λ-calculus. Note that quantum λ-calculus is a particular case of linear λ-calculus [SV08].

$$\frac{}{x : \sigma \vdash x : \sigma}\ (\text{var}) \qquad \frac{}{\vdash * : I}\ (\text{unit})$$

$$\frac{\Gamma \vdash V : \sigma \qquad \Delta \vdash U : \tau}{\Gamma, \Delta \vdash V \otimes U : \sigma \otimes \tau}\ (\text{tensor}) \qquad \frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash \lambda x : \sigma.\, M : \sigma \multimap \tau}\ (\text{abs})$$

$$\frac{\Gamma \vdash V : \sigma \multimap \tau \quad \Delta \vdash U : \sigma}{\Gamma, \Delta \vdash V\, U : \tau}\ (\text{app}) \qquad \frac{\Gamma \vdash M : \sigma \qquad \Delta, x : \sigma \vdash M : \tau}{\Gamma, \Delta \vdash x \leftarrow M; N : \tau}\ (\text{seq})$$

Figure 1: Term formation rules.

As one can see from Figure 1 and previously stated, linear logic restricts the use of variables. From Definition 4.9 we change rule 8, and erase rule 11. Regarding rule 8, instead of having

$$\langle P, Q \rangle [N/x] \equiv \langle P[N/x], Q[N/x] \rangle$$

we now have

$$(M \otimes P)[N/x] \equiv (M[N/x]) \otimes (P[N/x])$$

We do not distinguish if x is in M or P due to linearity, *i.e.* from linear logic we know that x can only be in M or in P. For example, if x is in M, then the substitution will only occur in M, while P remains the same. Note that this situation also occurs on rules 3 and 10 of Definition 4.9.

We are now capable to formulate the substitution lemma for the typing judgments. However, to prove it, we need an exchange lemma:

Lemma 4.18 (Exchange 1)**.** Consider a judgment of the form $\Gamma, x : \sigma, y : \tau, \Delta \vdash M : \gamma$. Then the following property hold:

$$\Gamma, y : \tau, x : \sigma, \Delta \vdash M : \gamma$$

Proof. The proof is made on induction over $\Gamma, x : \sigma, y : \tau, \Delta \vdash M : \gamma$. Cases are distinguished by the form of M. □

Lemma 4.19 (Substitution 1)**.** Consider the typing judgments $\Gamma, x : \gamma \vdash M : \sigma$ and $\Omega \vdash N : \gamma$. Then:

$$\frac{\Gamma, x : \gamma \vdash M : \sigma \quad \Omega \vdash N : \gamma}{\Gamma, \Omega \vdash M[N/x] : \sigma}$$

Proof. The proof is made by induction over $\Gamma, x : \gamma \vdash M : \sigma$ derivation. The cases are distinguished by the last rule used on the derivation. □

In the linear setting, we are going to use the category of finite-dimensional vector spaces over a field k, denoted $\mathbf{Vect_k}$, which is symmetric monoidal closed. $\mathbf{Vect_k}$ is defined in Appendix A and the definition of a symmetric monoidal closed category is given in Appendix B.

The operational semantics of the linear λ-calculus is analogous to that of classical λ-calculus, being the only difference the pair rule. Recall that the operational semantics is made from terms without free variables. Since we have to use tensors instead of products, the rule (PAIR) in Table 13 is now:

$$\textbf{(PAIR)} \quad \frac{V \Downarrow V' \quad U \Downarrow U'}{V \otimes U \Downarrow V' \otimes U'}$$

The denotational semantics of the linear λ-calculus, it is also analogous to that of classical λ-calculus. However there are some differences. And because of that we chose to write the rules.

$$\textbf{(AX)} \quad [\![x : \sigma \vdash x : \sigma]\!] = id : [\![\sigma]\!] \to [\![\sigma]\!]$$

$$\textbf{(APP)} \quad \frac{[\![\Gamma \vdash M : \sigma \multimap \tau]\!] = f : [\![\Gamma]\!] \to [\![\sigma]\!] \otimes [\![\tau]\!] \qquad [\![\Delta \vdash N : \sigma]\!] = g : [\![\Delta]\!] \to [\![\sigma]\!]}{[\![\Gamma, \Delta \vdash MN : \tau]\!] = app \circ (f \otimes g) : [\![\Gamma]\!] \otimes [\![\Delta]\!] \to [\![\tau]\!]}$$

$$\textbf{(ABS)} \quad \frac{[\![\Gamma, x : \sigma \vdash M : \tau]\!] = f : [\![\Gamma]\!] \otimes [\![\sigma]\!] \to [\![\tau]\!]}{[\![\Gamma \vdash \lambda x.M : \sigma \multimap \tau]\!] = \lambda(f) : [\![\Gamma]\!] \to [\![\sigma \multimap \tau]\!]}$$

$$\textbf{(SEQ)} \quad \frac{[\![\Gamma \vdash M : \sigma]\!] = f : [\![\Gamma]\!] \to [\![\sigma]\!] \qquad [\![\Delta, x : \sigma \vdash N : \tau]\!] = g : [\![\Delta]\!] \otimes [\![\sigma]\!] \to [\![\tau]\!]}{[\![\Gamma, \Delta \vdash x \leftarrow M; N : \tau]\!] = g \circ s \circ (f \otimes id_\Delta) : [\![\Gamma]\!] \otimes [\![\Delta]\!] \to [\![\tau]\!]}$$

$$\textbf{(PAIR)} \quad \frac{[\![\Gamma \vdash V : \sigma]\!] = f : [\![\Gamma]\!] \to [\![\sigma]\!] \qquad [\![\Delta \vdash U : \tau]\!] = g : [\![\Delta]\!] \to [\![\tau]\!]}{[\![\Gamma, \Delta \vdash V \otimes U : \sigma \otimes \tau]\!] = f \otimes g : [\![\Gamma]\!] \otimes [\![\Delta]\!] \to [\![\sigma]\!] \otimes [\![\tau]\!]}$$

$$\textbf{(UNIT)} \quad \frac{}{[\![\vdash * : 1]\!] = id : \{*\} \to \{*\}}$$

Table 15: Denotational semantics for the linear λ-calculus

The modifications made are imposed by linear logic

- since variables cannot be discarded, we changed the environments of all rules
- we also had to adapt the rules that have the morphism pairing to tensor

We can adapt Lemma 4.18 and Lemma 4.19 for the context of the denotational semantics.

Lemma 4.20 (Exchange 2)**.** Consider a judgment of the form $\Gamma, x : \sigma, y : \tau, \Delta \vdash M : \gamma$. Then the following property hold:

$$[\![\Gamma, x : \sigma, y : \tau, \Delta \vdash M : \gamma]\!] = [\![\Gamma, y : \tau, x : \sigma, \Delta \vdash M : \gamma]\!] \circ s$$

Proof. The proof is made on induction over $[\![\Gamma, x : \sigma, y : \tau, \Delta \vdash M : \gamma]\!]$. Cases are distinguished by the form of M. □

Lemma 4.21 (Substitution 2)**.** If $[\![\Gamma, x : \sigma \vdash M : \tau]\!]$ and $[\![\Omega \vdash N : \sigma]\!]$ then $[\![\Gamma, \Omega \vdash M[N/x] : \tau]\!] = [\![\Gamma, x : \sigma \vdash M : \tau]\!] \circ (id \otimes [\![\Omega \vdash N : \sigma]\!])$.

Proof. The proof is made under induction on the derivation of $[\![\Gamma, x : \gamma \vdash M : \tau]\!]$. Cases are distinguished by the last rule used in the derivation. □

The formulation of the soundness theorem is like the one presented in the classical setting.

Theorem 4.22. Let V be a value and M a term. Then the following condition hold:

$$\text{for all } \vdash M : \sigma \text{ if } M \Downarrow V \text{ then, } \vdash V : \sigma \text{ and } [\![\vdash M : \sigma]\!] = [\![\vdash V : \sigma]\!]$$

Proof. The proof is made under induction over $\Downarrow$ rules and cases are distinguished by the form of M. □

4.2.4 Classical λ-calculus with effects

We are interested in extending both classical and linear λ-calculus with wait call operations. Such operations reflect the fact that computations are *not* instantaneous, which is particularly relevant for quantum computation and specifically for quantum decoherence.

We begin by extending the classical λ-calculus first. For that we are going to use the category $\mathbf{Set_N}$ where:

- objects are sets
- the arrows are functions of the type $A \to B \times Nat$, being Nat the natural numbers
- the identity arrow is defined as follows:

$$\begin{aligned} id_A : & A \to A \times Nat \\ & a \mapsto (a, 0) \end{aligned}$$

- the composition of two arrows $f : A \to B \times Nat$ and $g : B \to C \times Nat$ is defined as follows (via Haskell notation):

$$\begin{aligned} g \circ f : & A \to C \times Nat \\ & a \mapsto \text{let } (b, n) = f(a),\ (c, m) = g(b) \end{aligned}$$

The elements of the set $A \times Nat$ are of the form (a, n), where $a \in A$ and $n \in Nat$.

Since now we are going to consider effects, it returns: the notations $\xrightarrow{n}$, $\xrightarrow{n}\!\!\!\!\rightarrow$, and $\Downarrow^n$ on the operational semantics; the distinction of values and producers in typing judgments.

Instead of rewriting the rules, we are going to locate them to ease the reader. The set of rules of the small-step operational semantics is on Table 12, the 'multi-step' operational semantics is defined in Definition 4.10, and the set of rules of the big-step operational semantics is on Table 4.11.

Regarding the denotational semantics, we developed a functor between the categories **Set** and $\mathbf{Set_N}$, such that every set A from **Set** its mapped to a set $A \times Nat$ from $\mathbf{Set_N}$, and for each **Set**-morphism is associated the correspondence:

$$\begin{aligned} Jf : & A \to B \times Nat \\ & a \mapsto (f(a), 0) \end{aligned}$$

The creation of this functor is necessary because typing judgments of type value have the elements in **Set**, while typing judgments of type producer have the elements in $\mathbf{Set_N}$. Henceforth, it is needed a way to jump from **Set** to $\mathbf{Set_N}$, *i.e.* a functor. Furthermore, one can see $\mathbf{Set_N}$ as the category where the effects occur.

With the introduction of effects, the morphisms *app* and $\lambda(f)$ have suffered a modification. Furthermore, we take this moment to present a new morphism $[\![wait_n]\!]$ that is used to interpret the delays on terms. The morphism *app* is now:

$$\begin{aligned} app : & (A \times Nat)^B \times B \to A \times Nat \\ & (f, b) \mapsto f(b) \end{aligned}$$

For $\lambda(f)$, let $f : C \times B \to A \times Nat$.

$$\begin{aligned} \lambda f : & C \to (A \times Nat)^B \\ & (\lambda f)c \mapsto (b \mapsto f(c, b)) \end{aligned}$$

At last, the function $[\![wait_n]\!]$ is defined as:

$$\begin{aligned} [\![wait_n]\!] : & A \times Nat \to A \times Nat \\ & (a, m) \mapsto (a, m + n) \end{aligned}$$

A remark about the denotational semantics. With the introduction of effects in the λ-calculus, either classical or linear, we have two options to form the denotational semantics: distinguish between *values* and *producers* or not. It is evident from Table 9 that we have chosen the first approach. But why? Because without distinguish values and producers, the denotational semantics would be more difficult to work with. Since we are in the beginning, we think that is better to work with simple things first, see them working, and then complicate them, instead of complicate them right at the start. However, by distinguish between values and producers, we have a restriction: values cannot be delayed, as one shall see in Table 16.

The set of rules of the denotational semantics is on Table 16.

$$\textbf{(AX)} \quad \llbracket \Gamma, x : \sigma \vdash^v x : \sigma \rrbracket = \pi_x : \llbracket \Gamma \rrbracket \times \llbracket \sigma \rrbracket \to \llbracket \sigma \rrbracket$$

$$\textbf{(APP)} \quad \frac{\llbracket \Gamma \vdash^v M : \sigma \to \tau \rrbracket = f : \llbracket \Gamma \rrbracket \to (\llbracket \tau \rrbracket \times Nat)^{\llbracket \sigma \rrbracket} \qquad \llbracket \Gamma \vdash^v N : \sigma \rrbracket = g : \llbracket \Gamma \rrbracket \to \llbracket \sigma \rrbracket}{\llbracket \Gamma \vdash^w MN : \tau \rrbracket = app \circ \langle f, g \rangle : \llbracket \Gamma \rrbracket \to \llbracket \tau \rrbracket \times Nat}$$

$$\textbf{(ABS)} \quad \frac{\llbracket \Gamma, x : \sigma \vdash^w M : \tau \rrbracket = f : \llbracket \Gamma \rrbracket \times \llbracket \sigma \rrbracket \to \llbracket \tau \rrbracket \times Nat}{\llbracket \Gamma \vdash^v \lambda x.M : \sigma \to \tau \rrbracket = \lambda f : \llbracket \Gamma \rrbracket \to (\llbracket \tau \rrbracket \times Nat)^{\llbracket \sigma \rrbracket}}$$

$$\textbf{(WAIT)} \quad \frac{\llbracket \Gamma \vdash^w M : \tau \rrbracket = f : \llbracket \Gamma \rrbracket \to \llbracket \tau \rrbracket \times Nat}{\llbracket \Gamma \vdash^w wait_n(M) : \tau \rrbracket = \llbracket wait_n \rrbracket \circ f : \llbracket \Gamma \rrbracket \to \llbracket \tau \rrbracket \times Nat}$$

$$\textbf{(SEQ)} \quad \frac{\llbracket \Gamma \vdash^w M : \sigma \rrbracket = f : \llbracket \Gamma \rrbracket \to \llbracket \sigma \rrbracket \times Nat \qquad \llbracket \Gamma, x : \sigma \vdash^w N : \tau \rrbracket = g : \llbracket \Gamma \rrbracket \times \llbracket \sigma \rrbracket \to \llbracket \tau \rrbracket \times Nat}{\llbracket \Gamma \vdash^w x \leftarrow M; N : \tau \rrbracket = (id \times (+)) \circ (g \times id) \circ \langle id, f \rangle : \llbracket \Gamma \rrbracket \to \llbracket \tau \rrbracket \times Nat}$$

$$\textbf{(PAIR)} \quad \frac{\llbracket \Gamma \vdash^v V : \sigma \rrbracket = f : \llbracket \Gamma \rrbracket \to \llbracket \sigma \rrbracket \qquad \llbracket \Gamma \vdash^v U : \tau \rrbracket = g : \llbracket \Gamma \rrbracket \to \llbracket \tau \rrbracket}{\llbracket \Gamma \vdash^v \langle V, U \rangle : \sigma \times \tau \rrbracket = \langle f, g \rangle : \llbracket \Gamma \rrbracket \to \llbracket \sigma \rrbracket \times \llbracket \tau \rrbracket}$$

$$\textbf{(UNIT)} \quad \frac{}{\llbracket \Gamma \vdash^v * : 1 \rrbracket = f : \llbracket \Gamma \rrbracket \to \{*\}}$$

Table 16: Denotational semantics for the classical λ-calculus with effects

From the rules on Table 16 we want to highlight three: **(APP)**, **(WAIT)**, and **(SEQ)**. From the premises of the first one, one notices that we cannot have terms like $M(wait_n(N))$, *i.e.* values cannot be delayed, as previously stated. The reason for that is the premise $\llbracket \Gamma \vdash^v N : \sigma \rrbracket = g : \llbracket \Gamma \rrbracket \to \llbracket \sigma \rrbracket$ that is interpreted in **Set**, which is the category where effects are not considered. On the other hand, despite the typing judgment be of type value, the premise $\llbracket \Gamma \vdash^v M : \sigma \to \tau \rrbracket = f : \llbracket \Gamma \rrbracket \to (\llbracket \tau \rrbracket \times Nat)^{\llbracket \sigma \rrbracket}$ acts on $\mathbf{Set_N}$ and, consequently, it allows us to delay the execution of functions.

The rule **(WAIT)**, appears with the introduction of timing constraints. So, it is straightforward that its job is to interpret the delay found on λ-terms. This is done by composing the function f with the delay function $\llbracket wait_n \rrbracket$, which indicates that the term is going to be delayed n units of time.

At last, the rule **(SEQ)** is highlighted because of its conclusion definition. Its explanation is better understood through the following scheme:

$$\llbracket \Gamma \rrbracket \xrightarrow{\langle id, f \rangle} \llbracket \Gamma \rrbracket \times \llbracket \sigma \rrbracket \times Nat \xrightarrow{g \times id} \llbracket \tau \rrbracket \times Nat \times Nat \xrightarrow{id \times (+)} \llbracket \tau \rrbracket \times Nat$$

The lemmas of Subsection 4.2.2 and the soundness theorem need to be proved again, because of the introduction of effects on the λ-calculus. Recall that in lemmas and theorems, we do not distinguish if a term is a value or a producer, henceforth $\llbracket \Gamma \vdash M : \sigma \rrbracket$ can be a value, $\llbracket \Gamma \vdash^v M : \sigma \rrbracket$, or a producer, $\llbracket \Gamma \vdash^w M : \sigma \rrbracket$.

Lemma 4.23 (Exchange)**.** Consider a judgment of the form $\Gamma, x : \sigma, y : \tau, \Delta \vdash M : \gamma$. The the following conditions hold:

- $\Gamma, y : \tau, x : \sigma, \Delta \vdash M : \gamma$
- $\llbracket \Gamma, x : \sigma, y : \tau, \Delta \vdash M : \gamma \rrbracket = \llbracket \Gamma, y : \tau, x : \sigma, \Delta \vdash M : \gamma \rrbracket \circ \phi$

Proof. The proof is made using induction over $\llbracket \Gamma, x : \sigma, y : \tau, \Delta \vdash M : \gamma \rrbracket$ rules. The cases are distinguished by the form of M. □

Lemma 4.24 (Weakening)**.** Consider a judgment of the form $\Gamma \vdash M : \sigma$ and a variable $x : \gamma \notin \Gamma$. Then the following conditions hold:

- $\Gamma, x : \gamma \vdash M : \sigma$
- $\llbracket \Gamma, x : \gamma \vdash M : \sigma \rrbracket = \llbracket \Gamma \vdash M : \sigma \rrbracket \circ \pi_1$

Proof. The proof is made under induction over $\llbracket \Gamma, x : \gamma \vdash M : \sigma \rrbracket$. The cases are distinguished by the form of M. □

So, the substitution is presented on Lemma 4.25.

Lemma 4.25 (Substitution)**.** If $\Gamma, x : \sigma \vdash M : \tau$ and $\Gamma \vdash^v N : \sigma$ then

1. $\Gamma \vdash^w M[N/x] : \tau$
2. $\llbracket \Gamma \vdash M[N/x] : \tau \rrbracket = \llbracket \Gamma, x : \sigma \vdash M : \tau \rrbracket \circ \langle id, \llbracket \Gamma \vdash^v N : \sigma \rrbracket \rangle$ [2]

[2] FiXme Note: Ver http://alfa.di.uminho.pt/~nevrenato/pdfs/neves2020.pdf. Este documento prova uma versão mais geral disto.

Proof. The proof is made under induction over the derivation of $[\![\Gamma, x : \gamma \vdash M : \tau]\!]$. Cases are distinguished by the last rule used in the derivation. □

Theorem 4.26 presents us the soundness theorem, that relates the denotational and operational semantics.

Theorem 4.26 (Soundness I)**.** Let V be a value and M a term. Then the following conditions hold:

1. for all $\vdash^v V : \sigma$ if $V \Downarrow^n V'$, then $\vdash^v V' : \sigma$ and $[\![\vdash^v V : \sigma]\!] = [\![\vdash^v V' : \sigma]\!]$
2. for all $\vdash^w M : \sigma$ if $M \Downarrow^n V'$ then $\vdash^v V' : \sigma$ and $[\![\vdash^w M : \sigma]\!] = [\![wait_n]\!] \circ J([\![\vdash^v V' : \sigma]\!])$ [3]

Proof. The proof is made under induction over $\Downarrow$ rules, and cases are distinguished by the form of M. □

Another property that one can verifies is the soundness of denotational semantics in order to w-equivalence:

Theorem 4.27 (Soundness II)**.** If $M =_w N$ then $M \Downarrow^n V$ and $N \Downarrow^m V$, and $[\![\vdash M : \sigma]\!] = [\![wait_n]\!] \circ J([\![\vdash^v V : \tau]\!])$ and $[\![\vdash N : \sigma]\!] = [\![wait_n]\!] \circ J([\![\vdash^v V : \tau]\!])$. [4]

Proof. The proof is straightforward. It was already proved that the big-step operational semantics is sound in respect to w-equivalence (Theorem 4.13) and to the denotational semantics (Theorem 4.26). What is left to proof is the soundness of the denotational semantics in order to w-equivalence. Well, since the denotational semantics and the big-step operational semantics are equivalent, and that the big-step operational semantics is equivalent to w-equivalence, it follows from transitivity that denotational semantics is equivalent to w-equivalence. □

Until here, we have detailed the denotational semantics of classical λ-calculus and presented some results. However, there are some abstract notions that can also be used to help us in the development of the denotational semantics, such like *monads* and *equational theory*. The rest of this subsection is dedicated to them.

The extension of *classical* λ-calculus with wait call operations can be framed in the setting of λ-calculus with algebraic operations (see [PP01, LPT03]). Let us elaborate on this last statement. Recall that an equational theory is a tuple (Σ, E) where Σ is a set of operations of finite arity and E is a set of equations relating the operations (see details in [Cro93]).

Example 4.28 (The equational theory of wait calls)**.** The tuple (Σ, E) for this theory is defined as $\Sigma = \{\text{wait}_n : 1 \mid n \in \mathbb{N}\}$ – i.e. we have a wait call operation of arity 1 (i.e. takes just *one* argument) for each natural number n – and $E = \{\text{wait}_n(\text{wait}_m(x)) = \text{wait}_{n+m}(x), \text{wait}_0(x) = x\}$ for a free variable x.

Next, it is a well-known fact that every equational theory induces a *monad* and that monads serve as models of *λ-calculus extended with the respective equational theory*. Before detailing this bit, let us first recall what is a monad.

Definition 4.29 (Monads and Kleisli categories [Wad92, Wad95])**.** A monad $\mathbb{T} = (T, \eta, (-)^\star)$ on a category C is an endomap T on $|\mathsf{C}|$, together with a $|\mathsf{C}|$-indexed family of morphisms $\eta_X : X \to TX$ and a so-called *Kleisli lifting* sending each $f : X \to TY$ to $f^\star : TX \to TY$ and obeying the following laws: $\eta^\star = \mathsf{id}$, $f^\star \cdot \eta = f$, $(f^\star \cdot g)^\star = f^\star \cdot g^\star$ (it follows from this definition that T extends to a functor and η to a natural transformation).

Every monad $\mathbb{T}$ induces a so-called Kleisli category $\mathsf{C}_\mathbb{T}$ whose objects are the same than those of C and whose morphisms $X \to Y$ are those of type $X \to TY$ available in C. The identity morphism for X in $\mathsf{C}_\mathbb{T}$ is the unit $\eta_X : X \to TX$ and composition of morphisms in $\mathsf{C}_\mathbb{T}$ is defined via the Kleisli lifting. Finally, there is a functor $J : \mathsf{C} \to \mathsf{C}_\mathbb{T}$ that acts as the identity on objects and that post-composes morphisms with the unit of the monad $\mathbb{T}$.

Informally, a monad can be seen as a tool that is used to apply effects. Consequently, a Kleisli category can be seen as a category of effects, *i.e.* where the effects are applied. Regarding what was previously shown, the category $\mathbf{C}$ corresponds to $\mathbf{Set}$ and $\mathbf{C}_\mathbb{T}$ to $\mathbf{Set_N}$.

The functor $J : \mathsf{C} \to \mathsf{C}_\mathbb{T}$ serves as a model for *λ-calculus extended with the equational theory* respective to $\mathbb{T}$. Specifically, the category C is used to interpret classical λ-terms and the category $\mathsf{C}_\mathbb{T}$ is used to interpret λ-terms involving the equational theory. This is described in [LPT03] where the functor $J : \mathsf{C} \to \mathsf{C}_\mathbb{T}$ is called a Freyd category.

We mentioned that every equational theory induces a monad $\mathbb{T}$ on Set. The functorial component of the monad $\mathbb{T}$ is build by mapping a set X to the set of terms that can be built over Σ and X. For example, in the case of wait calls,

$$x = \text{wait}_0(x) \in TX, \qquad \text{wait}_1(x) \in TX, \qquad \text{wait}_2(x) \in TX, \qquad \text{wait}_2(\text{wait}_1(x)) = \text{wait}_3(x) \in TX$$

[3] FiXme Note: Ver http://alfa.di.uminho.pt/~nevrenato/pdfs/neves2020.pdf. Este documento prova uma versão mais geral disto.
[4] FiXme Note: Ver http://alfa.di.uminho.pt/~nevrenato/pdfs/neves2020.pdf. Este documento prova uma versão mais geral disto.

where $x \in X$. Hence, the functor T can be equivalently defined as the one that sends X to $X \times Nat$. The unit $X \to TX$ is defined by $x \mapsto (x, 0)$. The Kleisli lifting $(-)^\star$ of the monad is defined by,

$$f^\star(x, n) = \texttt{let} \quad f(x) = (y, m) \quad \texttt{in} \quad (y, n + m)$$

By general results in [PP01] this model is sound for β-reduction + the equations of wait calls that were presented above.

The generation of a monad from the equational theory of wait calls works in any Cartesian closed category and not just the category Set of sets.

4.2.5 Linear λ-calculus with effects

The term formation rules for linear λ-calculus with effects is presented on Table 17.

$$\begin{array}{lll|ll}
\textbf{(AX)} & x : a \vdash^v x : a & & \textbf{(*)} & \vdash^v * : 1 \\
(\to \textbf{E}) & \dfrac{\Gamma \vdash^v M : \sigma \multimap \tau \quad \Delta \vdash^v N : \sigma}{\Gamma, \Delta \vdash^w MN : \tau} & & (\to \textbf{I}) & \dfrac{\Gamma, x : \sigma \vdash^w M : \tau}{\Gamma \vdash^v \lambda x.M : \sigma \multimap \tau} \\
(\to \textbf{W}) & \dfrac{\Gamma \vdash^w M : \tau}{\Gamma \vdash^w wait_n(M) : \tau} & & \textbf{(P)} & \dfrac{\Gamma \vdash^v V : \sigma \qquad \Delta \vdash^v U : \tau}{\Gamma, \Delta \vdash^v V \otimes U : \sigma \otimes \tau} \\
\textbf{(S)} & \dfrac{\Gamma \vdash^w M : \sigma \qquad \Delta, x : \sigma \vdash^w N : \tau}{\Gamma, \Delta \vdash^w x \leftarrow M; N : \tau} & &
\end{array}$$

Table 17: Linear type system

Recall that in linear logic variables cannot de copied nor discarded and they can only be used once. This effects echoes in the rules of Table 17. For example the environment of **(AX)** only as the variable x, while in the classical case, in Table 9 it also as a set Γ. Furthermore, it is implicit that in the environment with Γ and Δ there are not shared variables, *i.e.* $\Gamma \cap \Delta = \varnothing$.

Regarding the denotational semantics, we are going to use $\mathbf{Vect_k}$ as the category in which effects are not considered and in which effects are considered. Why? $\mathbf{Vect_k}$ has coproducts, which are interpreted as direct sums in that category. Henceforth, it allows us to give two interpretations for $\mathbf{Vect_k}$:

1. vector spaces are not represented through direct sums (effects are not considered)
2. vector spaces are represented through direct sums (effects are considered)

To understand what is direct sum, attention on the example:

Example 4.30. Direct sums are denoted by $\oplus$. Given two arrays $v = (a, b, c)$ and $u = (d, e)$ its direct sum is

$$v \oplus u = (a, b, c, d, e)$$

i.e. the array u was inserted in the end of v. Since, direct sums gives us arrays, and since every element on the array has a position, we can use that position to identify a temporal instant. Henceforth, the instant $t = 0$ corresponds to the element a while $t = 3$ corresponds to the element d.

From this example, is straightforward to understand why we are going to use the notion of direct sums to represent the temporal evolution.

As in the classical setting, we need to define a functor that goes from $\mathbf{Vect_k}$ to $\mathbf{Vect_k}$. Concretely, from $\mathbf{Vect_k}$ where the effects are not considered to $\mathbf{Vect_k}$ where we consider effects. The functor D maps every object V in $\mathbf{Vect_k}$ to an object $D(V) = \bigoplus_{n \in \mathbb{N}} V_n$ in $\mathbf{Vect_k}$, and for each $\mathbf{Vect_k}$-morphism is associated the correspondence:

$$\begin{aligned} D(f) : & D(V) \to D(U) \\ & \bigoplus_{n \in \mathbb{N}} f = \left[(V_1, V_2, \dots) \mapsto (f(V_1), f(V_2), \dots) \right] \end{aligned}$$

Before presenting the rules of the denotational semantics, we are going to approach the point of view from monads and equational theory.

The generation of a monad from the theory of wait calls actually works in any category with coproducts. This includes those categories that are traditionally used in quantum computing (see [SV08])

Definition 4.31. Let $\mathbf{C}$ be a category with coproducts. Then the monad $\mathbb{T}$ of durations is defined as $X \mapsto \coprod_{n \in \mathbb{N}} X$ with $\eta(x) = i_0(x)$ and

$$f^\star(x, n) = \texttt{let} \quad f(x) = (y, m) \quad \texttt{in} \quad (y, n + m)$$

Analogous to the classical case, a monad $\mathbb{T}$ in a monoidal-closed category can also be used as a model for linear λ-calculus extended with an equational theory.

The notation for the Kleisli lifting is abbreviated. Recall that a direct sum is an array of the form $(V_0, V_1, \dots, V_n, \dots)$, and that we can identify an element on the array by its index. By doing that, V_0 can be represented by $(V_0, 0)$, V_1 by $(V_1, 1)$ and so on. This is the notation that the Kleisli lifting uses.

We wanted to have this point of view from the monad, because the Kleisli lifting is needed in one rule of the denotational semantics. However, it lacks to define how the delay function acts:

$$\begin{aligned}[\![wait_m]\!] : & D(V) \to D(V) \\ & (V_0, \dots, V_n, \dots) \to (0, \dots V_0, \dots, V_n, \dots)\end{aligned}$$

Informally, what the delay function does is shift $(V_0, \dots, V_n, \dots)$ by m indexes. Another way to see this, is by creating an array of size m full of zeros and do the direct sum: $(0, \dots, 0) \oplus (V_0, \dots, V_n, \dots) = (0, \dots V_0, \dots, V_n, \dots)$.

According to [SV08], the interpretation of $\multimap$ is: $[\![A \multimap B]\!] = [\![A]\!] \otimes [\![B]\!]$.

The action of the meaning function on typing judgments is as follows:

$$\begin{aligned}[\![\Gamma \vdash^v V : \sigma]\!] &= f : [\![\Gamma]\!] \to [\![\sigma]\!] \\ [\![\Gamma \vdash^v \lambda x.M : \sigma \multimap \tau]\!] &= f : [\![\Gamma]\!] \to [\![\sigma \multimap \tau]\!] = [\![\Gamma]\!] \to [\![\sigma]\!] \otimes [\![\tau]\!] \\ [\![\Gamma \vdash^w M : \sigma]\!] &= f : [\![\Gamma]\!] \to D[\![\sigma]\!]\end{aligned}$$

The interpretation of the first two is straightforward, once we are only leading with values. The last one, in its turn, is a bit more complicated. Since it is a producer, we know that the final result must be on the category of effects. That is the reason why we use the functor D in the interpretation of σ, *i.e.* in $[\![\sigma]\!]$. However, on the rules of Table 18 we omit the action of the functor.

Now, for the set of rules of the denotational semantics:

$$\textbf{(AX)} \quad [\![x : \sigma \vdash^v x : \sigma]\!] = id : [\![\sigma]\!] \to [\![\sigma]\!]$$

$$\textbf{(APP)} \quad \frac{[\![\Gamma \vdash^v M : \sigma \multimap \tau]\!] = f : [\![\Gamma]\!] \to [\![\sigma]\!] \otimes [\![\tau]\!] \qquad [\![\Delta \vdash^v N : \sigma]\!] = g : [\![\Delta]\!] \to [\![\sigma]\!]}{[\![\Gamma, \Delta \vdash^w MN : \tau]\!] = app \circ (f \otimes g) : [\![\Gamma]\!] \otimes [\![\Delta]\!] \to [\![\tau]\!]}$$

$$\textbf{(ABS)} \quad \frac{[\![\Gamma, x : \sigma \vdash^w M : \tau]\!] = D(f) : [\![\Gamma]\!] \otimes [\![\sigma]\!] \to [\![\tau]\!]}{[\![\Gamma \vdash^v \lambda x.M : \sigma \multimap \tau]\!] = \lambda(D(f)) : [\![\Gamma]\!] \to [\![\sigma \multimap \tau]\!]}$$

$$\textbf{(WAIT)} \quad \frac{[\![\Gamma \vdash^w M : \tau]\!] = D(f) : [\![\Gamma]\!] \to [\![\tau]\!]}{[\![\Gamma \vdash^w wait_n(M) : \tau]\!] = [\![wait_n]\!] \circ D(f) : [\![\Gamma]\!] \to [\![\tau]\!]}$$

$$\textbf{(SEQ)} \quad \frac{[\![\Gamma \vdash^w M : \sigma]\!] = f : [\![\Gamma]\!] \to [\![\sigma]\!] \qquad [\![\Delta, x : \sigma \vdash^w N : \tau]\!] = g : [\![\Delta]\!] \otimes [\![\sigma]\!] \to [\![\tau]\!]}{[\![\Gamma, \Delta \vdash^w x \leftarrow M; N : \tau]\!] = g^\star \circ s \circ (f \otimes id_\Delta) : [\![\Gamma]\!] \otimes [\![\Delta]\!] \to [\![\tau]\!]}$$

$$\textbf{(PAIR)} \quad \frac{[\![\Gamma \vdash^v V : \sigma]\!] = f : [\![\Gamma]\!] \to [\![\sigma]\!] \qquad [\![\Delta \vdash^v U : \tau]\!] = g : [\![\Delta]\!] \to [\![\tau]\!]}{[\![\Gamma, \Delta \vdash^v V \otimes U : \sigma \otimes \tau]\!] = f \otimes g : [\![\Gamma]\!] \otimes [\![\Delta]\!] \to [\![\sigma]\!] \otimes [\![\tau]\!]}$$

$$\textbf{(UNIT)} \quad \frac{}{[\![\vdash^v * : 1]\!] = id : \{*\} \to \{*\}}$$

Table 18: Denotational semantics for the linear λ-calculus with effects

The rule **(SEQ)** uses the Kleisli lifting on the function g. Why? When one performs $f \otimes id_\Delta$, the result is on the category with effects, and from now on we need to act on this category. However, g acts on the category without effects. So, to make g act on the category with effects, we need to apply the Kleisli lifting.

The results obtained for the linear λ-calculus without effects, can also be reproduced here.

Lemma 4.32 (Exchange)**.** Consider a judgment of the form $\Gamma, x : \sigma, y : \tau, \Delta \vdash M : \gamma$. Then the following properties hold:

- $\Gamma, y : \tau, x : \sigma, \Delta \vdash M : \gamma$
- $[\![\Gamma, x : \sigma, y : \tau, \Delta \vdash M : \gamma]\!] = [\![\Gamma, y : \tau, x : \sigma, \Delta \vdash M : \gamma]\!] \circ s$

Proof. The proof is made on induction over $[\![\Gamma, x : \sigma, y : \tau, \Delta \vdash M : \gamma]\!]$. Cases are distinguished by the form of M. □

Note that for $M = x$ and $M = *$ Lemma 4.32 is not applicable.

Lemma 4.33 (Substitution)**.** If $\Gamma, x : \sigma \vdash M : \tau$ and $\Omega \vdash^v N : \sigma$ then

1. $\Gamma \vdash^w M[N/x] : \tau$

2. $[\![\Gamma, \Omega \vdash M[N/x] : \tau]\!] = [\![\Gamma, x : \sigma \vdash M : \tau]\!] \circ (id \otimes [\![\Omega \vdash^v N : \sigma]\!])$ [5]

Proof. The proof is made under induction over the derivation of $[\![\Gamma, x : \gamma \vdash M : \tau]\!]$. Cases are distinguished by the last rule used in the derivation. □

The soundness theorem is shown in Theorem 4.34.

Theorem 4.34 (Soundness)**.** Let V be a value and M a term. Then the following conditions hold:

1. for all $\vdash^v V : \sigma$ if $V \Downarrow^n V'$, then $\vdash^v V' : \sigma$ and $[\![\vdash^v V : \sigma]\!] = [\![\vdash^v V' : \sigma]\!]$
2. for all $\vdash^w M : \sigma$ if $M \Downarrow^n V'$ then $\vdash^v V' : \sigma$ and $[\![\vdash^w M : \sigma]\!] = [\![wait_n]\!] \circ D([\![\vdash^v V' : \sigma]\!])$ [6]

Proof. The proof is made under induction over $\Downarrow$ rules, and cases are distinguished by the form of M. □

[5] FiXme Note: Ver http://alfa.di.uminho.pt/~nevrenato/pdfs/neves2020.pdf. Este documento prova uma versão mais geral disto.
[6] FiXme Note: Ver http://alfa.di.uminho.pt/~nevrenato/pdfs/neves2020.pdf. Este documento prova uma versão mais geral disto.

5 Work plan

In this section we present a work plan for the duration of the Map-i doctoral programme: started in September 2019 and ends in July 2022. The project is divided into five work packages as detailed in Table 19.

Work package	Name	Start	End	Acc. duration
1	Timed-Quantum Lambda Calculus	09/19	12/20	15 months
2	Program calculi	01/21	04/21	19 months
3	Quantitative semantics	05/21	10/21	25 months
4	Tool implementation and case-studies	11/21	03/22	31 months
5	Thesis writing	04/22	07/22	35 months

Table 19: Work packages duration

The description of the work packages is given next:

- The goal of the first work package is to study λ-calculus, classical and quantum, and to develop a quantum λ-calculus with temporal notions jointly with a corresponding typing system. The result of this work package will allows us to calculate the execution time of quantum programs and statically ensure that they do not violate certain time thresholds. The work developed in this task will thus be the basis of this project. It is intended that it will result on a submission to an international peer-reviewed conference.

- With the second work package, we plan to extend the result of the previous work package with a notion of equivalence for quantum programs and also study the impact of quantum decoherence on the notion of equivalence. This will be used to provide a suitable equational system for the extended quantum λ-calculus. In practice, it will bring us a tool to suitable compare quantum programs with timing constraints. We aim for submission to an international peer-reviewed conference.

- In the third work package we intend to develop a quantitative semantics for the extended quantum λ-calculus. Concretely, we will develop a suitable notion of distance between quantum programs with timing constraints, and use it to refine the equational system developed in the previous work package. In practice, this will allow the engineer to permit small errors in a controlled, precise way. We expect that this work package will give a submission to an international peer-reviewed conference.

- The fourth work package will use the previous work packages to implement proof-of-concept tools. Specifically, it will use the equational system of the previous work package to implement a (semi-automated) theorem prover, that will allow us to semi-automatically check whether two programs are equivalent or not. In the negative case, it will determine the distance between the two programs. This tool is intended to be used in the analysis of typical quantum algorithms subjected to quantum decoherence. We expect that the work of this task will give a submission to an internation peer-reviewed conference.

- Finally, the last work package will be totally dedicated to writing the PhD thesis.

6 Conclusion

The document's goal was to present the author's PhD project. In Section 1, we motivated the reader, presented our goals, and explained why λ-calculus was chosen as the basis of our work. Sections 2 and 3, presented a part of the necessary background to successfully accomplish the PhD project. Furthermore, these sections were complemented by two appendices, providing basic details of category theory. In Section 4, we presented the work developed since the beginning of the project. At last, in Section 5, we detailed the work plan of the project.

This PhD project is framed in the Map-i doctoral programme, where is necessary to successfully concluded several courses to complete the academical year. In particular, in the author's case:

- three optional trimestral curricular units: *Cyber-Physical Computation*, *Quantum Computing*, and *Engineering Web Applications*. The first two provided some background to the realization of this thesis, while the last one was chosen due to academic curiosity;
- one mandatory semestral curricular unit, Seminar, whose goal was to give (to the PhD students) advice on how to perform successfully in the investigation field;
- the external option, which consisted in the realization of a curricular unit called *Universal Algebra and Categories*, it provided further background to the PhD project;
- another mandatory course called *free option*. In this course we studied the background related with λ-calculus and quantum λ-calculus as well as it was developed the content presented in Section 4.

To conclude this document, we would like to state that throughout this year, the author:

- participated on an article related with quantum simulation, more concretely, the calculation of the ground-state energy of molecules using quantum computers, which was accepted in the journal of Soft Computing;
- wrote a short article jointly with the supervisors Renato Neves and Luís Soares Barbosa, based on its MSc dissertation which extended quantum process algebras with certain temporal notion. The article was published in the TYPES2020 conference's proceedings.

References

[AC98] Roberto M. Amadio and Pierre-Louis Curien. *Domains and Lambda-Calculi*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1998.

[AT10] S. Abramsky and N. Tzevelekos. Introduction to categories and categorical logic. *Lecture Notes in Physics*, page 3–94, 2010.

[Awo06] S. Awodey. *Category Theory*. Oxford Logic Guides. Ebsco Publishing, 2006.

[Chu36] Alonzo Church. An unsolvable problem of elementary number theory. *American journal of mathematics*, 58(2):345–363, 1936.

[Cro93] Roy L Crole. *Categories for types*. Cambridge University Press, 1993.

[GJN18] Sergey Goncharov, Julian Jakob, and Renato Neves. A semantics for hybrid iteration. *arXiv preprint arXiv:1807.01053*, 2018.

[Gro96] Lov K Grover. A fast quantum mechanical algorithm for database search. *arXiv preprint quant-ph/9605043*, 1996.

[Hin97] J. Roger Hindley. *Basic Simple Type Theory*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1997.

[Höf09] Peter Höfner. *Algebraic calculi for hybrid systems*. BoD–Books on Demand, 2009.

[Kah87] Gilles Kahn. Natural semantics. In *Annual symposium on theoretical aspects of computer science*, pages 22–39. Springer, 1987.

[Kni96] Emmanuel Knill. Conventions for quantum pseudocode. Technical report, Los Alamos National Lab., NM (United States), 1996.

[Lan98] S.M. Lane. *Categories for the Working Mathematician*. Graduate Texts in Mathematics. Springer New York, 1998.

[LPT03] Paul Levy, John Power, and Hayo Thielecke. Modelling environments in call-by-value programming languages. *Information and computation*, 185(2):182–210, 2003.

[NBHM16] Renato Neves, Luis S Barbosa, Dirk Hofmann, and Manuel A Martins. Continuity as a computational effect. *Journal of Logical and Algebraic Methods in Programming*, 85(5):1057–1085, 2016.

[Öme98] Bernhard Ömer. A procedural formalism for quantum computing. 1998.

[Pla10] André Platzer. *Logical analysis of hybrid systems: proving theorems for complex dynamics*. Springer Science & Business Media, 2010.

[Plo75] G.D. Plotkin. Call-by-name, call-by-value and the lambda-calculus. *Theoretical Computer Science*, 1(2):125 – 159, 1975.

[PP01] Gordon Plotkin and John Power. Adequacy for algebraic effects. In *Proceedings of FoSSaCS'01*, volume 2030, pages 1–24. Springer, 2001.

[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*, pages 647–658, 2014.

[Sel04] Peter Selinger. Towards a quantum programming language. *Mathematical Structures in Computer Science*, 14(4):527–586, 2004.

[Sel08] Peter Selinger. Lecture notes on the lambda calculus. *arXiv preprint arXiv:0804.3434*, 2008.

[Sho94] Peter W Shor. Algorithms for quantum computation: Discrete logarithms and factoring. In *Proceedings 35th annual symposium on foundations of computer science*, pages 124–134. Ieee, 1994.

[Spi13] David I. Spivak. Category theory for scientists (old version), 2013.

[SS71] Dana Scott and Christopher Strachey. *Toward a mathematical semantics for computer languages*, volume 1. Oxford University Computing Laboratory, Programming Research Group Oxford, 1971.

[SV08] Peter Selinger and Benoît Valiron. On a fully abstract model for a quantum linear functional language. *Electronic Notes in Theoretical Computer Science*, 210:123–137, 2008.

[SZ00] Jeff W Sanders and Paolo Zuliani. Quantum programming. In *International Conference on Mathematics of Program Construction*, pages 80–99. Springer, 2000.

[Tur36] Alan Mathison Turing. On computable numbers, with an application to the entscheidungsproblem. *J. of Math*, 58(345-363):5, 1936.

[Wad92] Philip Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2(04):461–493, 1992.

[Wad95] Philip Wadler. Monads for functional programming. In Johan Jeuring and Erik Meijer, editors, *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques, Båstad, Sweden, May 24-30, 1995, Tutorial Text*, volume 925 of *Lecture Notes in Computer Science*, pages 24–52. Springer, 1995.

A Basic categorical notions

The goal of this section is to introduce simple notions of category theory. The curious reader will find further details of cateogry theory *e.g.* in [AT10, Awo06, Lan98, Spi13].

When explaining the concepts of category theory the examples will be focused on the categories of **Set** and $\mathbf{Vect_k}$. The reason is that these categories are the main examples of Cartesian closed and monoidal closed categories respectively, and these two classes of categories have a main role in the semantics of (linear) λ-calculus.

Let us begin with the definition of a category [Awo06]:

Definition A.1. A category $\mathbf{C}$ consists of:

- Objects: $A, B, C \ldots$
- Arrows/homomorphisms between objects: $f : A \to B, g : B \to C, h : A \to C \ldots$, where, for the case of f, $dom(f) = A$ is the *domain* and $cod(f) = B$ is the *codomain* of f
- An identity arrow for each object A: $id_A : A \to A$
- A composite operation $(\circ)$ such that if $f : A \to B$ and $g : B \to C$, then $g \circ f : A \to C$

such that the following conditions are satisfied:

- Associativity: $\forall f : A \to B, g : B \to C, h : C \to D$

$$h \circ (g \circ f) = (h \circ g) \circ f$$

- Unit: $\forall f : A \to B$

$$f \circ id_A = f = id_B \circ f$$

The set of homomorphisms from an object A to an object B can be denoted as $\mathbf{C}(A, B)$ or $hom_{\mathbf{C}}(A, B)$. The next example introduces the categories of **Set** and $\mathbf{Vect_k}$.

Example A.2. The categories of **Set** and $\mathbf{Vect_k}$ are formed as shown in Table 20.

	Set	$\mathbf{Vect_k}$
Objects	sets	vector spaces
Arrows	functions	linear maps
Identity arrow	identity function	identity linear map
Composition	function composition	linear map composition

Table 20: **Set** and $\mathbf{Vect_k}$ categories

With the definition of category given, we can start to explore basic concepts of category theory. Let us begin with some definitions related with arrows [Awo06].

Definition A.3. In any category $\mathbf{C}$, an arrow $f : A \to B$ is called a:

- *monomorphism*, in short *mono*, if given any $g, h : C \to A$, $f \circ g = f \circ h \Rightarrow g = h$
  it can be represented as $f : A \rightarrowtail B$
- *epimorphism*, in short *epi*, if given any $i, j : B \to D$, $i \circ f = j \circ f \Rightarrow i = j$
  it can be denoted as $f : A \twoheadrightarrow B$

Furthermore, a homomorphism that is both mono and epi is called a *bimorphism*. In the next example we establish a relation between monos and the arrows in the categories **Set** and $\mathbf{Vect_k}$. The same is made for epi.

Example A.4. Attention on Table 21.

	Set	$\mathbf{Vect_k}$
Mono	injective functions	injective linear maps
Epi	surjective functions	surjective linear maps

Table 21: Relation between mono and epi on **Set** and $\mathbf{Vect_k}$ categories

The table tells: in the category **Set**, monomorphisms are injective functions and epimorphisms are surjective functions; in the category $\mathbf{Vect_k}$, monomorphisms are injective linear maps and epimorphisms are surjective linear maps.

If one can go from A to B, then there must be a way to do the inverse, *i.e.* go from B to A. In category theory such is sometimes possible, as the next definition hints at Definition A.5.

Definition A.5. For any category $\mathbf{C}$ and arrow $f : A \to B$ in $\mathbf{C}$, we say that f is an *isomorphism*, in short *iso*, if there exists $g : B \to A$ satisfying

$$j \circ i = id_A, \quad i \circ j = id_B$$

If this condition condition holds we call g the inverse of f.

There is a relation between iso and mono and epi. An isomorphism is a monomorphism and a epimorphism [Awo06]. However the contrary does not always holds.

We can define the opposite of a category $\mathbf{C}$, denoted $\mathbf{C}^{op}$. Formally [AT10]:

Definition A.6. The category $\mathbf{C}^{op}$ has as objects those of $\mathbf{C}$ and as morphisms those of $\mathbf{C}$ but reversal, *i.e.*

$$\mathbf{C}^{op}(A, B) \coloneqq \mathbf{C}(B, A)$$

Composition and identities are inherited from $\mathbf{C}$.

There is a principle between the categories $\mathbf{C}$ and $\mathbf{C}^{op}$, known as the *Principle of duality*, that tells that a statement S is true about $\mathbf{C}$ if and only if its 'dual' is true about $\mathbf{C}^{op}$.

Some objects have some particular characteristics that make them special. That is the case of initial and terminal objects. Intuitively and as suggested by their name, they are objects on which 'everything starts or everything ends'. Formally:

Definition A.7. Consider a category $\mathbf{C}$. An object A is:

- initial, if for every object C there is a unique morphism $A \to C$
- terminal, if for every object C there is a unique morphism $C \to A$

Usually, the initial object is denoted as 0 and the terminal object is denoted as 1. Furthermore, an object that is initial and terminal is known as **zero object**. The next example shows the relation between these concepts and the categories **Set** and $\mathbf{Vect_k}$.

Example A.8. Attention on Table 22:

	Set	$\mathbf{Vect_k}$
Initial objects	empty set	vector space of dimension zero
Terminal objects	singletons (sets with one element)	vector space of dimension zero
Zero objects	——-	vector space of dimension zero

Table 22: Relation of initial, terminal, and zero object on the categories of **Set** and $\mathbf{Vect_k}$

The next concepts we present are product and coproduct.

Let us proceed to the definition of a product:

Definition A.9. In any category $\mathbf{C}$, a product of the objects A and B consists of an object $P = A \times B$ and arrows $\pi_1 : P \to A$ and $\pi_2 : P \to B$, such that for every X in the objects of $\mathbf{C}$ with arrows $f : X \to A$ and $g : X \to B$ there exists a **unique** morphism $u : X \to P$ such that $f = \pi_1 \circ u$ and $g = \pi_2 \circ u$. Diagrammatically (Figure 2):

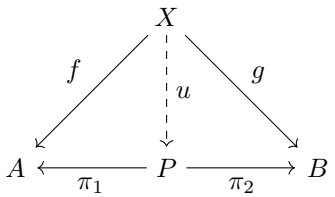


Figure 2: Product of the objects A and B

The unique morphism u, also represented as $\langle f, g \rangle$, is called the *pairing* of f and g. The morphisms π_1 and π_2 are known as projections.

Example A.10. Relating this concept and the category **Set**, the product of $A \times B$ is the Cartesian product between the sets A and B. In the category of $\mathbf{Vect_k}$, products are direct sums.

Now we present the dual concept of product, the coproduct.

Definition A.11. Consider a category $\mathbf{C}$. A coproduct of the objects A and B consists of an object $P = A + B$ and arrows $i_1 : A \to P$ and $i_2 : B \to P$, such that for every X in the objects of $\mathbf{C}$ with arrows $f : A \to X$ and $g : B \to X$ there exists a **unique** morphism $u : P \to X$ such that $f = u \circ i_1$ and $g = u \circ i_2$. Diagrammatically (Figure 3):

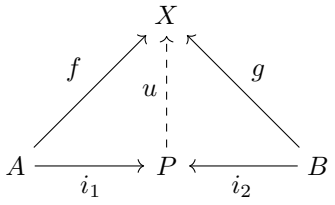


Figure 3: Coproduct of the objects A and B

The unique morphism u, also represented as $[f, g]$, is called the *copairing* of f and g. The morphisms i_1 and i_2 are known as injections.

Example A.12. Relating this concept and the category **Set**, the coproduct of $A + B$ is the disjoint union of sets. In the category of $\mathbf{Vect_k}$, direct sums are coproducts.

The basics of category proceeds with concepts such as *pullbacks*, *equalisers*, *limits*, *colimits*. Since we do not use them, the reader may look at them in [AT10].

We now ended the concepts that relates arrows and objects of a category. However, category theory is so powerful that allows us to stablish relations also between different categories. To do that, we use the concept of a *functor*:

Definition A.13. A functor $F : \mathbf{C} \to \mathbf{D}$ between categories $\mathbf{C}$ and $\mathbf{D}$ maps any object A of $\mathbf{C}$ to an object $F(A)$ of $\mathbf{D}$ and any arrow $f : A \to B$ in $\mathbf{C}$ to an arrow $F(f) : F(A) \to F(B)$ in $\mathbf{D}$, such that

- $F(id_A) = id_{F(a)}$, for any object A in $\mathbf{C}$
- $F(g \circ f) = F(g) \circ F(f)$, for any arrows $f : A \to B$ and $g : B \to C$ in $\mathbf{C}$

These conditions are known as *functoriality*.

Like the arrows, functors also have relevant properties [AT10].

Definition A.14. A functor $F : \mathbf{C} \to \mathbf{D}$ is:

- **faithful** if each map $F_{A,B} : \mathbf{C}(A, B) \to \mathbf{D}(FA, FB)$ is injective
- **full** if each each map $F_{A,B} : \mathbf{C}(A, B) \to \mathbf{D}(FA, FB)$ is surjective
- an **embedding** if F is full, faithful, and injective on objects
- an **equivalence** if F is full, faithful, and *essentially surjective*: *i.e.* for every object B of $\mathbf{D}$ there is an object A of $\mathbf{C}$ such that $F(A) \cong B$
- an *isomorphism* if there is a functor $G : \mathbf{D} \to \mathbf{C}$ such that

$$G \circ F = id_{\mathbf{C}}, \quad F \circ G = id_{\mathbf{D}}$$

Furthermore, we say that two categories $\mathbf{C}$ and $\mathbf{D}$ are isomorphic, denoted $\mathbf{C} \cong \mathbf{D}$, if there is an isomorphism between them.

Since functors also maps arrows, one may question if certain property of an arrow is *preserved* or *reflected*. To explain these concepts consider P as a property of arrows. A functor $F : \mathbf{C} \to \mathbf{D}$ is said to

- preserve P if whenever f satisfies P, so does $F(f)$
- reflect P if whenever $F(f)$ satisfies P, so does f

It is well-known that:

- all functors preserve isomorphisms
- faithful functors reflect mono and epi
- full and faithful functors reflect isomorphisms
- equivalences preserve mono and epi

Until now we saw what it is a category and a functor. However, it is possible to establish morphisms between functors. Those are called *natural transformations* and are formally defined as follows [AT10]:

Definition A.15. Let $F, G : \mathbf{C} \to \mathbf{D}$ be functors. A natural transformation

$$t : F \to G$$

is a family of morphisms in $\mathbf{D}$ indexed by objects A of $\mathbf{C}$,

$$\{t_a : FA \to GA\}_{A \epsilon Obj(\mathbf{C})}$$

such that, for all $f : A \to B$, the following diagram commutes.

$$\begin{array}{ccc} FA & \xrightarrow{Ff} & FB \\ {\scriptstyle t_A}\downarrow & & \downarrow{\scriptstyle t_B} \\ GA & \xrightarrow[Gf]{} & GB \end{array}$$

Figure 4: Naturality condition diagrammatically

This condition is known as *naturality*. If each t_A is an isomorphism, we say that t is a **natural isomorphism**:

$$t : F \xrightarrow{\cong} G$$

B Cartesian closed and monoidal closed categories

In this section we define *Cartesian closed categories* and *symmetric monoidal closed categories*. Let us begin with an informal view of the first. Informally, Cartesian closed categories possess special objects in the sense that they have mathematical properties mimicking those of $\mathbf{C}(A, B)$. In other words,, they can be seen as collections of arrows from A to B.

The formal definition of Cartesian closed category has some concepts that were not introduced before, such as *exponentials*, *evaluation* or *application*, and *currying*. So, a brief introduction to them shall be given.

Consider the category of **Set**. Let A, B, C be sets, and $f : A \times B \to C$ a function. The exponent B^A is the set of functions from A to B ($A \Rightarrow B$). The application morphism is defined as:

$$\begin{aligned} app : & B^A \times A \to B \\ & (f, a) = f(a) \end{aligned}$$

Furthermore, the application morphism has the following couniversal property [AT10]. For any $g : C \times A \to B$, exists a unique map

$$\begin{aligned} \lambda f : & C \to A^B \\ & (\lambda f)c \mapsto (b \mapsto f(c, b)) \end{aligned}$$

called the currying, such that $app \circ (\lambda f \times id) = f$.

For a more detailed view of this concepts see [AT10].

Now, the definition of a Cartesian closed category is as follows [AC98]:

Definition B.1. A category $\mathbf{C}$ is called Cartesian closed if it has:

- A terminal object
- Binary products

- For any objects A, B, B^A with $app : B^A \times A \to B$, such that the following equation holds: $app \circ (\lambda f \times id) = f$

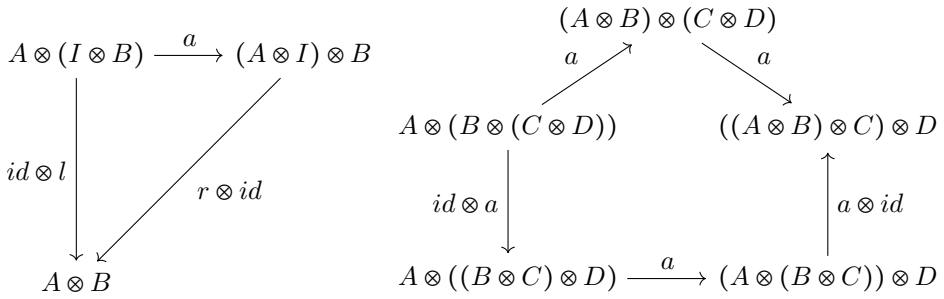
Let us follow a similar strategy for symmetric monoidal closed categories. A category $\mathbf{C}$ is closed if for any $A, B \in Obj(\mathbf{C})$ the set of morphisms from A to B is itself an object of $\mathbf{C}$. A monoidal category is a category that has the notion of a tensor functor. A symmetric monoidal category is a category in which the tensor is symmetric, *i.e.* $A \otimes B \cong B \otimes A$. Formally [AT10]:

Definition B.2. A *symmetric monoidal closed category* is a tuple $(\mathbf{C}, \otimes, I, a, l, r, s)$ where:

1. $\mathbf{C}$ is a category
2. $\otimes : \mathbf{C} \times \mathbf{C} \to \mathbf{C}$ is a functor (tensor)
3. I is a distinguished object of $\mathbf{C}$ (unit)
4. a, l, r are natural isomorphisms (structural isos) with components:

$$a_{A,B,C} : A \otimes (B \otimes C) \xrightarrow{\cong} (A \otimes B) \otimes C$$
$$l_A : I \otimes A \xrightarrow{\cong} A, \qquad r_A : A \otimes I \xrightarrow{\cong} A$$

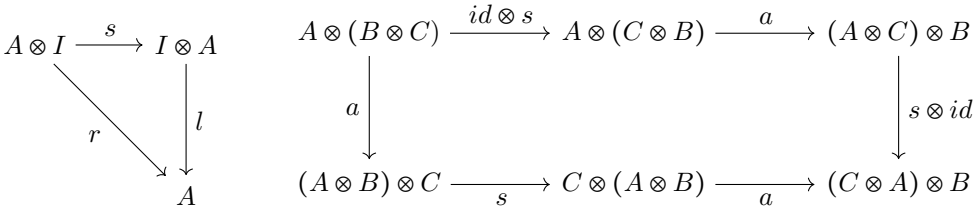
such that the following diagrams commute (the subscripts in the natural transformations are omitted for simplicity)



5. s is a natural isomorphism (symmetric),

$$s_{A,B} : A \otimes B \xrightarrow{\cong} B \otimes A$$

such that $s_{B,A} = s^{-1}_{A,B}$ and the following diagrams commute.



6. for all pairs A, B there is an object $A \multimap B$ and a morphism

$$app_{A,B} : (A \multimap B) \otimes A \to B$$

such that, for every morphism $f : C \otimes A \to B$, there is a *unique* morphism $\lambda(f) : C \to (A \multimap B)$ such that

$$app_{A,B} \circ (\lambda(f) \otimes id_A) = f$$

Note that we use the same notation *app* and $\lambda(f)$ from Cartesian closed categories, although they are different structures.

In sum:

- a monoidal category obeys points 1-4 of Definition B.2
- a symmetric monoidal category verifies points 1-5 of Definition B.2
- a symmetric monoidal closed category obeys all points of Definition B.2