UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

JULIANA KAIZER VIZZOTTO

Structuring General and Complete Quantum Computations in Haskell: The Arrows Approach

Thesis presented in partial fulfillment
of the requirements for the degree of
Doctor of Computer Science

Prof. Dr. Antônio Carlos Rocha Costa
Advisor

Prof. Dr. Amr A. Sabry
Co-advisor

Porto Alegre, July 2006

CIP – CATALOGING-IN-PUBLICATION

Vizzotto, Juliana Kaizer

Structuring General and Complete Quantum Computations in Haskell: The Arrows Approach / Juliana Kaizer Vizzotto. – Porto Alegre: PPGC da UFRGS, 2006.

128 f.: il.

Thesis (Ph.D.) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2006. Advisor: Antônio Carlos Rocha Costa; Co-advisor: Amr A. Sabry.

1. Quantum Programming Languages. 2. Haskell. 3. Density Matrices. 4. Monads. I. Rocha Costa, Antônio Carlos. II. Sabry, Amr A. III. Title.

“If quantum mechanics hasn’t profoundly shocked you,
you haven’t understood it yet.”
— Niels Bohr

ACKNOWLEDGMENTS

First of all I am very grateful to my advisor Antônio Carlos da Rocha Costa for introducing me to this *magic* field of research called *Quantum Computing*.

This thesis has a long trajectory and I have many people to thank in many different places. I am going to start thanking my first colleagues at PPGC/UFRGS, Bohrer, João, Júlio, Márcia Pasin, Mozart, Pilla, Pati, Tati, Vânia, Wives, and Zé. In particular, thanks to Mônica for her support and special friendship. I'd also like to thank to my second round colleagues at PPGC/UFRGS Fábio, Márcia, and Pablo. Many thanks to all people from the Computer Science Department at Indiana University, specially to Prof. Amr Sabry for the many helpful discussions and pointers, Prof. Daniel Friedman for his hospitality, and friends Eric, Marina, and Will, for their support and company in stargazing nights. My gratitude goes also to Jonathan Grattage, and Thorsten Altenkirch for our helpful quantum discussions. Finally, I would like to thank my actual and lovely colleagues at UCPel, Adenauer, André, Graça, Marilton, Pilla, and Renata. They had an important role in the conclusion of this work.

Thanks to my best friends Fernanda and Flávia for their encouragement and special friendship.

Thanks to my wonderful family. Specially, to my parents, Angelo Renato and Terezinha, and to my sister, Daniele, for their love and support.

Finally, I would like to express my special gratitude to my advisor Antônio Carlos da Rocha Costa and to my co-advisor Amr Sabry for their teaching, guidance, support, and for their belief in my ability to finish this task.

This research was supported by a grant from CNPq and another grant from CAPES, during the period I was visiting Prof. Amr Sabry at Indiana University.

CONTENTS

LIST OF FIGURES

ABSTRACT

Quantum computation can be understood as *transformation* of information encoded in the state of a *quantum* physical system. The basic idea behind quantum computation is to encode data using quantum bits (qubits). Differently from the classical bit, the qubit can be in a *superposition* of basic states leading to "quantum parallelism", which is an important characteristic of quantum computation since it can greatly increase the speed processing of algorithms. However, quantum data types are computationally very powerful not only due to superposition. There are other odd properties like *measurement* and *entangled*.

In this thesis we argue that a realistic model for quantum computations should be *general* with respect to measurements, and *complete* with respect to the information flow between the quantum and classical worlds. We thus explain and structure general and complete quantum programming in Haskell using well known constructions from classical semantics and programming languages, like *monads* and *arrows*. In more detail, this thesis focuses on the following contributions.

Monads and Arrows. Quantum parallelism, entanglement, and measurement certainly go beyond "pure" functional programming. We have shown that quantum parallelism can be modelled using a slightly generalisation of monads called *indexed monads*, or *Kleisli structures*. We have also build on this insight and showed that quantum measurement can be explained using a more radical generalisation of monads, the so-called *arrows*, more specifically, *indexed arrows*, which we define in this thesis. This result connects "generic" and "complete" quantum features to well-founded semantics constructions and programming languages.

Understanding of Interpretations of Quantum Mechanics as Computational Effects. In a thought experiment, Einsten, Podolsky, and Rosen demonstrate some counter-intuitive consequences of quantum mechanics. The basic idea is that two entangled particles appear to always communicate some information even when they are separated by arbitrarily large distances. There has been endless debate and papers on this topic, but it is interesting that, as proposed by Amr Sabry, this strangeness can be essentially modelled by assignments to global variables. We build on that, and model this strangeness using the general notions of computational effects embodied in monads and arrows.

Reasoning about Quantum Programs Using Algebraic Laws. We have developed a preliminary work to do equational reasoning about quantum algorithms written in a *pure* sublanguage of a functional quantum programming language, called QML.

Keywords: Quantum Programming Languages, Haskell, Density Matrices, Monads.

Estruturando Computações Quânticas Gerais e Completas em Haskell: Abordagem das Setas

RESUMO

Computação *quântica* pode ser entendida como *transformação* da informação codificada no estado de um sistema físico *quântico*. A idéia básica da computação quântica é codificar dados utilizando bits quânticos (qubits). Diferentemente do bit clássico, o qubit pode existir em uma *superposição* dos seus estados básicos permitindo o "paralelismo quântico", o qual é uma característica importante da computação quântica visto que pode aumentar consideravelmente a velocidade de processamento dos algoritmos. Entretanto, tipos de dados quânticos são bastante poderosos não somente por causa da superposição de estados. Existem outras propriedades ímpares como *medida* e *emaranhamento*.

Nesta tese, nós discutimos que um modelo realístico para computações quânticas deve ser *geral* com respeito a medidas, e *completo* com respeito a comunicação entre o mundo quântico e o mundo clássico. Nós, então, explicamos e estruturamos computações quânticas gerais e completas em Haskell utilizando construções conhecidas da área de semântica e linguagens de programação clássicas, como *mônadas* e *setas*. Em mais detalhes, esta tese se concentra nas seguintes contribuições.

Mônadas e Setas. Paralelismo quântico, emaranhamento e medida quântica certamente vão além do escopo de linguagens funcionais "puras". Nós mostramos que o paralelismo quântico pode ser modelado utilizando-se uma pequena generalização de mônadas, chamada *mônadas indexadas* ou *estruturas Kleisli*. Além disso, nós mostramos que a medida quântica pode ser explicada utilizando-se uma generalização mais radical de mônadas, as assim chamadas *setas*, mais especificamente, *setas indexadas*, as quais definimos nesta tese. Este resultado conecta características quânticas "genéricas" e "completas" à construções semânticas de linguagens de programação bem fundamentadas.

Entendendo as Interpretações da Mecânica Quântica como Efeitos Computacionais. Em um experimento hipotético, Einstein, Podolsky e Rosen demonstraram algumas conseqüências contra-intuitivas da mecânica quântica. A idéia básica é que duas partículas parecem sempre comunicar alguma informação mesmo estando separadas por uma distância arbitrariamente grande. Existe muito debate e muitos artigos sobre esse tópico, mas é interessante notar que, como proposto por Amr Sabry, essas características estranhas podem ser essencialmente modeladas por atribuições a variáveis globais. Baseados nesta idéia nós modelamos este comportamento estranho utilizando noções gerais de efeitos computacionais incorporados nas noções de mônadas e setas.

Provando Propriedades de Programas Quânticos Utilizando Leis Algébricas. Nós desenvolvemos um trabalho preliminar para fazer provas equacionais sobre algoritmos quânticos escritos em uma sublinguagem *pura* de uma linguagem de programação funcional quântica, chamada QML.

Palavras-chave: Linguagens de Programação Quântica, Haskell, Matrizes de Densidade.

1 INTRODUCTION

1.1 Quantum Computation

The first insight on quantum computation is generally accepted as Feynman's observation that simulation of quantum systems in classical computers is expensive (FEYNMAN, 1982), i.e., we need exponential time to simulate polynomial circuits. Three years later, Deutsch (DEUTSCH, 1985) explicitly asked whether it is possible to compute more efficiently on a quantum computer than on a classical computer. By addressing this question, he further extended the theory of quantum computation with the development of the quantum Turing machine. However, it was after Shor's quantum algorithm (SHOR, 1994) to factor an integer in polynomial time in the number of its digits, and its interplay with cryptography which has the potential to undermine many current cryptosystems, that quantum computing has become a fast growing research area. In 1996, Grover showed a fast quantum algorithm for database search (GROVER, 1996) evidencing another task that could also be made more efficient by the use of quantum computers. Lastly, we cannot forget to mention the substantial research that has been done on quantum cryptographic techniques based on the pioneer work by Bennet and Brassard (BENNETT; BRASSARD, 1984).

The basic idea behind quantum computation is to encode data using quantum bits. A quantum bit or *qubit* is a physical system which has two basic states, usually written in the Dirac notation $|0\rangle$ and $|1\rangle$. Differently from the classical bit, the qubit can be in a *superposition* of these two basic states written as $\alpha|0\rangle + \beta|1\rangle$, with $|\alpha|^2 + |\beta|^2 = 1$. Intuitively, one can think that a qubit can exist as a 0, a 1, or simultaneously as both 0 and 1, with a numerical coefficient which determines the probability of each state. Formally, a qubit can be modelled as a normalized vector in a two-dimensional Hilbert space, i.e., a complex vector space equipped with an inner product satisfying certain axioms.

The quantum superposition phenomena is responsible for the so called "quantum parallelism". To understand what this means consider a (boolean) function that takes a single bit x to a single bit $f(x)$. In a quantum computer we can apply the function f to both inputs at once, that is the function can act in a *superposition* of $|0\rangle$ and $|1\rangle$. This idea is used in the famous Deutsch's (DEUTSCH, 1985) algorithm, which was one of the first demonstrations that a quantum computer can solve problems more efficiently than a classical one.

We can perform a *measurement* operation projecting a quantum state like $\alpha|0\rangle + \beta|1\rangle$ onto the basis $|0\rangle$,$|1\rangle$. The outcome of the measurement is not deterministic and it is given by the probability amplitude, i.e., the probability that the state after the measurement is $|0\rangle$ is $|\alpha|^2$ and the probability that that the state is $|1\rangle$ is $|\beta|^2$. If the value of the qubit is initially unknown, than there is no way to determine α and β with that single measurement, as the measurement may *disturb* the state. But, *after* the measurement, the qubit is in a *known*

state; either $|0\rangle$ or $|1\rangle$.

The disturbance related to the measurement is also connected with another essential characteristic of quantum states: the *non cloning property* of quantum states (NIELSEN; CHUANG, 2000). If we could make a perfect copy of a qubit, we could measure the original without disturbing it in contradiction with the disturbance principle.

This issue is sometimes called the problem of *decoherence*. For example, consider a qubit that is in a coherent state. As soon as its measurable interacts with the environment it will decohere and fall into one of the two basic states. The decoherence is a stumbling block for quantum computers (BONE; CASTRO, 1997), and a semantically quite complicated issue to deal with for quantum programming languages.

Surprisingly, more recently there has been several proposals of different models of quantum computation based only on measurements. One example is the "1-way quantum computer" by Raussendorf and Briegel (RAUSSENDORF; BROWNE; BRIEGEL, 2001, 2003). In such a computer the computation starts which a *cluster state* (BRIEGEL; RAUSSENDORF, 2001) of certain size and uses only 1-qubit measurements. Other works also suggesting that measurements could be the actual driving force behind quantum computations are (NIELSEN, 2003; KASHEFI; PANANGADEN; DANOS, 2004; LEUNG, 2004; DANOS et al., 2005).

Quantum data types are computationally very powerful not only due to superposition (and measurements). Moreover, qubits can be in an *entangled* state. In such a state, two or more qubits have to be described with reference to each other, even though the individuals may be spatially separated. For instance, a state of two qubits is a vector of the tensor product (usually, represented by $\otimes$) of two Hilbert spaces. Some of these two qubit states can be written as the tensor product of its constituent parts like $|\phi_1\rangle \otimes |\phi_2\rangle$, but there are also the entangled states, which cannot be written as the tensor product of its parts. A well-known example of entangled state is the EPR pair $\alpha|00\rangle + \beta|11\rangle$ [1]. Quantum entanglement is the basis for emerging quantum algorithms, for instance these states have been used for experiments in quantum teleportation.

Besides measurements, *unitary transformations* are the only operations acting on qubits. For a general introduction to quantum computation, see e.g. (NIELSEN; CHUANG, 2000). We also recommend Preskill's excellent online notes (PRESKILL, 1999).

More abstractly, quantum computation can be organised into two main approaches: *classical control and quantum data*, and *quantum control and quantum data*. Essentially, the former follows the work by Knill (KNILL, 1996), where a quantum computer consists of a *quantum random access machine* (QRAM). In this model the programmer assumes the existence of predefined universal set of unitary operations. The quantum control approach follows the model of a quantum Turing machine introduced by Deutsch (DEUTSCH, 1985). Quantum control means that the control can also be in a superposition, allowing the programmer to define any physically realisable unitary operation.

1.2 Domains of Quantum Computations

We call *strict* or *pure* or *reversible* quantum computations the evolution of a quantum state by the means of unitary gates; measurements are not considered. The objects in

[1] The name of the vector "EPR" refers to the initials of Einstein, Podolsky, and Rosen who used such a vector in a thought experiment to demonstrate some strange consequences of quantum mechanics (EINSTEIN; PODOLSKY; ROSEN, 1935).

this pure domain are normalized vectors in a complex vector space, i.e., functions from a classical state space to complex numbers. The normalization condition is because measurements on those states have probabilistic outcomes related to the complex amplitude. Hence, we require that the sum of probabilities of all possible outcomes of a measurement add up to 1. The pure computations are unitary maps, these are linear isomorphisms which preserve the probabilistic interpretation of amplitudes.

Irreversible programs involve measurements because we cannot dispose of a quantum bit without measuring it, and leading to mixed states, i.e., probabilistic distribution of pure states. *Superoperators* (that is, completely positive maps, see (SELINGER, 2004)) acting on density matrices, a notation for mixed quantum states due to von Neumann, are the well accepted domain to interpret the general quantum operations involving measurements.

1.3 Quantum Programming Languages

The research area in quantum programming languages has been mainly stimulated from the fact that quantum information processing devices, like their classical counterparts, should be programmed in high level, structured and well-defined languages. We believe that high level quantum programming languages can improve our understanding of the power of quantum computation.

Even though, the implementations of quantum computers are still very limited, working with only a few qubits in physics laboratories, we believe this topic of research is very fruitful and it has been pointed in (GAY, 2006) that many criticisms are ill-founded, for several reasons:

1. It overlooks the progress which has been made in the practical implementation of quantum cryptography.

2. On one hand, the early work on the foundations of classical programming languages (that is, Alonzo Church's famous paper (CHURCH, 1936) presenting the lambda calculus, and the work by Alan Turing showing his Turing machine as a universal computing model (TURING, 1936)) has been dated several years before the development of practical and commercial computing devices in the 50's, and has inspired the design of many actual programming languages. On the other hand, nowadays, every computer scientist is familiar with the problems caused in software engineering by the widespread use of programming languages which do not have firm semantic foundation. Mainly, this is due to the fact that computing technologies have raced ahead of theoretical studies. From these two points of view, the work on quantum programming languages and its foundations before the hardware exists is, in some sense, a very good situation.

3. Lastly, it seems that the application of semantic, logical and specially category-theoretic techniques is providing new perspective on quantum theory itself. For instance the works by Abramsky, Duncan, and Coecke (ABRAMSKY; DUNCAN, 2004; ABRAMSKY; COECKE, 2004; COECKE, 2005).

Essentially, following Simon Gay's quantum programming languages survey (GAY, 2006) the design of quantum languages can be classified as: a) imperative languages, b) functional languages, and c) other paradigms. In this context, we can rearrange the languages in two branches: 1) those ones that follow Knill's *quantum random access machine* (KNILL, 1996) (QRAM), usually called by the slogan "classical control and

quantum data". In this model a quantum computer can be seen as a classical computer, the controller, with a quantum device, the quantum memory, attached to it. The classical controller has the ability to perform a previously defined set (ideally universal) of quantum operations, including state preparation, unitary transformation and measurement, on quantum registers. And 2) those ones in which control, as well as data, may be quantum. The quantum Turing machine (DEUTSCH, 1985), in which the entire machine state, including the tape, and the position of the head is assumed to be in quantum superposition, is an example of this model.

We summarize some of the main quantum programming languages that have been developed in Tables 1.1 and 1.2. For a complete survey see (GAY, 2006).

Table 1.1: Quantum programming languages with *classical* control.

	Imperative	Functional
Classical Control	QCL by Bernhard Ömer (ÖMER, 1998): the first real quantum programming language, with a syntax inspired by C.	Peter Selinger's influential quantum language (SELINGER, 2004). Combines high level classical structures with operations on quantum data. This language has a clear mathematical semantics in terms of *superoperators*.
	Betteli, Calarco and Serafini (BETTELLI; SERAFINI; CALARCO, 2003) define a combination of C++ with a collection of low-level primitives based on the QRAM model.	Selinger and Valiron's language (SELINGER; VALIRON, 2006) based on the work above by Selinger. This language is based on a call-by-value λ-calculus, and has an affine type system (no contraction).
	qGCL by Sandres and Zuliani (SANDERS; ZULIANI, 2000) which is based on a guarded command language.	Arrighi and Dowek (ARRIGHI; DOWEK, 2005) define a linear algebraic λ-calculus in which all functions are linear operators on vector spaces.

1.4 Monads and Arrows

The mathematical concept of monads (MACLANE, 1971) was introduced to computer science by Moggi (MOGGI, 1989) in the late 1980's as a way of structuring denotational semantics of programming languages. Several different language features, including nontermination, state, exceptions, continuations, and interaction can be viewed as monads. More recently, this construction has been internalised in the programming language Haskell as a tool to elegantly express computational effects within the context of a pure functional language.

Since the work of Moggi, several natural notions of computational effects were discovered which could only be expressed as generalisations of monads. Of particular importance to us is the generalisation of monads known as arrows (HUGHES, 2000) which

Table 1.2: Quantum programming languages with *quantum* control.

	Functional
Quantum Control	Andre van Tonder (TONDER, 2003, 2004) has proposed a quantum λ-calculus incorporating higher order quantum programs, but no measurements. He also suggests an equational theory for strict (higher order) computations, but shows neither completeness nor normalisation.
	QML (ALTENKIRCH; GRATTAGE, 2005) is a first order functional quantum programming language added with quantum data and control. QML has a quantum $\mathbf{if}^\circ$, which analyzes the data without measuring, and hence without changing the data..

is also internalised in the programming language Haskell.

1.5 Contributions of this thesis

The main objective of this thesis is to explain and structure quantum programming using well known constructions from classical semantics and programming languages. In more detail, this thesis focuses on the following subjects.

- **Monads and Arrows**. Quantum parallelism, entanglement, and measurement certainly go beyond "pure" functional programming. We have shown (Chapter 5) that quantum parallelism can be modelled using a slightly generalisation of monads called *indexed monads*, or *Kleisli structures* (ALTENKIRCH; REUS, 1999). We have also build on this insight and showed (Chapter 6) that quantum measurement can be explained using a more radical generalisation of monads called *arrows* (HUGHES, 2000), more specifically, *indexed arrows*, which we define on Section 3.2.4. This result connects "generic" (including measurement) and "complete" (including communication between quantum and classical data) quantum features to well-founded semantics constructions and programming languages (Chapters 7 and 8).

- **Understanding of Interpretations of Quantum Mechanics as Computational Effects**. In a thought experiment, Einsten, Podolsky, and Rosen (EINSTEIN; PODOLSKY; ROSEN, 1935) demonstrate some counter-intuitive consequences of quantum mechanics. The basic idea is that two entangled particles appear to always communicate some information even when they are separated by arbitrarily large distances. There has been endless debate and papers on this topic, but it is interesting that, as proposed by Amr Sabry (SABRY, 2003), this strangeness can be essentially modelled by assignments to global variables. We build on that, and model this strangeness using the general notions of computational effects embodied in monads and arrows.

- **Reasoning about Quantum Programs Using Algebraic Laws**. We have developed in Chapter 4 a preliminary work to do equational reasoning about quantum algorithms written in a small subset of a functional quantum programming language, called QML (ALTENKIRCH; GRATTAGE, 2005).

1.6 Plan

In Chapter 2 we present a brief review on quantum computation. In Chapter 3 we show indexed monads and indexed arrows. The categorical notions we use to structure quantum computations. Chapter 4 describes an equational theory for reasoning about programs written in a pure subset of QML (ALTENKIRCH; GRATTAGE, 2005), a quantum functional language. Additionally we proof soundness and completeness for the pure subset of the language. Chapter 5 describes a monadic approach for "pure" (without measurement) quantum programming in Haskell. In Chapter 6 after modelling density matrices and superoperators in Haskell, we structure this model for "general" quantum computations (including measurements) using a generalisation of monads called indexed arrows. In Chapter 7 we extend the approach for "complete" quantum computations (including communication between quantum and classical data). Chapter 8 presents an alternative model for general and complete quantum computations using explicit probability distribution of state vectors. Chapter 9 concludes.

1.7 Publications

Some of the work described in this thesis has been published:

- Juliana K. Vizzotto, Antônio Carlos da Rocha Costa and Amr Sabry. *Quantum Arrows*. 4th International Workshop on Quantum Programming Languages, July 2006. To appear in ENTCS. (VIZZOTTO; COSTA; SABRY, 2006)

- Juliana K. Vizzotto, Thorsten Altenkirch and Amr Sabry. Structuring Quantum Effects: Superoperators as Arrows. In *Journal of Mathematical Structures in Computer Science: special issue in quantum programming languages*. 2006. (VIZZOTTO; ALTENKIRCH; SABRY, 2006)

- Juliana K. Vizzotto and Antônio Carlos da Rocha Costa. *Concurrent Quantum Programming in Haskell*. In VII Congresso Brasileiro de Redes Neurais (2005). Sessão de Computação Quântica. (VIZZOTTO; COSTA, 2005)

- Thorsten Altenkirch, Jonathan Grattage, Juliana K. Vizzotto and Amr Sabry. *An Algebra of Pure Quantum Programming*. 3rd International Workshop on Quantum Programming Languages, July 2005. To appear in ENTCS. (ALTENKIRCH et al., 2005)

2 QUANTUM COMPUTATION

Quantum computation can be understood as *transformation* of information encoded in the state of a *quantum* physical system. Hence, we start the chapter describing the main laws which preview the behaviour of quantum mechanical systems.

In classical computation the indivisible unit of information is the bit: an object that can take one of the possible values $\{0, 1\}$. In this chapter we describe the *qubit*, the corresponding unit of quantum information, and how computation can be carried out over an *array* of qubits. We also discuss two characteristic of quantum states which are claimed to be essential ingredients for the power of quantum computation: quantum parallelism and entanglement. As an example, we show Deutsch's (DEUTSCH, 1985) algorithm, demonstrating a specific problem which can be solved more efficiently in a quantum computer than in a classical one. We also briefly discuss some quantum computer models. In this chapter we consider a background on linear algebra which is discussed in the Appendix A.

2.1 Axioms of Quantum Mechanics

Quantum theory is a mathematical model of the physical world. To characterize the model we need to specify how it will represent: states, observables, measurements, dynamics, and composite systems.

2.1.1 States

In quantum theory a physical state is represented as a *unit* vector living in a complex inner product vector space know as *Hilbert space*. We call such a vector a *ket* (see appendix A) and denote it by $|\alpha\rangle$. This state ket contains complete information about the physical state.

2.1.2 Observables

An observable is a property of a physical system that can be measured. In quantum mechanics, an observable can be represented by a Hermitian *operator*, A, acting in the vector space in question. Remember from Section A.5.4 that there are particular kets of importance, known as *eigenkets* of the operator A, denoted by

$$|a'\rangle, |a''\rangle, |a'''\rangle, \ldots$$

with the property

$$A|a'\rangle = a'|a'\rangle,\ A|a''\rangle = a''|a''\rangle, \ldots$$

where $a', a'', \ldots$ are just real numbers, called *eigenvalues* of the operator A.

2.1.3 Measurements

"A measurement always causes the system to jump into an eigenstate of the dynamical variable that is being measured" (P. A. M. Dirac).

Following Sakurai (SAKURAI, 1994) we may interpret Dirac's words above as follows: before a measurement of an observable A, the system is assumed to be represented by some linear combination

$$|\alpha\rangle = \sum_{a'} c_{a'}|a'\rangle = \sum_{a'} |a'\rangle\langle a'|\alpha\rangle.$$

When the measurement is performed, the system is "thrown into" one of the eigenstates, say $|a'\rangle$ of the observable A. In other words,

$$|\alpha\rangle \overset{A\ meas}{\longrightarrow} |a'\rangle.$$

Thus a *measurement usually changes the state*. The only exception is when the state is already in one of the eigenstates of the observable being measured, in which case

$$|a'\rangle \overset{A\ meas}{\longrightarrow} |a'\rangle$$

with certainty. When a measurement causes $|\alpha\rangle$ to change into $|a'\rangle$, it is said that A is measured to be a'. It is in this sense that the result of the measurement yields one of the eigenvalues of the observable being measured.

Given

$$|\alpha\rangle = \sum_{a'} c_{a'}|a'\rangle = \sum_{a'} |a'\rangle\langle a'|\alpha\rangle.$$

which is the state ket of a physical system before the measurement, we do not know in advance into which of the various $|a'\rangle$'s the system will be thrown as the result of the measurement. However, we do know that the *probability* for jumping into some particular $|a'\rangle$ is given by

$$|\langle a'|\alpha\rangle|^2$$

provided that $|\alpha\rangle$ is normalized.

This probabilistic interpretation for the squared inner product above is one of the fundamental postulates of quantum mechanics. Suppose the state ket is $|a'\rangle$ itself even before the measurement is made. Then, according to the postulate, the probability for getting a' - or more precisely, for being thrown into $|a'\rangle$ - as the result of the measurement is predicted to be 1, which is just what we expect.

There is also the notion of *seletive measurement*, or *filtration*. More generally, we consider a measurement process with a device that selects only one of the eigenkets of A, say $|a'\rangle$ and rejects all others. Mathematically, we can say that such a selective measurement amounts to applying the projection operator

$$\Lambda_{a'} \equiv |a'\rangle\langle a'|$$

to $|\alpha\rangle$:

$$\Lambda_{a'}|\alpha\rangle = |a'\rangle\langle a'|\alpha\rangle.$$

2.1.4 Dynamics

Time evolution of a *closed* quantum state is described by a *unitary transformation*. That is, the state $|\alpha\rangle$ of the system at time t_1 is related to the state $|\alpha'\rangle$ of the system at time t_2 by a unitary operator U:

$$|\alpha'\rangle = U|\alpha\rangle.$$

2.1.5 Composite Systems

The state space of a composite physical system is the *tensor product* (see definition A.4.2) of the state spaces of the component physical systems. Moreover, if we have systems numbered 1 through n, and system number i is prepared in the state $|\alpha_i\rangle$, then the joint state of the total system is $|\alpha_1\rangle \otimes |\alpha_2\rangle, \ldots, |\alpha_n\rangle$.

2.2 Qubits

The qubit is the simplest possible quantum system, that is, it is represented as a vector in a two-dimensional Hilbert space (i.e., in a complex vector space with inner product). Usually, the elements of an orthonormal basis in this space are called $|0\rangle$ and $|1\rangle$ in Dirac notation. Then a normalized vector can be represented as a linear combination of basic states:

$$|\alpha\rangle = a|0\rangle + b|1\rangle,\ |a|^2 + |b|^2 = 1$$

that can also be written as the column vector

$$\begin{bmatrix} a \\ b \end{bmatrix}$$

where a and $b \in \mathbb{C}$. Coefficients, like a and b, are called complex amplitudes.

It is this ability of the qubit of being in a *linear combination* of basic states, also often called *superposition*, that is responsible for the so called "quantum parallelism". To understand what this means consider a (boolean) function that takes a single bit x to a single bit $f(x)$. In a quantum computer we can apply the function f to both inputs at once, that is the function can be applied to a *superposition* of $|0\rangle$ and $|1\rangle$. This feature is used in an immediate way in Deutsch's algorithm (Section 2.5).

A quantum state of N qubits can be expressed as a vector in a space of dimension 2^N. A 2^N dimensional qubit space is given by the *tensor product* ($\otimes$) of N spaces of single qubits. For instance, an orthonormal basis for a quantum state of 2 qubits could be $\{|0\rangle, |1\rangle\} \otimes \{|0\rangle, |1\rangle\} = \{|0\rangle \otimes |0\rangle, |0\rangle \otimes |1\rangle, |1\rangle \otimes |0\rangle, |1\rangle \otimes |1\rangle\}$, usually written as $\{|00\rangle, |01\rangle, |10\rangle, |11\rangle\}$. The tensor product gives as a general 2 qubit state a linear combination of this four basic states, i.e.,

$$a_{00}|00\rangle + a_{01}|01\rangle + a_{10}|10\rangle + a_{11}|11\rangle = \sum_{i,j\in\{0,1\}} a_{ij}|i,j\rangle.$$

In general, a state of N quantum bits is a non-zero vector in a Hilbert space, which can be represented as the following formal linear combination:

$$\sum_{b_1,\ldots,b_n\in\{0,1\}} a_{b_1\ldots b_n}|b_1\ldots b_n\rangle, \text{ with } \sum_{b_1,\ldots,b_n\in\{0,1\}} |a_{b_1\ldots b_n}|^2 = 1.$$

2.3 Quantum Operations on Qubits

2.3.1 Measurements

The normalization condition in the qubit vector is required because measurements on those states have probabilistic outcomes related to the complex amplitude. Hence, the sum of probabilities of all possible outcomes of a measurement must add up to 1. More specifically, we can perform a measurement that projects the qubit $a|0\rangle + b|1\rangle$ onto the basis $\{|0\rangle, |1\rangle\}$. Then, the post measurement state will be $|0\rangle$ with probability $|a|^2$, or $|1\rangle$ with probability $|b|^2$. The measurement theory in quantum mechanics (see Section 2.1.3) says that the value *output* by a measurement is one of the eigenvalues of the *observable* being measured and that the state is collapsed to a corresponding eigenstate. However, in quantum computation one is often more interested in the *post measurement state* than in the real value (the eigenvalue) measured. So, in most times, if the post measurement state is $|0\rangle$ one would say that 0 was measured.

If the value of the qubit is initially unknown, than there is no way to determine a and b with one single measurement. But, *after* the measurement, the qubit is in a *know* state, either $|0\rangle$ or $|1\rangle$. In particular, if after a measurement the same measurement is performed again it will give the same answer as in the first time.

The situation is more complex if more than one qubit is involved. Consider a two-qubit system in the state $a_{00}|00\rangle + a_{01}|01\rangle + a_{10}|1,0\rangle + a_{11}|11\rangle$. If we measure the value of the first qubit, we obtain:

- 0 with probability $|a_{00}|^2 + |a_{01}|^2$, and the quantum state will collapse to

$$\frac{1}{\sqrt{|a_{00}|^2 + |a_{01}|^2}} (a_{00}|00\rangle + a_{01}|01\rangle), \text{ and}$$

- 1 with probability $|a_{10}|^2 + |a_{11}|^2$, and the quantum state will collapse to

$$\frac{1}{\sqrt{|a_{10}|^2 + |a_{11}|^2}} (a_{10}|10\rangle + a_{11}|11\rangle).$$

Note that at each step we normalize the states in such a way that the sum of the squares of the amplitudes of the new reached state is 1. A similar situation happens if we measure the second qubit.

In a general state of N quantum bits

$$\sum_{b_1,\ldots,b_n \in \{0,1\}} a_{b_1 \ldots b_n} |b_1 \ldots b_n\rangle, \text{ with } \sum_{b_1,\ldots,b_n \in \{0,1\}} |a_{b_1 \ldots b_n}|^2 = 1.$$

the probability to get $|b_1 \ldots b_n\rangle$ when measuring the system is $|a_{b_1 \ldots b_n}|^2$.

2.3.2 Unitary Transformations

The other kind of operations we can apply to qubits are *unitary transformations*, which can be represented by unitary matrices.

Suppose $|\alpha\rangle = a|0\rangle + b|1\rangle$, then we can perform a reversible transformation on that by the application of a 2×2 unitary S:

$$\begin{pmatrix} s_{11} & s_{12} \\ s_{21} & s_{22} \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix}$$

which is given by usual matrix multiplication. In general, a $2^n \times 2^n$ matrix acts on n quibt system. The rows and columns of the unitary transformations are labeled from left to right and top to botton as $00\ldots0, 00\ldots1$ to $11\ldots1$.

Usually, these unitary matrices are called quantum gates as they are used in *quantum circuits*. Some important quantum gates are:

$$\text{NOT or Pauli-X} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \quad \text{H or Hadamard} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$$

$$\text{S or Phase} = \begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix} \qquad \text{Z or Pauli-Z} = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

$$\text{CNOT or Controlled-Not} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

$$\text{Toffoli} = \left(\begin{array}{c|c} id_4 & 0 \\ \hline 0 & CNOT \end{array}\right)$$

The unary gate NOT is the quantum version of the boolean not mapping $a|0\rangle + b|1\rangle$ to $b|0\rangle + a|1\rangle$. The unary Hadamard is sometimes described as turning $|0\rangle$ into "halfway" between $|0\rangle$ and $|1\rangle$ (first column of H), and also $|1\rangle$ into "halfway" between $|0\rangle$ and $|1\rangle$ (second column of H). The Hadamard gate is one of the most useful gates, it is used when one wants to prepare a quantum state in a coherent superposition. The unary gates S and Z represent complex phase changes. The binary controlled gate CNOT applies the NOT gate to the second qubit if the first one is 1; if the first qubit is 0 it does nothing. Similarly, the Toffoli is a controlled-controlled NOT, which applies the NOT gate to the third qubit if the first and second qubits are 1. The Toffoli gate can be used to simulate NAND (initializing the third qubit to 1) and FANOUT (initializing the first and third qubits to 1 and 0, respectively) gates.

Quantum circuits, as their classical counterparts consist of *wires* and *logic gates*. The wires are used to carry information around the circuit, while the logic gates perform manipulations of the information. For instance consider the circuit in Figure 2.1, which is read from left to right, and from top to bottom, and computes the state

$$G_2 \circ (G_1 \otimes id)(x \otimes y).$$

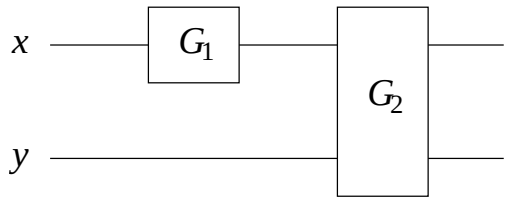


Figure 2.1: Quantum Circuit.

In general, there is a special notation for controlled gates. Instead of using simple boxes they are written using a filled circle in the control qubits. For instance see the circuit for controlled-NOT below:

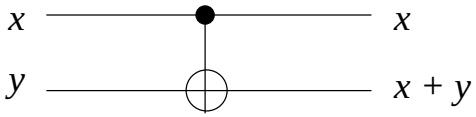


Figure 2.2: Controlled-NOT.

2.4 Characteristics of Quantum States

2.4.1 Entanglement

It is interesting to note that information is *non-local* in an array of qubits due to *entanglement*. A 2-qubit state $|\alpha\rangle$ is said to be *entangled* if it cannot be written as the tensor product of its constituent parts, i.e., as $|\alpha_1\rangle \otimes |\alpha_2\rangle$. For example, consider the *Bell state* or EPR pair:

$$|\alpha\rangle = \frac{|00\rangle + |11\rangle}{\sqrt{2}}.$$

One cannot find $|\alpha_1\rangle$ and $|\alpha_2\rangle$ such that $|\alpha_1\rangle \otimes |\alpha_2\rangle = |\alpha\rangle$. One may see that $|\alpha_1\rangle = a|0\rangle + b|1\rangle$ and $|\alpha_2\rangle = c|0\rangle + d|1\rangle$ such that

$$|\alpha\rangle = (a|0\rangle + b|1\rangle) \otimes (c|0\rangle + d|1\rangle) = ac|00\rangle + ad|01\rangle + bc|10\rangle + bd|11\rangle.$$

But, that is impossible to get $|00\rangle + |11\rangle$ from above, because we would need to have $ac \neq 0$ and $bd \neq 0$, with $ad = 0$. Indeed information is non-local in the Bell state; the measurement outcomes are *correlated* (NIELSEN; CHUANG, 2000). More specifically, the Bell state has the property that upon measuring the first qubit, one obtains two possible results: 0 with probability $1/2$, leaving the post measurement state $|00\rangle$, and 1 with probability $1/2$, leaving the post measurement state $|11\rangle$. As a result, a measurement of the second qubit always gives the same result as the measurement of the first qubit.

2.4.2 Copying a Qubit

Consider the task of copying a classical bit. This may be done using a classical CNOT gate, which takes an unknown bit x to copy and a "scratchpad" bit initialized to 0, as illustrated in Figure 2.3. The output is two bits, both in the same state x.

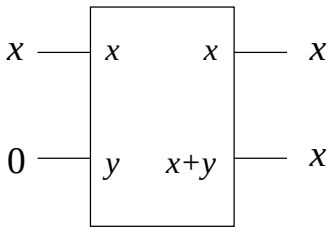


Figure 2.3: Classical circuit to *copy*.

Suppose we want to copy a qubit in the unknown state $|\alpha\rangle = a|0\rangle + b|1\rangle$ in the same manner by using a quantum CNOT gate. The input state of two qubits can be written as

$$[a|0\rangle + b|1\rangle]|0\rangle = a|00\rangle + b|10\rangle.$$

The function of CNOT is to negate the second qubit when the first qubit is 1. Thus, the output is simply $a|00\rangle + b|11\rangle$. In the case where $a = 0$ or $b = 0$ the circuit indeed successfully copies $|\alpha\rangle$. However, for a general state $|\alpha\rangle$ we see that the result of the copy operation would be

$$|\alpha\rangle\ |\alpha\rangle = a^2|00\rangle + ab|01\rangle + ab|10\rangle + b^2|11\rangle$$

which is not a linear operation. In fact, it turns out to be *impossible* to make a copy of an unknown quantum state. This is called the *non-cloning* (NIELSEN; CHUANG, 2000) property of quantum states.

This property can also be explained by the disturbance related to the measurement. If we could make a perfect copy of qubit, we could measure the original without disturbing it in contradiction with the disturbance principle.

The CNOT gate applied to a qubit and "scratchpad" bit initialized to 0 is sometimes refereed as "sharing" and it is used as the semantics of duplicating variables in quantum programming languages, see for instance (ALTENKIRCH; GRATTAGE, 2005; ARRIGHI; DOWEK, 2005).

2.4.3 Discarding a Qubit

Consider any composite quantum state living in a composite vector space and recall that such a vector space is formed by the *tensor product* of its component spaces. Any quantum state living in this composite space is *global* and *possible entangled*. Hence, any quantum operation is considered *global*.

Now suppose we just have *ignored* some specific part of the global quantum state. But by just *ignoring* part of the state we maintain the global entanglement, and any operation acting on the ignored part may affect other parts still entangled with it. The general way of destroying entanglement and taking apart a specific part of a quantum state is by performing a measurement on that.

More specifically, suppose we want to discard the left qubit of the EPR pair:

$$|\alpha\rangle = \frac{|00\rangle + |11\rangle}{\sqrt{2}},$$

which means to measure it. Hence, the state $|\alpha\rangle$ after discarding the left qubit will be $|0\rangle$ with probability $1/2$ or $|1\rangle$ with probability $1/2$.

This *decoherence* caused by discarding is, thus, a very trick situation for quantum programming (for a concrete discussion see Section 6.5).

2.5 The Deutsch Algorithm

Deutsch's (DEUTSCH, 1985) algorithm is used to find out whether a boolean function f is balanced or constant. Here we show a version of the algorithm presented in (GAY, 2006). Classically, to solve the problem we must evaluate $f(0)$ and $f(1)$ and compare the results. The appeal of the algorithm is that a quantum computer can answer the question with only one evaluation of f.

Firstly, we need to build a quantum version of f, that is a *unitary transformation* F, which performs the same compution as f. Note that in general f need not to be reversible. However, it is possible to construct a unitary transformation F on two qubits such that

$$F|x\rangle|y\rangle = |x\rangle|y \oplus f(x)\rangle$$

We therefore can assume that this is a quantum version of f.

The trick now is to apply F to the state

$$|+\rangle|-\rangle = \frac{1}{2}(|00\rangle - |01\rangle + |10\rangle - |11\rangle)$$

where

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

Note that this state can be created by applying $H \otimes H$ to $|01\rangle$.

Then we need to do some calculation in order to be able to express the result in terms of the unknown function f.

From the definition of F we have

$$\begin{array}{rcl} F|x\rangle|0\rangle & = & |x\rangle|f(x)\rangle \\ F|x\rangle|1\rangle & = & |x\rangle|1 \oplus f(x)\rangle. \end{array}$$

Combining this equations to calculate $F|x\rangle|-\rangle$, we see that if $f(x) = 0$ then $F|x\rangle|-\rangle = \frac{1}{2}|x\rangle(|0\rangle - |1\rangle)$, and if $f(x) = 1$ then $F|x\rangle|-\rangle = \frac{1}{2}|x\rangle(|1\rangle - |0\rangle)$. Hence

$$\begin{array}{rcl} F|x\rangle|-\rangle & = & \frac{(-1)^{f(x)}}{\sqrt{2}}|x\rangle(|0\rangle - |1\rangle) \\ & = & (-1)^{f(x)}|x\rangle|-\rangle. \end{array}$$

Thus

$$\begin{array}{rcl} F|+\rangle|-\rangle & = & \frac{1}{2}(F|0\rangle|-\rangle + F|1\rangle|-\rangle \\ & = & \begin{cases} \stackrel{+}{-}|+\rangle|-\rangle & \text{if } f(0) = f(1) \\ \stackrel{+}{-}|-\rangle|-\rangle & \text{if } f(0) \neq f(1) \end{cases} \end{array}$$

and the information about whether or not f is constant has been concentrated into the first qubit.

From that $H|+\rangle = |0\rangle$ and $H|-\rangle = |1\rangle$, then applying H to the first qubit we get

$$\begin{array}{ll} \stackrel{+}{-}|0\rangle|-\rangle & \text{if } f(0) = f(1) \\ \stackrel{+}{-}|1\rangle|-\rangle & \text{if } f(0) \neq f(1) \end{array}$$

2.6 Quantum Computer Models

It is useful to keep in mind a hypothetical hardware device on which one can execute quantum algorithms.

2.6.1 Quantum Computer Models with Classical Control

One of the first proposals for quantum hardware devices was given by Knill (KNILL, 1996). In this model a practical quantum computer will take place on a QRAM (quantum random access machine), which consists of a general-purpose classical computer *controlling* a special quantum hardware device which provides a bank of individually addressable quantum bits. The classical device acts on the QRAM by sending to it a sequence of commands to perform initializations (setting a qubit to $|0\rangle$ or $|1\rangle$), built-in unitary operations and measurements.

2.6.2 Quantum Computer Models with Quantum Control

In such models, *control*, as well a data, may be quantum. The quantum Turing machine (DEUTSCH, 1985), in which the entire machine state, including the tape, and the the position of the head is assumed to be in quantum superposition, is an example of this model. The QML: quantum data and control (ALTENKIRCH; GRATTAGE, 2005b) language encompass both data and control quantum structures. In quantum computers with classical control, quantum data can only me processed using combinators corresponding to quantum circuits or by measurements. In contrast, QML has a quantum $\mathbf{if}^\circ$, which analyzes the data without measuring, and hence without changing the data. There is no need for a finite set of built-in unitary operations. For instance, the Hadamard gate can be written using the quantum control $\mathbf{if}^\circ$ as follows:

$$had\ x = \mathbf{if}^\circ\ x\ \mathbf{then}\ ((-1) * true + false)\ \mathbf{else}\ (true + false)$$

where $(true + false)$ represents an equal superposition of $true$ and $false$.

2.7 Summary

In this Chapter we reviewed basic principles of quantum mechanics and presented the main concepts of quantum programming. Essentially, in quantum programming one codes the state of a system using a quantum state, and then transforms it by means of unitary transformations and measurements. Also, one can combine classical control structures with quantum operations.

There are many reasons for using quantum mechanical devices for doing computation. First, consider a technological reason. The revolution in semiconductor technology has lead to a great effort into reducing the size and costs of binary bits. Nowadays a flat microchip with a surface area of order $1cm^2$ can hold of the order of 108 bits. The small size of these memory chips has also had the effect of speeding up the rate at which computers can run. Basicaly, this is because the electromagnetic signal has less distance to travel between components. Hence a motivation for imagining a "quantum computer" is to push these improvements in technology to their physical limit. The smallest device one can imagine, that can exist in two states, is a single quantum particle (the electron spin, for instance).

Second, as proved by Shor's (SHOR, 1994) factorization quantum algorithm, quantum data types, which feature *superposition* and *entangled*, can greatly increase the speed of computations.

3 INDEXED MONADS AND INDEXED ARROWS

The mathematical concept of monads (MACLANE, 1971) was introduced to computer science by Moggi (MOGGI, 1989) in the late 1980's as a way of structuring denotational semantics of programming languages. Several different language features, including nontermination, state, exceptions, continuations, and interaction can be viewed as monads. More recently, this construction has been internalised in the programming language Haskell as a tool to elegantly express computational effects within the context of a pure functional language.

Since the work of Moggi, several natural notions of computational effects were discovered which could only be expressed as generalisations of monads. Of particular importance to us is the generalisation of monads known as arrows (HUGHES, 2000) which is also internalised in the programming language Haskell.

In this Chapter we review these two concepts in the context of the programming language Haskell, as well we briefly discuss a small variation of these notions, which we call *indexed* monads and *indexed* arrows. Those are the right notions needed to structure quantum computations in Haskell.

3.1 Monads

A monad is a concept from category theory which is used in Computer Science for formulating definitions and structuring *notions of computations* in programming languages. Essentially, one can understand a notion of computation as a qualitative description of certain (possibly non-fucntional) program features such as side-effects, exceptions, partial and nondeterministic computations, etc. In this context, a *program*, which features notions of computations, can be viewed as a *function from values to computations*. For instance a program with exceptions can be viewed as a function that takes a value and return a *computation* that may succeed or may fail.

More precisely, one can consider a value category $\mathcal{C}$, as a model for functions, and build on top of that, notions of computation via an operator (functor) T acting on objects of $\mathcal{C}$ - i.e., T maps an object B from $\mathcal{C}$, viewed as the *set of values of type* τ, to an object TB corresponding to *computations of type* τ. Then a program which takes an input of type A, and after performing certain computation returns a value of type B, can be identified with a morphism from A to TB in $\mathcal{C}$ (MOGGI, 1991).

This reasoning about *computations* can be intuitively organized by Kleisli triples, leading to Kleisli categories as a model for programs.

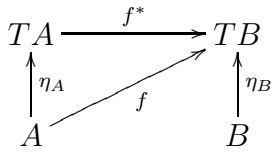
Definition 3.1.1 (Kleisli Triple) *A Kleisli triple over a category $\mathcal{C}$ is a triple $(T, \eta, \_^*)$, where*

- $T : Obj(\mathcal{C}) \rightarrow Obj(\mathcal{C})$,
- $\eta_A : A \rightarrow TA$,
- $f^* : TA \rightarrow TB$ *for* $f : A \rightarrow TB$,

and the following equations hold:

- $\eta_A^* = id_{TA}$
- $\eta_A; f^* = f$
- $f^*; g^* = (f; g^*)^*$

Intuitively η_A is the *inclusion* of values into computations and f^* is the *extension* of a function from values to computations to a function from computations to computations, which first evaluates the operand computation and then applies f to the resulting value:



In other words, this explicitly implies the existence of a *functor* T, of *complex objects*, *for all values* in $\mathcal{C}$, such that all computations over those complex objects are defined in terms of functions from values to complex objects, respectively. We call a complex object TA an *effect* involving A.

The axioms for Kleisli triples amount exactly to say that programs form a category, the *Kleisli category* $\mathcal{C}_T$, where the set $\mathcal{C}_T(A, B)$ of morphisms from A to B is $\mathcal{C}(A, TB)$, the identity over A is η_A, and composition of f followed by g is $f; g^*$ (MOGGI, 1991). Intuitively, $f; g^*$ takes a value a and applies f to produce a computation fa, then it *evaluates/executes* the computation fa to get a value b, and finally it applies g to b to produce the final computation.

Note that we are talking about monads but we have defined Kleisli structures. Indeed, there is a one-to-one correspondence between these two notions. However the definition of a monad is given in terms of functors and natural transformations, and although more elegant it is more abstract. We choose to present here the representation as a Kleisli structure because the Haskell's implementation of a monad mirrors the Kleisli version.

3.1.1 Monads in Haskell

Basically, monads are used in Haskell as a way to carry out computations with effects in the context of a pure functional language. A monad is represented in Haskell using a type constructor for computations m and two functions:

$return \in forall\ a.a \rightarrow m\ a$

$\gg= \in forall\ a\ b.m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b$

The operation $\gg=$ (pronounced "bind") specifies how to sequence computations and $return$ specifies how to lift values to computations. The function $return$ is exactly the η_A requirement of the Kleisli triple above. To understand the type of bind, one just need to consider the arguments for $\_^* : (A \rightarrow TA) \rightarrow TA \rightarrow TB$ in the inverse order. Observe the quantifier $forall$ preceding the types of the functions. This is to emphasize that m represents an *endofunctor* over the category of values, as the definition of T formally presented in definition 3.1.1.

To construe a proper monad, the $return$ and $\gg\!=$ functions must work together according to the three monad laws:

$$\begin{array}{rcl} m \gg\!= return & = & m \\ (return\ x) \gg\!= f & = & f x \\ (m \gg\!= f) \gg\!= g & = & m \gg\!= (\lambda x \rightarrow f x \gg\!= g) \end{array}$$

Note that the equations are the same as above just rephrased in terms of the Haskell's operations.

A simple example of a monad in Haskell is the $Maybe$ type:

$\mathbf{data}\ Maybe\ a = Nothing \mid Just\ a$

which represents the type of computations over a type a which may fail to return a result. It is similar to the idea of exceptions in programming languages: a computation may return a value ($Just\ a$), or fail returning $Nothing$. Then $returnM$ lifts simple values to computations which may fail, and $bindM$ combines computations of that type together:

$returnM \in forall\ a.a \rightarrow Maybe\ a$
$returnM\ a = Just\ a$
$(bindM) \in forall\ a\ b.Maybe\ a \rightarrow (a \rightarrow Maybe\ b) \rightarrow Maybe\ b$
$Nothing`bindM'\ f = Nothing$
$(Just\ x)`bindM'\ f = f\ x$

that is, the combined computation should yield $Nothing$ whenever either of the computations yield $Nothing$ and the combined computation should yield a computation of type b applied to the result of the computation $Maybe\ a$ when both computations succeed.

3.1.2 Monads in Haskell with Type Classes

Haskell' type classes allow the user to declare the *names* and *signatures* of the *class operations*. For instance, in Haskell, there is a standard $Monad$ class that defines the names and signatures of the two monad functions:

$\mathbf{class}\ Monad\ m\ \mathbf{where}$
$return \in forall\ a.a \rightarrow m\ a$
$(\gg\!=) \in forall\ a\ b.m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b$

This declares that a type m belongs to the class $Monad$ if there are two operations $return$ of type $a \rightarrow m\ a$ and $\gg\!=$ of type $m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b$. The definition of the Monad class above showed only the minimal complete definition. The full definition of the Monad class in Haskell actually includes two additional functions:

$fail \in a \rightarrow m\ a$
$\gg \in m\ a \rightarrow m\ b \rightarrow m\ b$

where, the default implementation of the fail function is:

$fail\ s = error\ s$

We only need to change this if we want to provide different behavior for failure or to incorporate failure into the computational strategy of our monad. The Maybe monad, for instance, defines fail as:

$fail\ \_ = Nothing$

so that $fail$ returns an instance of the $Maybe$ monad with meaningful behavior when it is bound with other functions in the Maybe monad.

The $fail$ function is not a required part of the mathematical definition of a monad, but it is included in the standard Monad class definition because of the role it plays in Haskell's do notation as explained below.

The $\gg$ function is a convenience operator that is used to bind a monadic computation that does not require input from the previous computation in the sequence. It is defined in terms of $\gg\!=$:

$$(\gg) \in m\ a \to m\ b \to m\ b$$
$$m \gg k = m \gg\!= (\lambda\_ \to k)$$

Then we can define which types are instances of which class, and provide definitions of the overloaded operations associated with a class, for example:

$$\begin{aligned}&\textbf{instance}\ \mathit{Monad}\ \mathit{Maybe}\ \textbf{where}\\ &\quad \mathit{return} = \mathit{returnM}\\ &\quad \gg\!= \ = \mathit{bindM}\\ &\quad \mathit{fail}\ \ = \mathit{Nothing}\end{aligned}$$

declares that type Maybe belongs to class Monad, and that the implementation of the two functions on Maybe type is given as in section above, and fail is $\mathit{Nothing}$.

It is not strictly necessary to make our monad instances of the Monad class, but Haskell has special support for Monad instances built into the language, called **do**-notation, which allow us to write cleaner and more elegant code. Basically, using the **do** notation we can write monadic computations in a pseudo-imperative style with named variables. The result of a monadic computation can be *assigned* to a variable using a left arrow $\leftarrow$ operator. Then using that variable in a subsequent monadic computation automatically performs the binding. The type of the expression to the right of the arrow is a monadic type $m\ a$. A traditional example would be to define division using the Maybe type:

$$\begin{aligned}&(//) \in \mathit{Maybe}\ \mathit{Float} \to \mathit{Maybe}\ \mathit{Float} \to \mathit{Maybe}\ \mathit{Float}\\ &x\ //\ y = \textbf{do}\ a \leftarrow x\\ &\qquad\qquad\ \ b \leftarrow y\\ &\qquad\qquad\ \ \textbf{if}\ b \equiv 0\ \textbf{then}\ \mathit{Nothing}\ \textbf{else}\ \mathit{return}\ (a\ /\ b)\end{aligned}$$

which is equivalent to the following awkward expression:

$$x\ //\ y = x \gg\!= \lambda a.y \gg\!= \lambda b.\textbf{if}\ b \equiv 0\ \textbf{then}\ \mathit{Nothing}\ \textbf{else}\ \mathit{return}\ (a\ /\ b)$$

Do-notation uses the following identies to translate **do**-expressions to respective monadic expressions with return, $\gg\!=$, and $\gg$:

$$\begin{aligned}&\textbf{do}\ \{p \leftarrow e; s\} = e \gg\!= \lambda p \to \textbf{do}\ \{s\}\\ &\textbf{do}\ \{e; s\}\qquad = e \gg \textbf{do}\ \{s\}\\ &\textbf{do}\ \{e\}\qquad\ \ = e\end{aligned}$$

The functions $\gg$, $\gg\!=$ are the functions in the Monad class. The fail function is called whenever a pattern matching failure occurs in a **do** block.

3.1.3 Monad Transformers

Consider we want to merge two monads, that is, we want to build a computation with two different kinds of effects, for instance computations which feature exceptions (Maybe type) and state passing. This can be designed systematically, using *monad transformers* (SHENG LIANG; JONES, 1995). A monad transformer is a monad parameterised on another monad, such that computations over the parameter monad can be *lifted* to computations over the new one.

For example, the Maybe monad above can be generalised to a monad transformer:

$$\begin{aligned}&\textbf{newtype}\ \mathit{MaybeMonadT}\ m\ a = \mathit{MT}\ (m\ (\mathit{Maybe}\ a))\\ &\mathit{unMT}\ (\mathit{MT}\ c) = c\end{aligned}$$

In general, the monad operators on the new type must be defined in terms of the monad operators in the parameter type:

$$\begin{array}{l}\textbf{instance}\ Monad\ m \Rightarrow Monad\ (MaybeMonadT\ m)\ \textbf{where}\\ \quad return\ a = MT\ (return\ (Just\ a))\\ \quad x \gg\!= f = MT\ (\textbf{do}\ a \leftarrow unMT\ x\\ \qquad\qquad\qquad\qquad \textbf{case}\ a\ \textbf{of}\\ \qquad\qquad\qquad\qquad\qquad Nothing \rightarrow return\ Nothing\\ \qquad\qquad\qquad\qquad\qquad Just\ a \rightarrow unMT\ (f\ a))\end{array}$$

Lifting of computations is defined by

$$\begin{array}{l}liftMaybe \in Monad\ m \Rightarrow m\ a \rightarrow MaybeMonadT\ m\ a\\ liftMaybe\ x = MT\ (x \gg\!= \lambda a \rightarrow return\ (Just\ a))\end{array}$$

3.1.4 Indexed Monads

In the definition of a Kleisli triple, the function T is an endofunctor on $\mathcal{C}$. Intuitively, this is the reason for the universal quantifier before the definitions of $return$ and $\gg\!=$ in Section 3.1.1, that is, the monadic constructor acts over all objects in the category of values $\mathcal{C}$.

However, sometimes we want to *select* some objects from $\mathcal{C}$ to apply the constructor T. This notion is slightly more general than Kleisli triples, and it is captured by the definition of *Kleisli structure* (ALTENKIRCH; REUS, 1999). Basically, for Kleisli structures, the function T does not need be an endofunctor on $\mathcal{C}$. We can select some objects from $\mathcal{C}$ to apply the constructor.

Definition 3.1.2 *A Kleisli structure* $(I, F, G, \eta^{F,G}, \_^{*F,G})$ *on a category* $\mathcal{C}$ *is given by:*

- *an index set* $I \in Set$*,*
- *families of objects indexed by* I*:* $F, G : I \rightarrow Obj(\mathcal{C})$*,*
- *a family of morphisms indexed by* $i \in I$*:* $\eta_i^{F,G} : F(i) \rightarrow G(i)$*,*
- *a family of functions indexed by* $i, j \in I$*:*

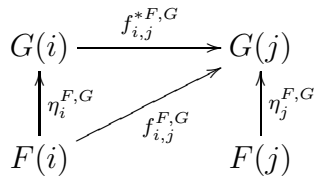
$$f_{i,j}^{*F,G} : G(i) \rightarrow G(j)$$

for $f_{i,j}^{F,G} \in F(i) \rightarrow G(j)$*.*

which are subject to the following equations:

1. $\eta_{i,i}^{*F,G} = id_{G(i)}$
2. $\eta_i^{F,G}; f_{i,j}^{*F,G} = f_{i,j}^{F,G}$ *where* $f_{i,j}^{F,G} : F(i) \rightarrow G(j)$*.*
3. $f_{i,j}^{*F,G}; g_{j,k}^{*F,G} = (f_{i,j}^{F,G}; g_{j,k}^{*F,G})_{i,k}^{*F,G}$ *where* $f_{i,j}^{F,G} : F(i) \rightarrow G(j)$*,* $g_{j,k}^{F,G} : F(j) \rightarrow G(k)$*.*

Note that Kleisli triples are a special case of Kleisli structures where $I = Obj(\mathcal{C})$ and F is the identity.



Now, the definitions of $return$ and $\gg=$ in Haskell would be rephrased as:

$return \in forall\ a.F\ (a) \Rightarrow a \rightarrow m\ a$

$\gg= \in forall\ a\ b.(F\ (a), F\ (b)) \Rightarrow m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b$

That is, for all a for which $F\ (a)$ holds we can apply the constructor m, and for all a and b for which $F\ (a)$ and $F\ (b)$ hold we can apply $\gg=$. In terms of type classes we would like to rewrite the $Monad$ class indexed in such a way:

$\mathbf{class}\ IMonad\ m\ \mathbf{where}$

$return \in F\ a \Rightarrow a \rightarrow m\ a$

$(\gg=) \in (F\ a, F\ b) \Rightarrow m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b$

Fortunately, in the new version of GHC (Glasgow Haskell Compiler - 6.5), the types for $return$, $\gg=$, and $\gg$ used in the **do**-notation may be overloaded to have our proper types. Therefore, we can define our *indexed monads* and still use **do**-notation for that type.

3.2 Arrows

To handle situations where monads are inapplicable, Hughes (HUGHES, 2000) introduced a new type class generalising monads, called *arrows*. Indeed, in addition to defining a notion of procedure which may perform computational effects, arrows may have a static component, or may accept more than one input.

Arrows were first introduced as an abstract interface for Swierstra and Duponcheel's parsing library (SWIERSTRA; DUPONCHEEL, 1996), which could not be modelled using monads. Essentially, they defined an efficient parsing library in the sense that the space leaks, caused in parsing grammars that define non-terminals via alternatives, are substantially reduced. The solution they proposed was to include a static component to the parser with some information about the tokens which could be accepted as the first in the input. Unfortunatelly, they coudn't define $\gg= \in Parser\ s\ a \rightarrow (a \rightarrow Parser\ s\ b) \rightarrow Parser\ s\ b$ using this representation (here s stands to the *static component*). The problem is that the static properties of the resulting $Parser\ s\ b$ depend on the static properties of *both* the first and the second arguments. Yet in the definition of $\gg=$, while we have acces to the static properties of the first argument, we cannot obtain the static properties of the second one without applying it to a value of type a.

Just as we think of a monadic type $m\ a$ as representing a *computation* delivering an a, so we think of an arrow type $a\ b\ c$ as representing a computation with input of type b delivering a c. Arrows make the dependence on input explicit.

Formally, $arrows$ give rise to *Freyd-categories* to model notions of computations. Here we present a simplified version of Freyd-categories as defined in (PATERSON, 2001), which is equivalent to the definition of Power and Robinson (POWER; ROBINSON, 1997).

Definition 3.2.1 (Freyd-category) *A Freyd-category is a structure* $(\mathcal{V}, \mathcal{C}, inc, \ltimes)$*, where:*

- *a category* $\mathcal{V}$ *with finite products (the value category),*
- *a category* $\mathcal{C}$ *with the same objects as* $\mathcal{V}$ *(the computation category),*
- *a functor* $inc :: \mathcal{V} \rightarrow \mathcal{C}$ *that is the identity on objects,*

- *a functor* $\ltimes :: \mathcal{C} \times \mathcal{V} \rightarrow \mathcal{C}$ *such that*

$$inc\ x \ltimes y = inc(x \times y)$$

and the following natural isomorphisms in $\mathcal{V}$

$$\begin{array}{rcl} assoc_{\times} & : & (A \times B) \times C \cong A \times (B \times C) \\ unitr_{\times} & : & A \times 1 \cong A \end{array}$$

extend to natural isomorphism in $\mathcal{C}$*:*

$$\begin{array}{rcl} inc\ assoc_{\times} & : & (A \ltimes B) \ltimes C \cong A \ltimes (B \times C) \\ inc\ unitr_{\times} & : & A \ltimes 1 \cong A \end{array}$$

The object preserving functor inc corresponds to the *lift* of functions from values to values to functions from computations to computations. Intuitively, the functor $\ltimes$ corresponds to say that we can always augment the state space of functions from computations to computations by applying a *program* that does nothing to the extra computations.

$$\begin{array}{ccc} \mathcal{C} & & \mathcal{C} \\ \uparrow{\scriptstyle inc} & & \uparrow{\scriptstyle \ltimes} \\ \mathcal{V} & & \mathcal{C} \times \mathcal{V} \end{array}$$

The last two axioms correspond to the naturality requeriments for the category.

3.2.1 Arrows in Haskell

In Haskell, the arrow interface is defined using the following class declaration:

$\textbf{class}\ Arrow\ a\ \textbf{where}$
$arr \quad \in forall\ b\ c.(b \rightarrow c) \rightarrow a\ b\ c$
$(\ggg) \in forall\ b\ c\ d.a\ b\ c \rightarrow a\ c\ d \rightarrow a\ b\ d$
$first \ \in forall\ b\ c\ d.a\ b\ c \rightarrow a\ (b,d)\ (c,d)$

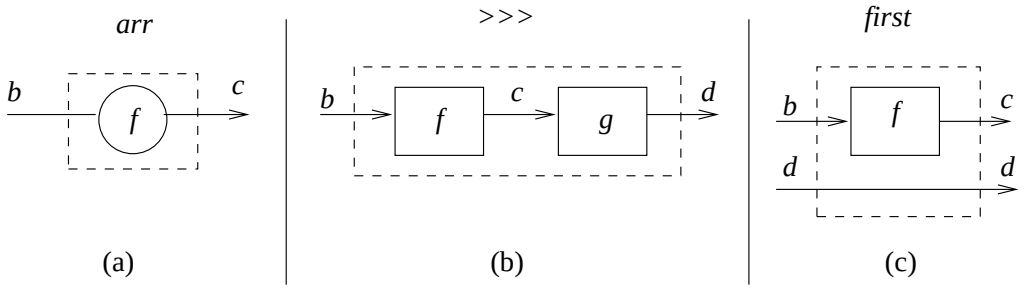
In other words, to be an arrow, a type a must support the three operations arr, $\ggg$, and *first* with the given types. Mirroring the naturality axioms in definition $3.2.1$, these operations must satisfy the following equations:

$$\begin{array}{rcl} arr\ id \ggg f & = & f \\ f \ggg arr\ id & = & f \\ (f \ggg g) \ggg h & = & f \ggg (g \ggg h) \\ arr\ (g\ .\ f) & = & arr\ f \ggg arr\ g \\ first\ (arr\ f) & = & arr\ (f \times id) \\ first\ (f \ggg g) & = & first\ f \ggg first\ g \\ first\ f \ggg arr\ (id \times g) & = & arr\ (id \times g) \ggg first\ f \\ first\ f \ggg arr\ fst & = & arr\ fst \ggg f \\ first\ (first\ f) \ggg arr\ assoc & = & arr\ assoc \ggg first\ f \end{array}$$

where the functions $\times$ and *assoc* are defined as follows:

$$\begin{array}{l} (f \times g)\ (a,b) = (f\ a, g\ b) \\ assoc\ ((a,b),c) = (a,(b,c)) \end{array}$$

Graphically the functions associated with the arrow type are the following:



The function *arr* allows us to introduce "pure" arrows which are simple functions from their inputs to their outputs. The function $\ggg$ is similar to $\gg=$: it composes two computations. The function *first* is the critical one for our purposes: it allows us to apply an arrow to a component of the *global state*. The equations above ensure that these operations are always well-defined even with arbitrary permutations and change of associativity.

Given these three basic functions, we can define more useful combinators. For instance, we can define a combinator that applies its argument to the second component instead of the first:

$$second \in Arrow\ a \Rightarrow a\ b\ c \rightarrow a\ (d, b)\ (d, c)$$
$$second\ f = arr\ swap \ggg first\ f \ggg arr\ swap$$
$$\textbf{where}\ swap\ (x, y) = (y, x)$$

and a combinator which processes both components of a pair:

$$(***) \in Arrow\ a \Rightarrow a\ b\ c \rightarrow a\ d\ e \rightarrow a\ (b, d)\ (c, e)$$
$$f *** g = first\ f \ggg second\ g$$

which is equivalent to first apply f to the *first* argument and then apply g to the second argument.

Also, we can define a combinator which builds a pair from the results of two arrows:

$$(\&\&\&) \in Arrow\ a \Rightarrow a\ b\ c \rightarrow a\ b\ d \rightarrow a\ b\ (c, d)$$
$$f \&\&\& g = arr\ (\lambda b \rightarrow (b, b)) \ggg (f *** g))$$

Now suppose we want to choose between two arrows on the basis of an input. For that Hughes (HUGHES, 2000) introduced a dynamic choice operator and instead of enlarging the existing $Arrow$ class further, he defined a new class called $ArrowChoice$. In this way we can define arrow types which do not support the dynamic choice operator.

The definition of the choice combinator uses the pre-defined Haskell's sum type:

$$\textbf{data}\ Either\ a\ b = Left\ a \mid Right\ b$$

Then, the new class (which can be viewed as a *subclass* of $Arrow$) requires a *left* function:

$$\textbf{class}\ Arrow\ a \Rightarrow ArrowChoice\ a\ \textbf{where}$$
$$left \in a\ b\ c \rightarrow a\ (Either\ b\ d)\ (Either\ c\ d)$$

where $left\ f$ invokes f only on $Left$ inputs, and leaves $Right$ inputs unchanged.

Using $Left$ and the other combinators we can derive some more interesting combinators:

$$right \in ArrowChoice\ a \Rightarrow a\ b\ c \rightarrow a\ (Either\ d\ b)\ (Either\ d\ c)$$
$$right\ f = arr\ mirror \ggg left\ f \ggg arr\ mirror$$
$$\textbf{where}\ mirror\ (Left\ x) = Right\ x$$
$$mirror\ (Right\ y) = Left\ y$$

similarly $right\ f$ invokes f only on $Right$ inputs, and leaves $Left$ inputs unchanged.

The last interesting function we show here is $f \mid\mid\mid g$ which passes $Left$ inputs to f and $Right$ inputs to g:

$$
\begin{aligned}
&(|||) \in \mathit{ArrowChoice}\ a \Rightarrow a\ b\ d \to a\ c\ d \to a\ (\mathit{Either}\ b\ c)\ d\\
&f\ |||\ g \quad = f +\!\!+\!\!+ g \ggg \mathit{arr}\ \mathit{untag}\\
&\qquad\qquad \mathbf{where}\ \mathit{untag}\ (\mathit{Left}\ x) = x\\
&\qquad\qquad\qquad\quad \mathit{untag}\ (\mathit{Right}\ y) = y\\
&(+\!\!+\!\!+) \in \mathit{ArrowChoice}\ a \Rightarrow a\ b\ c \to a\ b'\ c' \to a\ (\mathit{Either}\ b\ b')\ (c\ c')\\
&f +\!\!+\!\!+ g = \mathit{left}\ f \ggg \mathit{right}\ g
\end{aligned}
$$

3.2.2 A Better Notation for Arrows

Following the Haskell's monadic **do**-notation, Paterson (2001) presented an extension to Haskell with an improved syntax for writing computations using arrows. He defined a preprocessor, that reads as input a Haskell script augmented with arrow notation, and outputs a plain Haskell script. We concentrate only on the explanation of new forms which we use in our examples. Here is a simple example to illustrate the notation:

$$
\begin{aligned}
&\mathit{op} \in \mathit{Ty}\ (T_1, T)\ (T_1, T)\\
&\mathit{op} = \mathit{proc}\ (a, b) \to \mathbf{do}\\
&\qquad\quad r \leftarrow f \prec a\\
&\qquad\quad \mathit{returnA} \prec (r, b)
\end{aligned}
$$

The **do**-notation simply sequences the actions in its body. The function $\mathit{returnA}$ is the equivalent for arrows of the monadic function return. The two additional keywords are:

- the *arrow abstraction* **proc** which constructs an arrow instead of a regular function.
- the *arrow application* $\prec$ which feeds the value of an expression into an arrow.

Paterson (2001) shows that the above notation is general enough to express arrow computations and the preprocessor is implemented such that it translates the new syntax to regular Haskell. In the case of `op` above, the translation to Haskell produces the following code:

$$
\begin{aligned}
&e \in \mathit{Ty}\ (T_1, T)\ (T_1, T)\\
&e = \mathit{first}\ f
\end{aligned}
$$

for $f \in \mathit{Ty}\ T_1\ T_1$. As the example shows, the output of the preprocessor is quite optimised. However, the preprocessor should be executed manually and it is not strictly necessary to add types to the expressions.

The notation is also implemented directly in GHC, from version 6.2, where it is enabled by the $-\mathit{farrows}$ option. Hence, if our types are an instance of the Arrow class we can use the arrow notation directly in the Haskell code.

3.2.3 The Arrow Transformers

Arrows have the same property we present in Section 3.1.3 for monads, that is, we can define *arrow transformers* which map simpler arrow types to more complex ones. An arrow transformer is, by analogy with a monad transformer, just an arrow type parameterised on another arrow type, such that we can lift operations on the parameter type to the new type. For instance, any arrow type can be lifted to an arrow type supporting failures:

$$
\begin{aligned}
&\mathbf{newtype}\ \mathit{MaybeArrowT}\ a\ b\ c = \mathit{MAT}\ (a\ b\ (\mathit{Maybe}\ c))\\
&\mathit{unMAT}\ (\mathit{MAT}\ c) = c
\end{aligned}
$$

That is, the result of the arrow can indicate failure. We can lift arrows to this type using:

$$
\begin{aligned}
&\mathit{liftMaybe} \in \mathit{Arrow}\ a \Rightarrow a\ b\ c \to \mathit{MaybeArrowT}\ a\ b\ c\\
&\mathit{liftMaybe}\ f = \mathit{MAT}\ (f \ggg \mathit{arr}\ \mathit{Just})
\end{aligned}
$$

Again, in general, the arrow operators in the new type are defined in terms of the operators in the parameter type. Moreover, the arrow operations need to handle failures, which means they need to make dynamic decisions, and therefore must require that the parameter arrow type supports choice:

$$
\begin{aligned}
&\textbf{instance } \mathit{ArrowChoice}\ a \Rightarrow \mathit{Arrow}\ (\mathit{MaybeArrowT}\ a)\ \textbf{where}\\
&\quad \mathit{arr}\ f = \mathit{liftMaybe}\ (\mathit{arr}\ f)\\
&\quad f \ggg g = \textbf{let}\ f_0 = \mathit{unMAT}\ f\\
&\qquad\qquad\qquad\ g_0 = \mathit{unMAT}\ g\\
&\qquad\qquad\ \textbf{in}\ \mathit{MAT}\ (f_0 \ggg \mathit{arr}\ (\lambda z \rightarrow \textbf{case}\ z\ \textbf{of}\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{Just}\ c \quad \rightarrow \mathit{Left}\ c\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{Nothing} \rightarrow \mathit{Right}\ \mathit{Nothing}) \ggg\\
&\qquad\qquad\qquad\qquad\qquad\qquad (g_0 \mid\mid\mid \mathit{arr}\ \mathrm{id}))\\
&\quad \mathit{first}\ (\mathit{MAT}\ f) = \mathit{MAT}\ (\mathit{first}\ f \ggg\\
&\qquad\qquad\qquad\qquad\qquad \mathit{arr}\ (\lambda(c,d) \rightarrow \textbf{case}\ c\ \textbf{of}\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{Just}\ c \quad \rightarrow \mathit{Just}\ (c,d)\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{Nothing} \rightarrow \mathit{Nothing}))
\end{aligned}
$$

3.2.4 Indexed Arrows

As we have presented the definition of indexed monads, i.e., *Kleisli structures*, we want to define indexed arrows. Recall definition 3.2.1, where inc is a *functor* from $\mathcal{V}$ to $\mathcal{C}$. It is because of inc that we have the quantifier $forall$ in the definition of the $Arrow$ class in Section 3.2.1. But again, suppose we want to *select* some elements from $\mathcal{V}$ to build our computations. For that purpose we define here a generalisation of *Freyd-categories*, which we call *indexed Freyd-categories*. Essentially, a indexed Freyd-category is build on top of an *indexed* category of values.

An indexed category is a well know construction from category theory, which model uniformly defined families of categories (TARLECKI; BURSTALL; GOGUEN, 1991).

Definition 3.2.2 (Indexed Category) *An indexed category $\mathcal{C}$ over an index I is a functor $I^{op} \rightarrow Cat$. Given an index $i \in I$, we may write $\mathcal{C}_i$ for the category $\mathcal{C}(i)$, and given an index morphism $\sigma :: i \rightarrow j$, we may write $\mathcal{C}_\sigma$ for the functor $\mathcal{C}(\sigma) : \mathcal{C}(j) \rightarrow \mathcal{C}(i)$. Also, we may call $\mathcal{C}_i$ the i^{th} component category of $\mathcal{C}$.*

Definition 3.2.3 (Indexed Freyd-category) *A indexed Freyd-category is a structure $(I, \mathcal{V}, \mathcal{C}, inc, \ltimes)$ with:*

- *a category $\mathcal{V}$ with finite products (the value category),*
- *an index set $I \in Set$, taken as a trivial category and used as the index category,*
- *an indexed category $\mathcal{V}_i$, such that $i \in I$,*
- *a category $\mathcal{C}_i$ with the same objects as $\mathcal{V}_i$ (the computation category), such that $i \in I$,*
- *a family of functors inc indexed by $i \in I$, with $inc_i :: \mathcal{V}_i \rightarrow \mathcal{C}_i$ that is the identity on objetcs,*
- *a family of functors $\ltimes$ indexed by $i, j \in I$, with $\ltimes_{i,j} :: \mathcal{C}_i \times \mathcal{V}_j \rightarrow \mathcal{C}_{(i,j)}$ such that*

$$inc_i\ x \ltimes_{i,j} y = inc_{(i,j)}(x \times y)$$

and the following natural isomorphisms, indexed by $i, j, k \in I$*, in* $\mathcal{V}$

$$\begin{array}{lcl} assoc_{\times} & : & (A_i \times B_j) \times C_k \cong A_i \times (B_j \times C_k) \\ unitr_{\times} & : & A_i \times 1 \cong A_i \end{array}$$

extend to natural isomorphism in the indexed category $\mathcal{C}$*:*

$$\begin{array}{lcl} inc\ assoc_{\times} & : & (A_i \ltimes B_j) \ltimes C_k \cong A_i \ltimes (B_j \times C_k) \\ inc\ unitr_{\times} & : & A_i \ltimes 1 \cong A_i \end{array}$$

Below, we show the diagram for an indexed Freyd-category:

$$\begin{array}{ccc} \mathcal{C}_i & & \mathcal{C}_{(i,j)} \\ \uparrow{\scriptstyle inc} & & \uparrow{\scriptstyle \ltimes_{i,j}} \\ \mathcal{V}_i & & \mathcal{C}_i \times \mathcal{V}_j \end{array}$$

Now, the definitions of arr, $\ggg$, and $first$ in Haskell could be rephrased as:

$arr \quad \in forall\ b_i\ c_j.(b_i \to c_j) \to a\ b_i\ c_j$
$(\ggg) \in forall\ b_i\ c_j\ d_k.a\ b_i\ c_j \to a\ c_j\ d_k \to a\ b_i\ d_k$
$first\ \in forall\ b_i\ c_j\ d_k.a\ b_i\ c_j \to a\ (b_i, d_k)\ (c_j, d_k)$

such that $i, j, k \in I$.

In terms of type classes we would like to rewrite the $Arrow$ class to become a indexed $Arrow$ class allowing us to write, for instance:

$\mathbf{class}\ IArrow\ m\ \mathbf{where}$
$arr \in (I\ b, I\ c) \Rightarrow (b \to c) \to a\ b\ c$
$(\ggg) \in (I\ b, I\ c, I\ d) \Rightarrow a\ b\ c \to a\ c\ d \to a\ b\ d$
$first \in (I\ b, I\ c, I\ d) \Rightarrow a\ b\ c \to a\ (b, d)\ (c, d)$

Unfortunately, in the current version of GHC (6.5), the types for arr, $\ggg$, and $first$ used in the arrow notation may *not* be overload to have our proper types. This is a problem related to "rebindable syntax in GHC" (SABRY, 2006), and it seems that it is quite hard to solve. Therefore, we can define our *indexed arrows* but we *can not* run programs directly into Haskell based on them for the moment. As an option to compile them manually to pure Haskell we still can use Paterson's preprocessor.

3.3 Summary

In this Chapter we have presented two mathematical notions, *monads* and *arrows*, which are now widely used in computer science, mainly in programming language semantics and design. For us these constructions are interesting as a tool to structure and elegantly model computational effects introduced by quantum computations. Specially we have presented generalisations of these concepts, which we call *indexed monads* (respective to Kleisli structures (ALTENKIRCH; REUS, 1999)) and *indexed arrows*, which are the right structures to model finite complex vector spaces build over a computational basis set (that is, the set of classical observable values).

4 QML: QUANTUM DATA AND QUANTUM CONTROL

This chapter is based on (ALTENKIRCH et al., 2005), where we developed a sound and *complete* equational theory for a pure (omitting measurements) sublanguage of QML.

The language QML was introduced in (ALTENKIRCH; GRATTAGE, 2005) and (ALTENKIRCH; GRATTAGE, 2005b). QML is a first order functional language which features both quantum data structures and quantum control structures, in particular a quantum conditional structure $\mathbf{if}^\circ$ - which analyses quantum data without measuring, and hence without changing the data.

QML's type system is based on *strict linear logic*, that is linear logic with contraction, but without implicit weakening.

The chapter is divided in three parts: i) an informal view of the language; ii) proof of completeness for a classical sublanguage of QML; and iii) proof of completeness for a pure sublanguage of QML.

A next step would be generalise this approach to the full QML including measurements. In next chapters we structure a model for general (including measurements) and complete (including quantum and classical data as well as the interchanging between quantum and classical worlds) quantum computations using arrows. We hope to integrate the results of next chapters with a quantum programming language like QML.

4.1 The Language QML

We consider some interesting examples to give further intuition about the semantics of the language. We present the examples using global functions definitions.

The following three functions correspond to simple rotations on qubits:

$$\begin{aligned} qnot\ x &= \mathbf{if}^\circ\ x\ \mathbf{then}\ false\ \mathbf{else}\ true \\ had\ x\ &= \mathbf{if}^\circ\ x\ \mathbf{then}\ ((-1) * true + false)\ \mathbf{else}\ (true + false) \\ z\ x\ \ \ &= \mathbf{if}^\circ\ x\ \mathbf{then}\ (i * true)\ \mathbf{else}\ false \end{aligned}$$

The first is the quantum version of boolean negation: it behaves as usual when applied to classical values but it also applies to quantum data. Evaluating $qnot\ (\kappa * false + \iota * true)$ swaps the probability amplitudes associated with $false$ and $true$. The second function represents the fundamental *Hadamard* matrix, and the third represents the *phase* gate.

The function:

$$\begin{aligned} cnot\ c\ x = \mathbf{if}^\circ\ & c \\ & \mathbf{then}\ (true,\ qnot\ x) \\ & \mathbf{else}\ \ (false, x) \end{aligned}$$

is the conditional-not operation, which behaves as follows: if the control qubit c is $true$ it negates the second qubit x; otherwise it leaves it unchanged. When the control qubit

is in some superposition of $true$ and $false$, the result is a superposition of the two pairs resulting from the evaluation of each branch of the conditional. For example, evaluating $cnot\ (false + true)\ false$ produces the ***entangled*** pair $(false, false) + (true, true)$.

To motivate the main aspects of the QML type system (which we presented in the next chapter), we examine in detail the issues related to copying and discarding quantum data.

4.1.1 Copying Quantum Data

A simple example where quantum data appears to be copied, in violation of the *no-cloning* theorem (NIELSEN; CHUANG, 2000), is:

$$\begin{array}{l}\mathbf{let}\ x = false + true \\ \mathbf{in}\ (x, x)\end{array}$$

As the formal semantics of QML clarifies, this expression does not actually clone quantum data; rather it *shares* one copy of the quantum data. With this interpretation, one can freely duplicate variables bound to quantum data. When translated to the type system, this means that the type system imposes no restrictions on the use of the structural rule of *contraction*.

4.1.2 Discarting Quantum Data

In contrast, a simple example where quantum data appears to be discarded is:

$$\begin{array}{l}\mathbf{let}\ (x, y) = \{\,(false, false) \mid (true, true)\,\} \\ \mathbf{in}\ x\end{array}$$

Indeed the quantum data bound to y is discarded, which according to both the physical interpretation of quantum computation and the QML semantics explained in next chapters corresponds to a *measurement* of y. This measurement could be made explicit in the QML syntax by writing:

$$\begin{array}{l}\mathbf{let}\ (x, y) = \{\,(false, false) \mid (true, true)\,\} \\ \mathbf{in\ meas}\ y\ \mathbf{in}\ x\end{array}$$

Since measurement is semantically quite complicated to deal with, they should be explicitly represented in the syntax and typing judgments. Thus the type system is designed to reject the first expression and accept the second. This means that the structural rule of *weakening* is controlled and can only be used when it corresponds to an explicit measurement.

Of course, the situation is more subtle than just syntactically checking whether a variable is used or not. Consider the expression:

$$\mathbf{if}^{\circ}\ x\ \mathbf{then}\ true\ \mathbf{else}\ true$$

The expression appears, syntactically at least, to use x. However given the semantics of $\mathbf{if}^{\circ}$ which returns a superposition of the branches, the expression happens to return $true$ without really *using* any information about x. In order to maintain the invariant that all measurements are explicit, the type system rejects the above expression. In more detail, an expression:

$$\mathbf{if}^{\circ}\ x\ \mathbf{then}\ t\ \mathbf{else}\ u$$

is only accepted if t and u are *orthogonal* quantum values. This notion intuitively ensures that the conditional operator does not implicitly discard any information about x during the evaluation.

4.2 The Classical Sublanguage

4.2.1 Syntax

By the classical sublanguage, we mean a classical first-order functional language. The syntax of terms is the following:

$$
\begin{array}{lll}
(\mathit{Variables}) & x, y, ... \in & \mathit{Vars} \\
(\mathit{Patterns}) & p, q & ::= x \mid (x, y) \\
(\mathit{Terms}) & t, u, e & ::= x \mid () \mid (t, u) \\
 & & \mid \quad \mathbf{let}\ p = t\ \mathbf{in}\ u \\
 & & \mid \quad \mathbf{if}\ t\ \mathbf{then}\ u\ \mathbf{else}\ u' \\
 & & \mid \quad \mathit{false} \mid \mathit{true}
\end{array}
$$

The classic sublanguage consists of variables, **let**-expressions, unit, pairs, booleans, and conditionals.

4.2.2 Type System

The main rôle of the type system is to control the use of variables. The typing rules of QML are based on strict linear logic, where contractions are implicit and weakenings are not allowed when they correspond to information loss. As explained in the previous section, weakenings correspond to measurements, which are not supported in the subset of the language discussed in this work.

We use σ, τ, ρ to vary over QML types which are given by the following grammar:

$$\sigma = \mathcal{Q}_1 \mid \mathcal{Q}_2 \mid \sigma \otimes \tau$$

As apparent from the grammar, QML types are first-order and finite: there are no higher-order types and no recursive types. The only types we can represent are the types of collections of qubits.

Typing contexts (Γ, Δ) are given by:

$$\Gamma = \bullet \mid \Gamma, x : \sigma$$

where $\bullet$ stands for the empty context, but is omitted if the context is non-empty. For simplicity we assume that every variable appears at most once. Contexts correspond to functions from a finite set of variables to types. We introduce the operator $\otimes$, mapping pairs of contexts to contexts:

$$(\Gamma, x : \sigma) \otimes (\Delta, x : \sigma) = (\Gamma \otimes \Delta), x : \sigma \tag{4.1}$$

$$(\Gamma, x : \sigma) \otimes \Delta = (\Gamma \otimes \Delta), x : \sigma \quad \text{if } x \notin \mathrm{dom}\,(\Delta) \tag{4.2}$$

$$\bullet \otimes \Delta = \Delta \tag{4.3}$$

This operation is partial: it is only well-defined if the two contexts do not assign different types to the same variable. Whenever we use this operator we implicitly assume that it is well-defined.

Figure 4.1 presents the rules for deriving valid typing judgements $\Gamma \vdash t : \sigma$. The only variables that may be dropped from the context are the ones of type $\mathcal{Q}_1$ which, by definition, carry no information. Otherwise the type system forces every variable in the context to be used (perhaps more than once if it is shared).

To see how the type system works in more details consider the following derivation where the same variable is being used twice:

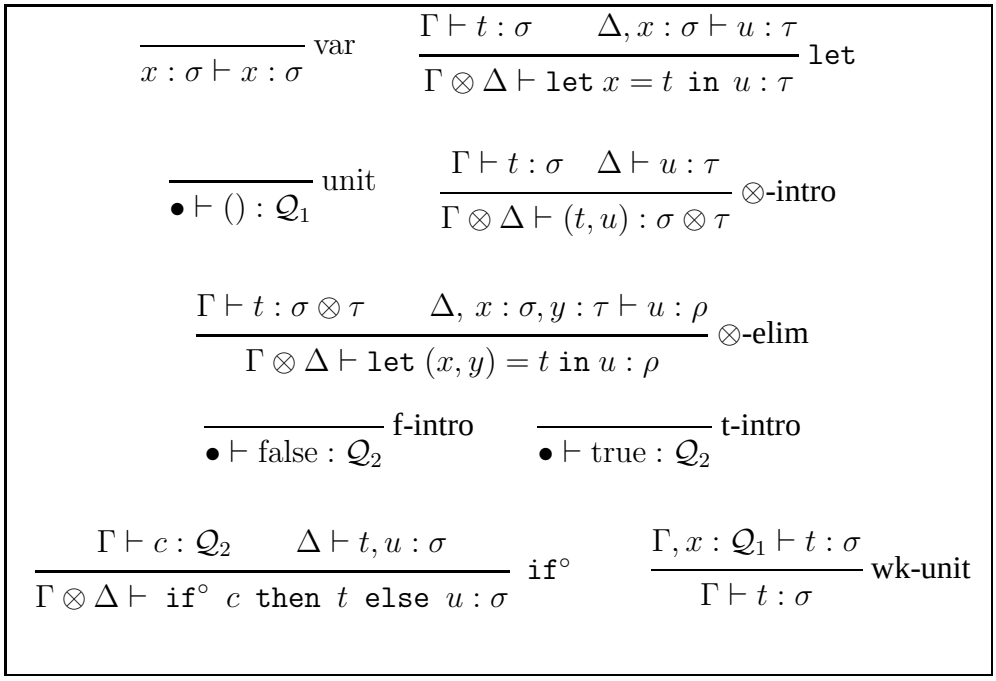


Figure 4.1: Typing classical terms

$$\dfrac{\dfrac{}{\bullet \vdash \text{true} : \mathcal{Q}_2}\ \text{t-intro} \qquad \dfrac{\dfrac{}{x : \mathcal{Q}_2 \vdash x : \mathcal{Q}_2}\ \text{var} \qquad \dfrac{}{x : \mathcal{Q}_2 \vdash x : \mathcal{Q}_2}\ \text{var}}{\bullet, x : \mathcal{Q}_2 \vdash (x, x) : \mathcal{Q}_2 \otimes \mathcal{Q}_2}\ \otimes}{\bullet \otimes \bullet \vdash \texttt{let}\ x = \text{true}\ \texttt{in}\ (x, x) : \mathcal{Q}_2 \otimes \mathcal{Q}_2}\ \text{let}$$

The key point here is the context. Note that we can only prove true with the empty context. Intuitively this means that the context cannot include variables which are never used. The **let** expression introduces the variable x in the context, and using the operation $\otimes$ we can *share* this variable:

$$\begin{array}{rcll} \bullet, x : \mathcal{Q}_2 & = & \bullet \otimes \bullet, x : \mathcal{Q}_2 & \text{(by 4.3)} \\ & = & (\bullet, x : \mathcal{Q}_2) \otimes (\bullet, x\mathcal{Q}_2) & \text{(by 4.1)} \end{array}$$

4.2.3 The Category of Typed Terms

The set of typed terms can be organised in an elegant categorical structure, which facilitates the proofs later. The objects of the category are contexts; the homset between the objects Γ and Δ, denoted $\mathrm{Tm}\,\Gamma\,\Delta$, consists of all the terms t such that $\Gamma \vdash t : |\Delta|$ where $|\Delta|$ views the context Δ as a type. This latter map is naturally defined as follows:

$$\begin{array}{rcl} |\bullet| & = & \mathcal{Q}_1 \\ |\Gamma, x : \sigma| & = & |\Gamma| \otimes \sigma \end{array}$$

For each context Γ, the identity $1_\Gamma \in \mathrm{Tm}\Gamma\Gamma$ is defined as follows:

$$\begin{array}{rcl} 1_\bullet & = & () \\ 1_{\Gamma, x:\sigma} & = & (1_\Gamma, x) \end{array}$$

To express composition, we first define:

$$\begin{array}{lcl} \textbf{let}^*\ \bullet = u\ \textbf{in}\ t & \equiv & t \\ \textbf{let}^*\ \Gamma, x : \sigma = u\ \textbf{in}\ t & \equiv & \textbf{let}\ (x_r, x) = u\ \textbf{in}\ \textbf{let}^*\ \Gamma = x_r\ \textbf{in}\ t \end{array}$$

Given $d \in \mathrm{Tm}\,\Delta\,\Gamma$ and $e \in \mathrm{Tm}\,\Gamma\,\Theta$, the composition $e \circ d \in \mathrm{Tm}\,\Delta\,\Theta$ is given by the term $\mathbf{let}^*\ \Gamma = d\ \mathbf{in}\ e$.

For example, consider we want to model in this category the term $\bullet \vdash \texttt{let}\ x = \mathrm{false}\ \texttt{in}\ (((), x), x) : (\mathcal{Q}_1 \otimes \mathcal{Q}_2) \otimes \mathcal{Q}_2$. Then, we need the following objects:

$$\begin{aligned}\Gamma &= \bullet \\ \Theta &= \bullet, y : Q_2, z : Q_2\end{aligned}$$

such that

$$\begin{aligned}|\Gamma| &= \mathcal{Q}_1 \\ |\Theta| &= (|\bullet| \otimes \mathcal{Q}_2) \otimes \mathcal{Q}_2 = (\mathcal{Q}_1 \otimes \mathcal{Q}_2) \otimes \mathcal{Q}_2\end{aligned}$$

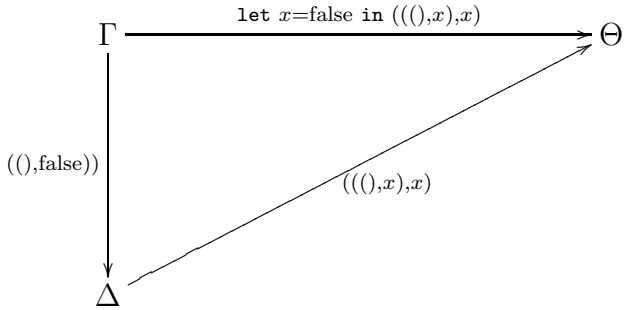
and we have that the term is the arrow below:

$$\Gamma \xrightarrow{\texttt{let}\ x=\mathrm{false}\ \texttt{in}\ (((),x),x)} \Theta$$

Now, as the typing rules suggest the let expression above is the same as the composition of $\bullet \vdash ((), \mathrm{false}) : \mathcal{Q}_1 \otimes \mathcal{Q}_2$ with $\mathcal{Q}_1 \otimes \mathcal{Q}_2 \vdash (((), x), x) : (\mathcal{Q}_1 \otimes \mathcal{Q}_2) \otimes \mathcal{Q}_2$, that is, given

$$\begin{aligned}\Delta &= \bullet, x : \mathcal{Q}_2 \\ |\Delta| &= |\bullet| \otimes \mathcal{Q}_2 = \mathcal{Q}_1 \otimes \mathcal{Q}_2\end{aligned}$$

we want the following diagram to comute:



Checking: the composition $(((), x), x) \circ (x, \mathrm{false})$ is:

$\mathbf{let}^*\ \Delta = (x, \mathit{false})\ \mathbf{in}\ (((), x), x)$
$(\mathit{by\ definition}\ \mathbf{of}\ \Delta)$
$= \mathbf{let}^*\ \bullet, x : \mathcal{Q}_2 = ((), \mathit{false})\ \mathbf{in}\ (((), x), x)$
$(\mathit{by\ definition}\ \mathbf{of}\ \mathbf{let}^*)$
$= \mathbf{let}\ (x_r, x) = ((), \mathit{false})\ \mathbf{in}\ \mathbf{let}^*\ \bullet = x_r\ \mathbf{in}\ (((), x), x)$
$(\mathit{by\ definition}\ \mathbf{of}\ \mathbf{let}^*)$
$= \mathbf{let}\ (x_r, x) = ((), \mathit{false})\ \mathbf{in}\ (((), x), x)$
$(\mathit{by}\ \beta\ \mathit{equation})$
$= \mathbf{let}\ x_r = ()\ \mathbf{in}\ \mathbf{let}\ x = \mathit{false}\ \mathbf{in}\ (((), x), x)$
$(\mathit{by\ substitution})$
$= \mathbf{let}\ x = \mathit{false}\ \mathbf{in}\ (((), x), x)$

4.2.4 Semantics

The intention is to interpret every type σ and every context Γ as finite sets $[\![\sigma]\!]$ and $[\![\Gamma]\!]$, and then interpret a judgement $\Gamma \vdash t : \sigma$ as a function $[\![\Gamma \vdash t : \sigma]\!] \in [\![\Gamma]\!] \to [\![\sigma]\!]$.

In the classical case, the type $\mathcal{Q}_2$ is simply the type of booleans; the types are interpreted as follows:

$$\begin{array}{rcl} [\![\mathcal{Q}_1]\!] & = & \{0\} \\ [\![\mathcal{Q}_2]\!] & = & \{0,1\} \\ [\![\sigma \otimes \tau]\!] & = & [\![\sigma]\!] \times [\![\tau]\!] \end{array}$$

We use the abbreviation $[\![\Gamma]\!]$ for $[\![|\Gamma|]\!]$.

The meaning function is defined in Figure 4.2 by induction over the structure of type derivations. It uses the following auxiliary maps:

- $id : S \to S$ defined by $id(a) = a$
- $id^* : S \to [\![\mathcal{Q}_1]\!] \times S$ and its inverse id_* defined by $id^*(a) = (0, a)$ and $id_*(0, a) = a$
- For $a \in S$, the family of constant functions *const* $a : [\![\mathcal{Q}_1]\!] \to S$ defined by $(\textit{const}\ a)(0) = a$.
- $\delta : S \to (S, S)$ defined by $\delta(a) = (a, a)$
- $swap : S \times T \to T \times S$ defined by $swap(a, b) = (b, a)$. We will usually implicitly use *swap* to avoid cluttering the figures with maps which just re-shuffle values.
- For any two functions $f \in S_1 \to T_1$ and $g \in S_2 \to T_2$, the function $(f \times g) : (S_1 \times S_2) \to (T_1 \times T_2)$ is defined as usual:

$$(f \times g)(a, b) \quad = \quad (f\ a, g\ b)$$

- $\delta_{\Gamma,\Delta} : [\![\Gamma \otimes \Delta]\!] \to [\![\Gamma]\!] \times [\![\Delta]\!]$. This map is defined by induction on the definition of $\Gamma \otimes \Delta$ as follows:

$$\delta_{\Gamma,\Delta} = \begin{cases} \delta_{\Gamma',\Delta'} \times \delta & \text{if } \Gamma = \Gamma', x : \sigma \text{ and } \Delta = \Delta', x : \sigma \\ \delta_{\Gamma',\Delta} \times id & \text{if } \Gamma = \Gamma', x : \sigma \text{ and } x \notin \text{dom}\,(\Delta) \\ id^* & \text{if } \Gamma = \bullet \end{cases}$$

  Intuitively, the map $\delta_{\Gamma,\Delta}$ takes an incoming environment for an expression, creates shared copies of the appropriate values, and rearranges them (the shuffling is implicit and not shown in the above definition) into two environments that are then passed to the subexpressions.

- For any two functions $f, g \in S \to T$, we define the conditional $f|g \in ([\![\mathcal{Q}_2]\!] \times S) \to T$ as follows:

$$\begin{array}{rcl} (f|g)\ (1, a) & = & f\ a \\ (f|g)\ (0, a) & = & g\ a \end{array}$$

4.2.5 Examples

We interpreted some simple QML expressions as explained above. Consider the following function representing boolean negation:

$qnot\ x = \mathbf{if}^\circ\ x\ \mathbf{then}\ \mathit{false}\ \mathbf{else}\ \mathit{true}$

Formally, this function can be written as the type judgement below:

$$x : \mathcal{Q}_2 \vdash \texttt{if}^\circ\ x\ \texttt{then}\ \text{false}\ \texttt{else}\ \text{true} : \mathcal{Q}_2$$

$$
\begin{array}{rcl}
[\![\bullet \vdash () : \mathcal{Q}_1]\!] & = & \mathit{const}\ 0 \\
[\![\bullet \vdash \mathrm{false} : \mathcal{Q}_2]\!] & = & \mathit{const}\ 0 \\
[\![\bullet \vdash \mathrm{true} : \mathcal{Q}_2]\!] & = & \mathit{const}\ 1 \\
[\![x : \sigma \vdash x : \sigma]\!] & = & id_* \\
[\![\Gamma \otimes \Delta \vdash \mathtt{let}\ x = t\ \mathtt{in}\ u : \tau]\!] & = & g \circ (f \times id) \circ \delta_{\Gamma,\Delta} \\
 & & \text{where}\ \ f = [\![\Gamma \vdash t : \sigma]\!] \\
 & & \qquad\ \ g = [\![\Delta, x : \sigma \vdash u : \tau]\!] \\
[\![\Gamma \otimes \Delta \vdash (t, u) : \sigma \otimes \tau]\!] & = & (f \times g) \circ \delta_{\Gamma,\Delta} \\
 & & \text{where}\ \ f = [\![\Gamma \vdash t : \sigma]\!] \\
 & & \qquad\ \ g = [\![\Delta \vdash u : \tau]\!] \\
[\![\Gamma \otimes \Delta \vdash \mathtt{let}\ (x, y) = t\ \mathtt{in}\ u : \rho]\!] & = & g \circ (f \times id) \circ \delta_{\Gamma,\Delta} \\
 & & \text{where}\ \ f = [\![\Gamma \vdash t : \sigma \otimes \tau]\!] \\
 & & \qquad\ \ g = [\![\Delta,\ x : \sigma, y : \tau \vdash u : \rho]\!] \\
[\![\Gamma \otimes \Delta \vdash\ \mathtt{if}^\circ\ c\ \mathtt{then}\ t\ \mathtt{else}\ u : \sigma]\!] & = & (g|h) \circ (f \times id) \circ \delta_{\Gamma,\Delta} \\
 & & \text{where}\ \ f = [\![\Gamma \vdash c : \mathcal{Q}_2]\!] \\
 & & \qquad\ \ g = [\![\Delta \vdash t : \sigma]\!] \\
 & & \qquad\ \ h = [\![\Delta \vdash u : \sigma]\!] \\
[\![\Gamma \vdash t : \sigma]\!] & = & f \circ id^* \\
 & & \text{where}\ f = [\![\Gamma, x : \mathcal{Q}_1 \vdash t : \sigma]\!]
\end{array}
$$

Figure 4.2: Meaning of classical derivations

Because the initial empty context is omitted if the context is non-empty, we have $\bullet, x : \mathcal{Q}_2$, which is equivalent to:

$$
\begin{array}{rcll}
\bullet, x : \mathcal{Q}_2 & = & (\bullet \otimes \bullet), x : \mathcal{Q}_2 & \text{(by 4.3)} \\
 & = & (\bullet, x : \mathcal{Q}_2) \otimes \bullet & \text{(by 4.2)}
\end{array}
$$

Therefore, we shall interpret the type judgement rewritten as:

$$
\begin{array}{l}
[\![(\bullet, x : \mathcal{Q}_2) \otimes \bullet \vdash\ \mathtt{if}^\circ\ x\ \mathtt{then}\ \mathrm{false}\ \mathtt{else}\ \mathrm{true} : \mathcal{Q}_2]\!] \\
= (g|h) \circ (f \times id) \circ \delta_{(\bullet, x:\mathcal{Q}_2),\bullet} \\
\text{where} \qquad f = [\![\bullet, x : \mathcal{Q}_2 \vdash x : \mathcal{Q}_2]\!] = id_* \\
\qquad\qquad\ \ g = [\![\bullet \vdash \mathrm{false} : \mathcal{Q}_2]\!] = \mathit{const}\ 0 \\
\qquad\qquad\ \ h = [\![\bullet \vdash \mathrm{true} : \mathcal{Q}_2]\!] = \mathit{const}\ 1 \\
\qquad\quad \delta_{(\bullet, x:\mathcal{Q}_2),\bullet} = \delta_{\bullet,\bullet} \times id = id^* \times id \\
= (\mathit{const}\ 0|\mathit{const}\ 1) \circ (id_* \times id) \circ (id^* \times id) \\
\text{– by definition of } id_* \text{ and } id^* \\
= (\mathit{const}\ 0|\mathit{const}\ 1) \circ (id \times id) \\
= (\mathit{const}\ 0|\mathit{const}\ 1) :: ([\![\mathcal{Q}_2]\!] \times [\![\mathcal{Q}_1]\!]) \to [\![\mathcal{Q}_2]\!]
\end{array}
$$

defined as

$$
\begin{array}{rcl}
(\mathit{const}\ 0|\mathit{const}\ 1)\ (\mathbf{1}, 0) & = & \mathit{const}\ 0\ 0 = \mathbf{0} \\
(\mathit{const}\ 0|\mathit{const}\ 1)\ (\mathbf{0}, 0) & = & \mathit{const}\ 1\ 0 = \mathbf{1}
\end{array}
$$

which exactly behaves as the boolean negation.

Other example is the function which copies classical data [1]:

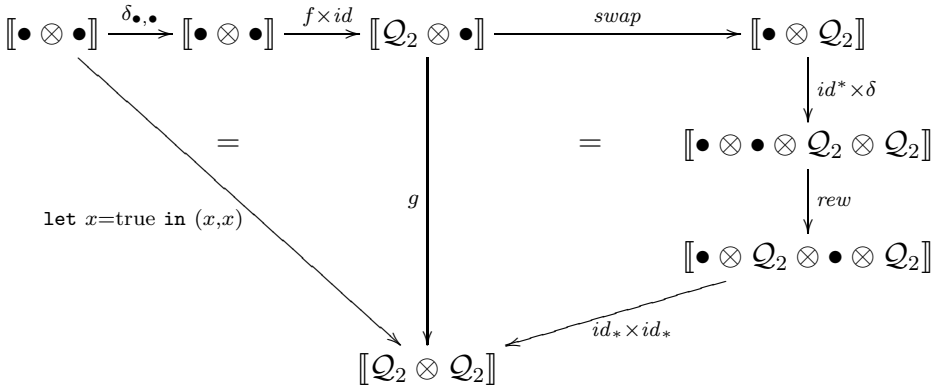
[1] This will be interesting to compare with the semantics in the next chapter for quantum data

$$
\begin{array}{l}
[\![\bullet \otimes \bullet \vdash \texttt{let}\ x = \text{true}\ \texttt{in}\ (x,x) : \mathcal{Q}_2 \otimes \mathcal{Q}_2]\!] \\
= g \circ (f \times id) \circ \delta_{\bullet,\bullet} \\
\text{where} \quad f = [\![\bullet \vdash \text{true} : \mathcal{Q}_2]\!] = \mathit{const}\ 1 \\
\qquad\quad g = [\![\bullet, x : \mathcal{Q}_2 \vdash (x,x) : \mathcal{Q}_2 \otimes \mathcal{Q}_2]\!] \\
\qquad\qquad \text{- by definition of } \otimes \text{ on contexts} \\
\qquad\quad = [\![\bullet \otimes \bullet, x : \mathcal{Q}_2 \vdash (x,x) : \mathcal{Q}_2 \otimes \mathcal{Q}_2]\!] \\
\qquad\qquad \text{- by definition of } \otimes \text{ on contexts} \\
\qquad\quad = [\![(\bullet, x : \mathcal{Q}_2) \otimes (\bullet, x : \mathcal{Q}_2) \vdash (x,x) : \mathcal{Q}_2 \otimes \mathcal{Q}_2]\!] \\
\qquad\quad = (f' \times g') \circ \delta_{(\bullet,x:\mathcal{Q}_2),(\bullet,x:\mathcal{Q}_2)} \\
\qquad\qquad \text{where} \quad f' = [\![\bullet, x : \mathcal{Q}_2 \vdash x : \mathcal{Q}_2]\!] = id_* \\
\qquad\qquad\qquad\quad g' = [\![\bullet, x : \mathcal{Q}_2 \vdash x : \mathcal{Q}_2]\!] = id_* \\
\qquad\qquad\qquad \delta_{(\bullet,x:\mathcal{Q}_2),(\bullet,x:\mathcal{Q}_2)} = \delta_{\bullet,\bullet} \times \delta = id^* \times \delta
\end{array}
$$

Remember that the shuffling is implicit in the definition of $\delta_{\Gamma,\Delta}$. Basically, for the case above, that is for $\delta_{(\bullet,x:\mathcal{Q}_2),(\bullet,x:\mathcal{Q}_2)}$, where the variable x is being shared, we will need the following function:

$$
\begin{array}{l}
rew :: \mathcal{Q}_1 \times \mathcal{Q}_1 \times \mathcal{Q}_2 \times \mathcal{Q}_2 \to \mathcal{Q}_1 \times \mathcal{Q}_2 \times \mathcal{Q}_1 \times \mathcal{Q}_2 \\
rew = id \times swap \times id
\end{array}
$$

The final interpretation for the `let` can be analyzed in the diagram below:



4.2.6 Equational Theory

We present the equational theory for the classical sublanguage and then show its soundness and completeness. The equations refer to a set of syntactic values defined as follows:

$$val \in ValC ::= x \mid () \mid \mathit{false} \mid \mathit{true} \mid (val_1, val_2)$$

Definition 4.2.1 *The* classical equations *are grouped in four categories.*

- **let**-*equation*

$$\textbf{let}\ p = val\ \textbf{in}\ u \qquad \equiv \qquad u\,[\,val\,/\,p\,]$$

- β-*equations*

$$
\begin{array}{lcl}
\textbf{let}\ (x,y) = (t,u)\ \textbf{in}\ e & \equiv & \textbf{let}\ x = t\ \textbf{in let}\ y = u\ \textbf{in}\ e \\
\textbf{if}^\circ\ \mathit{false}\ \textbf{then}\ t\ \textbf{else}\ u & \equiv & u \\
\textbf{if}^\circ\ \mathit{true}\ \textbf{then}\ t\ \textbf{else}\ u & \equiv & t
\end{array}
$$

- *η-equations*

$$\begin{array}{lcll} () & \equiv & t & \text{-- if } t{:}\mathcal{Q}_1 \\ \mathbf{let}\ x = t\ \mathbf{in}\ x & \equiv & t & \\ \mathbf{let}\ (x,y) = t\ \mathbf{in}\ (x,y) & \equiv & t & \\ \mathbf{if}^\circ\ t\ \mathbf{then}\ \mathit{true}\ \mathbf{else}\ \mathit{false} & \equiv & t & \end{array}$$

- *Commuting conversions*

$$\begin{array}{lcl} \mathbf{let}\ p = t\ \mathbf{in}\ \mathbf{let}\ q = u\ \mathbf{in}\ e & \equiv & \mathbf{let}\ q = u\ \mathbf{in}\ \mathbf{let}\ p = t\ \mathbf{in}\ e \\ \mathbf{let}\ p = \mathbf{if}^\circ\ t & \equiv & \mathbf{if}^\circ\ t \\ \quad \mathbf{then}\ u_0 & & \mathbf{then}\ \mathbf{let}\ p = u_0\ \mathbf{in}\ e \\ \quad \mathbf{else}\ u_1 & & \mathbf{else}\ \ \mathbf{let}\ p = u_1\ \mathbf{in}\ e \\ \mathbf{in}\ e & & \end{array}$$

We write $\Gamma \vdash t \equiv u : \sigma$ if $\Gamma \vdash t, u : \sigma$ and the equation $t \equiv u$ is derivable at the type σ.

Lemma 4.2.1 (Soundness) *The equational theory is sound: if $\Gamma \vdash t \equiv u : \sigma$ then the functions $[\![\Gamma \vdash t : \sigma]\!]$ and $[\![\Gamma \vdash u : \sigma]\!]$ are extensionally equal.*

Proof. By induction on the derivation $\Gamma \vdash t \equiv u : \sigma$.

- **let**-equation

$$\mathbf{let}\ p = \mathit{val}\ \mathbf{in}\ u \qquad \equiv \qquad u\,[\mathit{val}\,/\,p]$$

Note that this equation is a bit restrictive: p can only be bound to a *value*. This is because we want to use all these equations for the language added with quantum data. The key point here is the well-known "non-cloning" property of quantum states (NIELSEN; CHUANG, 2000). Imagine that p is bound to quantum data, and that it is being used more than once in the term u. This would correspond to make copies of qubits, which is physically not realizable.

To prove soundness for that we will interpret substitution as usual. From the rules on Figure 4.1 we can derive the following substitution rule:

$$\frac{\Gamma \vdash \mathit{val} : \sigma \quad \Delta, p : \sigma \vdash u : \tau}{\Gamma \otimes \Delta \vdash u[\mathit{val}/p] : \tau}\ \text{Subs}$$

where $u[val/p]$ denotes the result of substituting val for p in u, which is interpreted as:

$$\begin{array}{l} [\![\Gamma \otimes \Delta \vdash u[\mathit{val}/p] : \tau]\!] = \\ [\![\Delta, p : \sigma \vdash u : \tau]\!] \circ ([\![\Gamma \vdash \mathit{val} : \sigma]\!] \times \mathit{id}) \circ \delta_{\Gamma,\Delta} \end{array}$$

That is, exactly as given the meaning for $\mathbf{let}\ p = val\ \mathbf{in}\ u$ in Figure 4.2.

- β-equations

1. $\mathbf{let}\ (x,y) = (t,u)\ \mathbf{in}\ e \equiv \mathbf{let}\ x = t\ \mathbf{in}\ \mathbf{let}\ y = u\ \mathbf{in}\ e$

First, the left hand side.

$$\begin{array}{rl} \text{lhs} & = [\![\Gamma \otimes \Delta \otimes \Delta' \vdash \texttt{let}\ (x,y) = (t,u)\ \texttt{in}\ e : \rho]\!] \\ & = g_e \circ (f_{(t,u)} \times id) \circ \delta_{(\Gamma\otimes\Delta),\Delta'} \end{array}$$

where

$$\begin{array}{rcl} g_e & = & [\![\Delta', x:\sigma, y:\tau \vdash u : \rho]\!] \\ f_{(t,u)} & = & [\![\Gamma \otimes \Delta \vdash (t,u) : \sigma \otimes \tau]\!] \\ & = & (f_t \times f_u) \circ \delta_{\Gamma,\Delta} \end{array}$$

where

$$\begin{array}{rcl} f_t & = & [\![\Gamma \vdash t : \sigma]\!] \\ f_u & = & [\![\Delta \vdash u : \tau]\!] \end{array}$$

Then, we have:

$$\text{lhs} \ = \ g_e \circ (((f_t \times f_u)) \circ \delta_{\Gamma,\Delta}) \times id) \circ \delta_{(\Gamma\otimes\Delta),\Delta'}$$

To the right hand side:

$$\begin{array}{rl} \text{rhs} & = [\![\Gamma \otimes \Delta \otimes \Delta' \vdash \texttt{let}\ x = t\ \texttt{in}\ \texttt{let}\ y = u\ \texttt{in}\ e : \rho]\!] \\ & = g_{let} \circ (f_t \times id) \circ \delta_{\Gamma,(\Delta\otimes\Delta')} \end{array}$$

where

$$\begin{array}{rcl} f_t & = & [\![\Gamma \vdash t : \sigma]\!] \\ g_{let} & = & [\![\Delta \otimes \Delta', x:\sigma \vdash \texttt{let}\ y = u\ \texttt{in}\ e : \rho]\!] \\ & = & g_e \circ (f_u \times id) \circ \delta_{\Delta,(\Delta',x:\sigma)} \end{array}$$

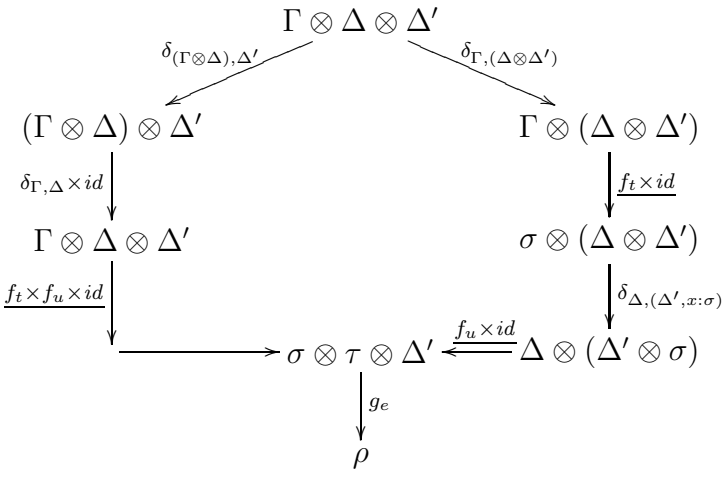
where

$$\begin{array}{rcl} f_u & = & [\![\Delta \vdash u : \tau]\!] \\ g_e & = & [\![\Delta', x:\sigma, y:\tau \vdash e : \rho]\!] \end{array}$$

Then, we have:

$$\text{rhs} \quad = g_e \circ (f_u \times id) \circ \delta_{\Delta,(\Delta',x:\sigma)} \circ (f_t \times id) \circ \delta_{\Gamma,(\Delta\otimes\Delta')}$$

Finally, because parallel composition is extensionally equal to sequential composition using identity in the extra wires, we have that lhs is extensionally equal to rhs :



2. $\mathbf{if}^\circ\ \mathit{false}\ \mathbf{then}\ t\ \mathbf{else}\ u \equiv u$

First the left hand side:

$$\begin{array}{rl} \text{lhs} & = [\![\Gamma \otimes \Delta \vdash \texttt{if}^\circ\ \mathit{false}\ \texttt{then}\ t\ \texttt{else}\ u : \sigma]\!] \\ & = (g|h) \circ (f \times id) \circ \delta_{\Gamma,\Delta} \end{array}$$

where

$$\begin{array}{rcl} f & = & [\![\bullet \vdash \mathit{false} : \mathcal{Q}_2]\!] \\ & = & \mathit{const}\ 0 \\ g & = & [\![\Delta \vdash t : \sigma]\!] \\ h & = & [\![\Delta \vdash u : \sigma]\!] \end{array}$$

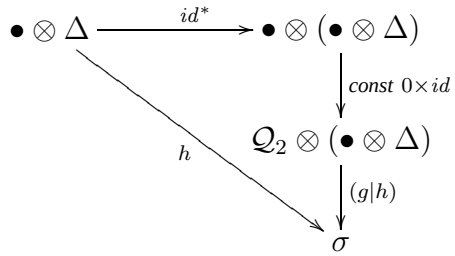
Then, using the conditional definition and because Γ is empty:

$$\text{lhs} = (g|h) \circ (\mathit{const}\ 0 \times id) \circ id^* = h$$

Now, the right hand side:

$$\text{rhs} = [\![\bullet \otimes \Delta \vdash u : \sigma]\!] = h.$$

The proof can be easily compacted in the diagram below:



3. $\mathbf{if}^\circ\ \mathit{true}\ \mathbf{then}\ t\ \mathbf{else}\ u \equiv t$

First the left hand side:

$$\begin{array}{rl} \text{lhs} & = [\![\Gamma \otimes \Delta \vdash \texttt{if}^\circ\ \mathit{true}\ \texttt{then}\ t\ \texttt{else}\ u : \sigma]\!] \\ & = (g|h) \circ (f \times id) \circ \delta_{\Gamma,\Delta} \end{array}$$

where

$$\begin{array}{rcl} f & = & [\![\bullet \vdash \mathit{true} : \mathcal{Q}_2]\!] \\ & = & \mathit{const}\ 1 \\ g & = & [\![\Delta \vdash t : \sigma]\!] \\ h & = & [\![\Delta \vdash u : \sigma]\!] \end{array}$$

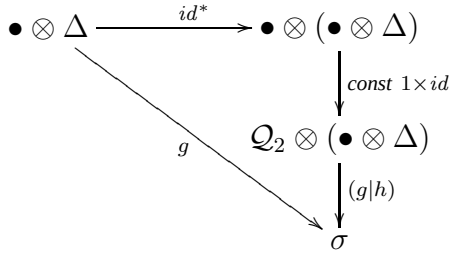
Then, using the conditional definition and because Γ is empty:

$$\text{lhs} = (g|h) \circ (\mathit{const}\ 1 \times id) \circ id^* = g$$

Now, the right hand side:

$$\text{rhs} = [\![\bullet \otimes \Delta \vdash t : \sigma]\!] = g.$$

Again, the proof can be easily compacted in the diagram below:



- η-equations

  1. $\mathbf{let}\ x = t\ \mathbf{in}\ x \equiv t$

$$\begin{array}{rl} \text{lhs} & = [\![\Gamma \otimes \Delta \vdash \texttt{let}\ x = t\ \texttt{in}\ x : \sigma]\!] \\ & = g \circ (f \times id) \circ \delta_{\Gamma,\Delta} \end{array}$$

where

$$\begin{array}{rcl} f & = & [\![\Gamma \vdash t : \sigma]\!] \\ g & = & [\![\Delta, x : \sigma \vdash x : \sigma]\!] \\ & = & [\![\bullet, x : \sigma \vdash x : \sigma]\!] \\ & = & id_* \end{array}$$

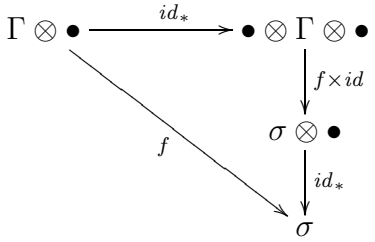
then, knowing that Δ is empty, we have:

$$\text{lhs} \quad = id_* \circ (f \times id) \circ \delta_{\Gamma,\bullet}$$

and

$$\begin{array}{rl} \text{rhs} & = [\![\Gamma \otimes \bullet \vdash t : \sigma]\!] \\ & = [\![\Gamma \vdash t : \sigma]\!] \\ & = f \end{array}$$

The proof is illustrated by the diagram:



  2. $\mathbf{let}\ (x, y) = t\ \mathbf{in}\ (x, y) \equiv t$

$$\begin{array}{rl} \text{lhs} & = [\![\Gamma \otimes \Delta \vdash \texttt{let}\ (x, y) = t\ \texttt{in}\ (x, y) : \rho]\!] \\ & = g \circ (f \times id) \circ \delta_{\Gamma,\Delta} \end{array}$$

where

$$\begin{array}{rcl} f & = & [\![\Gamma \vdash t : \sigma \otimes \tau]\!] \\ g & = & [\![\Delta, x:\sigma, y:\tau \vdash (x,y) : \rho]\!] \\ & = & [\![\bullet, x:\sigma, y:\tau \vdash (x,y) : \rho]\!] \\ & = & (g_x \times g_y) \circ \delta_{\bullet,x:\sigma,y:\tau} \end{array}$$

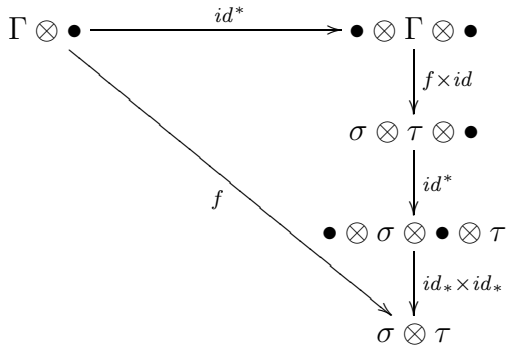
where

$$\begin{array}{rcl} g_x & = & [\![x : \sigma \vdash x : \sigma]\!] = id_* \\ g_y & = & [\![y : \tau \vdash y : \tau]\!] = id_* \end{array}$$

then

$$\text{lhs} \quad = \quad (id_* \times id_*) \circ id^* \circ (f \times id) \circ id^*$$

Using the facts that Δ is empty and that $\rho \equiv \sigma \otimes \tau$, we have:



3. $\mathbf{if}^\circ\ t\ \mathbf{then}\ \mathit{true}\ \mathbf{else}\ \mathit{false} \equiv t$

$$\begin{array}{rcl} \text{lhs} & = & [\![\Gamma \otimes \Delta \vdash \texttt{if}^\circ\ t\ \texttt{then}\ \text{true}\ \texttt{else}\ \text{false} : \mathcal{Q}_2]\!] \\ & = & (g|h) \circ (f \times id) \circ \delta_{\Gamma,\Delta} \end{array}$$

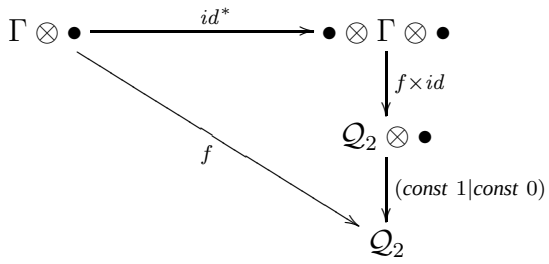
where

$$\begin{array}{rcl} f & = & [\![\Gamma \vdash t : \mathcal{Q}_2]\!] \\ g & = & [\![\bullet \vdash \text{true} : \mathcal{Q}_2]\!] \\ & = & \mathit{const}\ 1 \\ h & = & [\![\bullet \vdash \text{false} : \mathcal{Q}_2]\!] \\ & = & \mathit{const}\ 0 \end{array}$$

Then, using the conditional definition and because Δ is empty:

$$\begin{array}{rcl} \text{lhs} & = & (\mathit{const}\ 1 | \mathit{const}\ 0) \circ (f \times id) \circ id^* \\ & = & f \\ & = & [\![\Gamma \otimes \bullet \vdash t : \mathcal{Q}_2]\!] \\ & = & \text{rhs} \end{array}$$

More specifically, we have the following diagram:



- Commuting conversions

  1. $\textbf{let}\ p = t\ \textbf{in let}\ q = u\ \textbf{in}\ e$
     $\equiv \quad \textbf{let}\ q = u\ \textbf{in let}\ p = t\ \textbf{in}\ e$

$$\begin{array}{rcl} \text{lhs} & = & [\![\Gamma \otimes \Delta \otimes \Delta' \vdash \texttt{let}\ p = t\ \texttt{in}\ \texttt{let}\ q = u\ \texttt{in}\ e : \rho]\!] \\ & = & g_{let} \circ (f_t \times id) \circ \delta_{\Gamma,\Delta\otimes\Delta'} \end{array}$$

where

$$\begin{array}{rcl} f_t & = & [\![\Gamma \vdash t : \sigma]\!] \\ g_{let} & = & [\![\Delta \otimes p : \sigma, \Delta' \vdash \texttt{let}\ q = u\ \texttt{in}\ e : \rho]\!] \\ & = & g_e \circ (f_u \times id) \circ \delta_{\Delta,(p:\sigma,\Delta')} \\ & & \text{where } \begin{array}{rcl} f_u & = & [\![\Delta \vdash u : \tau]\!] \\ g_e & = & [\![p : \sigma, q : \tau, \Delta' \vdash e : \rho]\!] \end{array} \end{array}$$

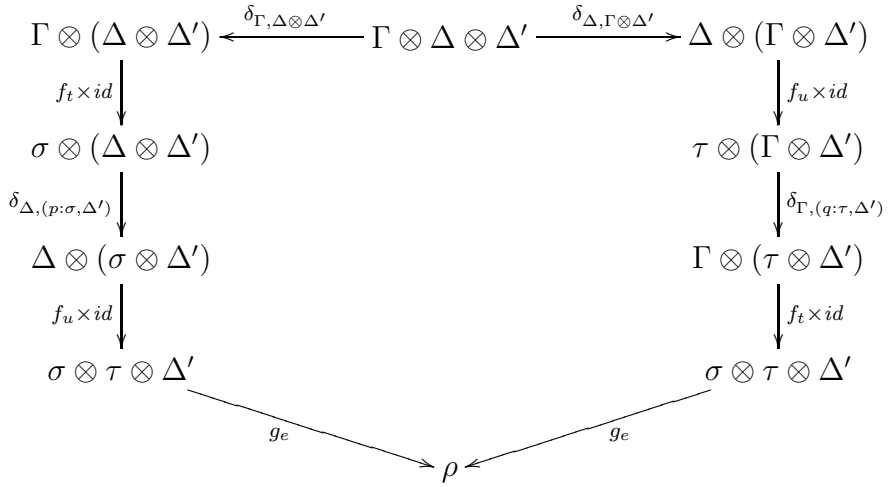
then

$$\text{lhs} \quad = \quad g_e \circ (f_u \times id) \circ \delta_{\Delta,(p:\sigma,\Delta')} \circ (f_t \times id) \circ \delta_{\Gamma,\Delta\otimes\Delta'}$$

Now,

$$\begin{array}{rcl} \text{rhs} & = & [\![\Gamma \otimes \Delta \otimes \Delta' \vdash \texttt{let}\ q = u\ \texttt{in}\ \texttt{let}\ p = t\ \texttt{in}\ e : \rho]\!] \\ & = & g_{let} \circ (f_u \times id) \circ \delta_{\Delta,\Gamma\otimes\Delta'} \\ & = & g_e \circ (f_t \times id) \circ \delta_{\Gamma,(q:\tau,\Delta')} \circ (f_u \times id) \circ \delta_{\Delta,\Gamma\otimes\Delta'} \end{array}$$

Therefore, the diagram commutes:



2. $\textbf{let}\ x = \textbf{if}^\circ\ t\ \textbf{then}\ u_0\ \textbf{else}\ u_1$
   $\textbf{in}\ e \quad \equiv$
   $\textbf{if}^\circ\ t$
   $\textbf{then let}\ x = u_0\ \textbf{in}\ e$
   $\textbf{else let}\ x = u_1\ \textbf{in}\ e$

$$\begin{aligned} \text{lhs} &= [\![\Gamma \otimes \Delta \otimes \Delta' \vdash \texttt{let}\ x = \texttt{if}^\circ\ t\ \texttt{then}\ u_o\ \texttt{else}\ u_1 \\ &\qquad \texttt{in}\ e : \tau]\!] \\ &= g_e \circ (f_{if} \times id) \circ \delta_{(\Gamma\otimes\Delta),\Delta'} \end{aligned}$$

where

$$\begin{aligned} f_{if} &= [\![\Gamma \otimes \Delta \vdash \texttt{if}^\circ\ t\ \texttt{then}\ u_o\ \texttt{else}\ u_1 : \sigma]\!] \\ &= (f_{u0}|f_{u1}) \circ (f_t \times id) \circ \delta_{\Gamma,\Delta} \\ f_t &= [\![\Gamma \vdash t : \mathcal{Q}_2]\!] \\ f_{u0} &= [\![\Delta \vdash u_0 : \sigma]\!] \\ f_{u1} &= [\![\Delta \vdash u_1 : \sigma]\!] \\ g_e &= [\![\Delta', x : \sigma \vdash e : \tau]\!] \end{aligned}$$

Therefore:

$$\begin{aligned} \text{lhs} &= g_e \circ (((f_{u0}|f_{u1}) \circ (f_t \times id) \circ \delta_{\Gamma,\Delta}) \times id) \circ \\ &\quad \delta_{(\Gamma\otimes\Delta),\Delta'} \end{aligned}$$

Now, the right hand side:

$$\begin{aligned} \text{rhs} &= [\![\Gamma \otimes \Delta \otimes \Delta' \vdash \texttt{if}^\circ\ t \\ &\qquad \texttt{then let}\ x = u_0\ \texttt{in}\ e \\ &\qquad \texttt{else let}\ x = u_1\ \texttt{in}\ e : \tau]\!] \\ &= (g_{let_1}|h_{let_2}) \circ (f_t \times id) \circ \delta_{\Gamma,(\Delta\otimes\Delta')} \end{aligned}$$

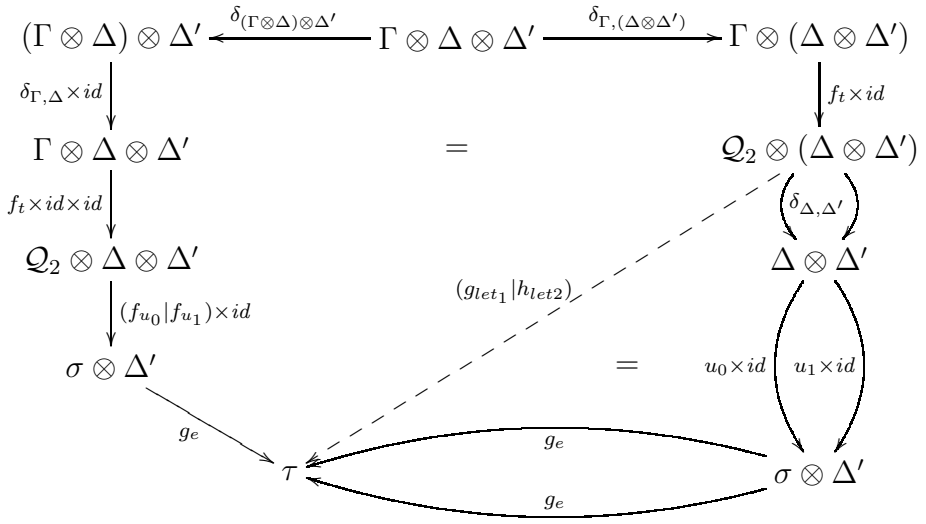
where

$$\begin{aligned} f_t &= [\![\Gamma \vdash t : \mathcal{Q}_2]\!] \\ g_{let1} &= [\![\Delta \otimes \Delta' \vdash \texttt{let}\ x = u_0\ \texttt{in}\ e : \tau]\!] \\ &= g_e \circ (f_{u0} \times id) \circ \delta_{\Delta,\Delta'} \\ f_{u0} &= [\![\Delta \vdash u_0 : \sigma]\!] \\ g_e &= [\![\Delta', x : \sigma \vdash e : \tau]\!] \\ h_{let2} &= [\![\Delta \otimes \Delta' \vdash \texttt{let}\ x = u_1\ \texttt{in}\ e : \tau]\!] \\ &= g_e \circ (f_{u1} \times id) \circ \delta_{\Delta,\Delta'} \\ f_{u1} &= [\![\Delta \vdash u_1 : \sigma]\!] \end{aligned}$$

Therefore:

$$\begin{aligned} \text{rhs} &= ((g_e \circ (f_{u0} \times id) \circ \delta_{\Delta,\Delta'})|(g_e \circ (f_{u1} \times id) \circ \delta_{\Delta,\Delta'})) \\ &\quad \circ (f_t \times id) \circ \delta_{\Gamma,(\Delta\otimes\Delta')} \end{aligned}$$

Finally, both sides of the equation are extensionally equal as shows the commutative diagram below:



□

We note that the equation

- Commuting conversion for $\mathbf{if}^\circ$

$$\begin{array}{l}\mathbf{if}^\circ\ (\mathbf{if}^\circ\ t\ \mathbf{then}\ u_0\ \mathbf{else}\ u_1)\ \mathbf{then}\ e_0\ \mathbf{else}\ e_1 \quad \equiv\\ \mathbf{if}^\circ\ t\\ \quad \mathbf{then}\ (\mathbf{if}^\circ\ u_0\ \mathbf{then}\ e_0\ \mathbf{else}\ e_1)\\ \quad \mathbf{else}\ (\mathbf{if}^\circ\ u_1\ \mathbf{then}\ e_0\ \mathbf{else}\ e_1)\end{array}$$

is derivable from the ones given above.

Proof. Assume that we have the following equivalence:

$\mathbf{let}\ x = e\ \mathbf{in}\ \mathbf{if}^\circ\ x\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 \equiv \mathbf{if}^\circ\ e\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2\ (*)$

Then we can prove the commuting conversion for $\mathbf{if}^\circ$ as follows:

$$\begin{array}{l}\quad \mathbf{if}^\circ\ (\mathbf{if}^\circ\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3)\ \mathbf{then}\ e_4\ \mathbf{else}\ e_5\\ \equiv (\mathit{assumed\ eq.}*)\\ \quad \mathbf{let}\ x = \mathbf{if}^\circ\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3\ \mathbf{in}\ \mathbf{if}^\circ\ x\ \mathbf{then}\ e_4\ \mathbf{else}\ e_5\\ \equiv (\mathit{commuting\ conversion}\ \mathbf{let}-\mathbf{if}^\circ\ (\mathit{def}.4.4.1))\\ \quad \mathbf{if}^\circ\ e_1\ \mathbf{then}\ (\mathbf{let}\ x = e_2\ \mathbf{in}\ \mathbf{if}^\circ\ x\ \mathbf{then}\ e_4\ \mathbf{else}\ e_5)\\ \qquad\quad \mathbf{else}\ (\mathbf{let}\ x = e_3\ \mathbf{in}\ \mathbf{if}^\circ\ x\ \mathbf{then}\ e_4\ \mathbf{else}\ e_5)\\ \equiv (\mathit{assumed\ eq\ twice})\\ \equiv \mathbf{if}^\circ\ e_1\ \mathbf{then}\ (\mathbf{if}^\circ\ e_2\ \mathbf{then}\ e_4\ \mathbf{else}\ e_5)\\ \qquad\quad \mathbf{else}\ (\mathbf{if}^\circ\ e_3\ \mathbf{then}\ e_4\ \mathbf{else}\ e_5)\end{array}$$

So really we only need to prove (*). We do the proof by cases on e:

- If e is a value then both sides are equivalent to: $\mathbf{if}^\circ\ e\ \mathbf{then}\ e_1\ \mathbf{then}\ e_2$.
- If $e = (e_3, e_4)$, then the equation is not well-typed.
- If $e = (\mathbf{let}\ p = e_3\ \mathbf{in}\ e_4)$, then the lhs is:

$$\begin{array}{l}\quad \mathbf{let}\ x = (\mathbf{let}\ p = e_3\ \mathbf{in}\ e_4)\ \mathbf{in}\ \mathbf{if}^\circ\ x\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2\\ \equiv (\eta\ \mathit{equation\ for}\ \mathbf{if}^\circ)\\ \quad \mathbf{let}\ x = (\mathbf{if}^\circ\ (\mathbf{let}\ p = e_3\ \mathbf{in}\ e_4)\ \mathbf{then}\ \mathit{true}\ \mathbf{else}\ \mathit{false})\end{array}$$

$$
\begin{array}{l}
\quad \textbf{in if}^\circ\ x\ \textbf{then}\ e_1\ \textbf{else}\ e_2 \\
\equiv (\textit{commuting conversion for}\ \textbf{let} - \textbf{if}^\circ) \\
\quad \textbf{if}^\circ\ (\textbf{let}\ p = e_3\ \textbf{in}\ e_4) \\
\qquad \textbf{then}\ (\textbf{let}\ x = \textit{true}\ \textbf{in if}^\circ\ x\ \textbf{then}\ e_1\ \textbf{else}\ e_2) \\
\qquad \textbf{else}\ (\textbf{let}\ x = \textit{false}\ \textbf{in if}^\circ\ x\ \textbf{then}\ e_1\ \textbf{else}\ e_2) \\
\equiv (\beta\ \textit{equations}) \\
\quad \textbf{if}^\circ\ (\textbf{let}\ p = e_3\ \textbf{in}\ e_4)\ \textbf{then}\ e_1\ e_2 \\
\equiv \textit{rhs}
\end{array}
$$

- If $e = \textbf{if}^\circ\ t\ \textbf{then}\ u\ \textbf{else}\ u'$, then the lhs is:

$$
\begin{array}{l}
\textbf{let}\ x = (\textbf{if}^\circ\ t\ \textbf{then}\ u\ \textbf{else}\ u')\ \textbf{in if}^\circ\ x\ \textbf{then}\ e_1\ \textbf{else}\ e_2 \\
\equiv (\eta\ \textit{equation for}\ \textbf{if}^\circ) \\
\textbf{let}\ x = \textbf{if}^\circ\ (\textbf{if}^\circ\ t\ \textbf{then}\ u\ \textbf{else}\ u') \\
\quad \textbf{then}\ \textit{true}\ \textbf{else}\ \textit{false} \\
\textbf{in if}^\circ\ x\ \textbf{then}\ e_1\ \textbf{else}\ e_2 \\
\equiv (\textit{commuting conversion for}\ \textbf{let} - \textbf{if}^\circ) \\
\textbf{if}^\circ\ (\textbf{if}^\circ\ t\ \textbf{then}\ u\ \textbf{else}\ u') \\
\quad \textbf{then}\ (\textbf{let}\ x = \textit{true}\ \textbf{in if}^\circ\ x\ \textbf{then}\ e_1\ \textbf{else}\ e_2) \\
\quad \textbf{else}\ (\textbf{let}\ x = \textit{false}\ \textbf{in if}^\circ\ x\ \textbf{then}\ e_1\ \textbf{else}\ e_2) \\
\equiv (\beta\ \textit{equations}) \\
\textbf{if}^\circ\ (\textbf{if}^\circ\ t\ \textbf{then}\ u\ \textbf{else}\ u') \\
\quad \textbf{then}\ e_1 \\
\quad \textbf{else}\ e_2 \\
\equiv \textit{rhs}
\end{array}
$$

□

4.2.7 Completeness of the Classical Theory

The equational theory is *complete* in a strong technical sense: as we prove in the remainder of the section, any equivalence implied by the semantics is derivable in the theory. The proof technique is based on current work by Thorsten Altenkirch with Tarmo Uustalu (ALTENKIRCH; UUSTALU, 2004). The proof we present extends and simplifies the method presented in that work.

4.2.7.1 Proof Technique

The ultimate goal is to prove the following statement.

Proposition 4.2.1 (Completeness) *If* $[\![\Gamma \vdash t : \sigma]\!]$ *and* $[\![\Gamma \vdash u : \sigma]\!]$ *are extensionally equal, then we can derive* $\Gamma \vdash t \equiv u : \sigma$.

In model theory (BRIDGE, 1997), in general one starts with a semantic structure and then from that defines the language (i.e., the syntactic structure). In this context, one wants to define a *representative* or *inverse* function to show how semantic objects are represented in the syntactic structure.

However, in programming languages design, it is natural to follow a contrary path. We start with a syntactic structure (the language) and then give the semantics using a meaning or evaluation function as we do here.

Therefore, in order to prove the statement above, we define a function q^{σ}_{Γ} which inverts evaluation by producing a canonical syntactical representative. In fact, we define the function q^{σ}_{Γ} such that it maps a denotation $[\![\Gamma \vdash t : \sigma]\!]$ to the normal form of t.

Definition 4.2.2 *The* normal form *of t is given by $nf^{\sigma}_{\Gamma}(t) = q^{\sigma}_{\Gamma}([\![\Gamma \vdash t : \sigma]\!])$.*

The normal form is well-defined: given an equation $\Gamma \vdash t \equiv u : \sigma$, we know by soundness that $[\![\Gamma \vdash t : \sigma]\!]$ is extensionally equal $[\![\Gamma \vdash u : \sigma]\!]$ and hence we get that $\mathrm{nf}^{\sigma}_{\Gamma}(t) = \mathrm{nf}^{\sigma}_{\Gamma}(u)$. If we can now prove that the syntactic theory can prove that every term is equal to its normal form, then we can prove the main completeness result. Indeed given the following lemma, we can prove completeness.

Lemma 4.2.2 (Inversion) *The equation $\Gamma \vdash nf^{\sigma}_{\Gamma}(t) \equiv t : \sigma$ is derivable.*

Proof. Proof of Proposition 4.2.1 (Completeness) We have:

$$\begin{array}{rl} \Gamma \vdash t \equiv q^{\sigma}_{\Gamma}[\![\Gamma \vdash t : \sigma]\!] : \sigma & \text{by inversion} \\ \Gamma \vdash q^{\sigma}_{\Gamma}[\![\Gamma \vdash t : \sigma]\!] \equiv q^{\sigma}_{\Gamma}[\![\Gamma \vdash u : \sigma]\!] : \sigma & \text{by assumption} \\ \Gamma \vdash q^{\sigma}_{\Gamma}[\![\Gamma \vdash u : \sigma]\!] \equiv u : \sigma & \text{by inversion} \end{array}$$

□

To summarise we can establish completeness by defining a function q^{σ}_{Γ} that inverts evaluation and that satisfies Inversion Lemma 4.2.2. Essentially, the approach can be analyzed in Figure 4.3. Consider we have two different syntactic terms (t_1, t_2) and that they are given the same denotation (f). Then, we can use the function which inverts evaluation (q) to map the meaning back to the normal form of the terms (nf1). Finally, we have completeness if we can derive the dashed arrows (in the syntax) from the equational theory.

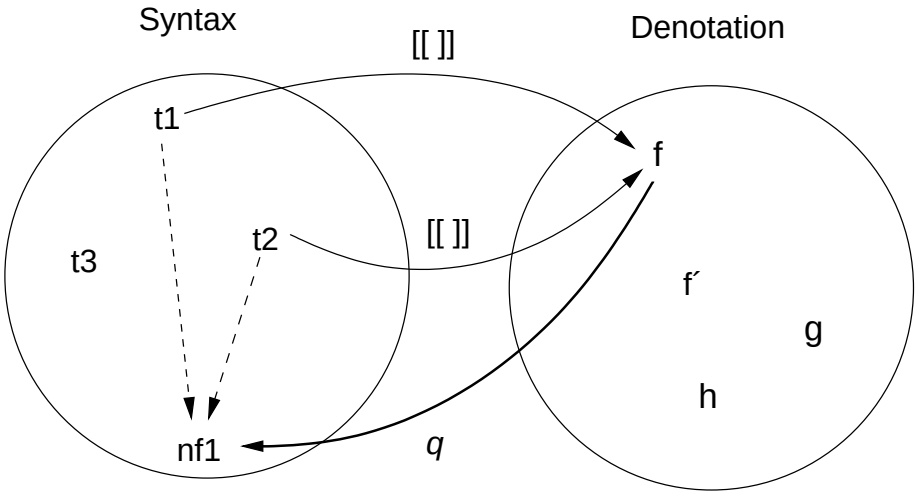


Figure 4.3: Diagram for completeness proof technique.

4.2.7.2 Adequacy

We begin by defining a family of functions q^{σ} ("*quote*") which invert the evaluation of *closed* terms and prove a special case of the inversion lemma for closed terms, called *adequacy*. These functions and the adequacy result are then used in the next section to invert the evaluation of open terms and prove the general inversion lemma.

Definition 4.2.3 *The* syntactic representations of denotations *is given by:*

$$q^{\sigma} \in [\![\sigma]\!] \to \mathrm{Val}^{\mathrm{C}}\sigma$$

defined by induction over σ*:*

$$\begin{array}{rcl} q^{\mathcal{Q}_1}\,0 & = & () \\ q^{\mathcal{Q}_2}\,0 & = & \mathrm{false} \\ q^{\mathcal{Q}_2}\,1 & = & \mathrm{true} \\ q^{\sigma\otimes\tau}\,(a,b) & = & (q^{\sigma}\,a, q^{\tau}\,b) \end{array}$$

The version of the inversion lemma for closed terms is called *adequacy*. It guarantees that the equational theory is rich enough to equate every closed term with its final observable value.

Remark 4.2.1 *Note that* $e \in [\![\Gamma \otimes \Delta]\!]$ *is different from* $g \in [\![\Gamma]\!]$ *and* $d \in [\![\Delta]\!]$*. But* $\delta_{\Gamma,\Delta}e = (g,d)$*. For instance, consider* $\Gamma = \bullet, x : \mathcal{Q}_1,\ \Delta = \bullet, x : \mathcal{Q}_1, y : \mathcal{Q}_1$ *then* $\Gamma \otimes \Delta = \bullet, x : \mathcal{Q}_1, y : \mathcal{Q}_1$*.*

Lemma 4.2.3 *For* $e \in [\![\Gamma \otimes \Delta]\!]$ *and* $g \in [\![\Gamma]\!],\ d \in [\![\Delta]\!]$*, such that* $\delta_{\Gamma,\Delta}e = (g,d)$*, then*

- *For* $\mathbf{let}$

$$\begin{array}{l} \textbf{\textit{let}}^*\ \Gamma \otimes \Delta = q^{\Gamma\otimes\Delta}e\ \mathbf{in}\ \mathbf{let}\ x = t\ \mathbf{in}\ u \\ \equiv \\ \mathbf{let}\ x = (\textbf{\textit{let}}^*\ \Gamma = q^{\Gamma}g\ \mathbf{in}\ t)\ \mathbf{in}\ \textbf{\textit{let}}^*\ \Delta = q^{\Delta}d\ \mathbf{in}\ u. \end{array}$$

- *For product*

$$\begin{array}{l} \textbf{\textit{let}}^*\ \Gamma \otimes \Delta = q^{\Gamma\otimes\Delta}e\ \mathbf{in}\ (t,u) \\ \equiv \\ (\textbf{\textit{let}}^*\ \Gamma = q^{\Gamma}(g)\ \mathbf{in}\ t, \textbf{\textit{let}}^*\ \Delta = q^{\Delta}(d)\ \mathbf{in}\ u). \end{array}$$

Proof. The proof is by induction over the definition of $\otimes$ on contexts.

- Base Case: $\bullet \otimes \Delta = \Delta$
  - For $\mathbf{let}$: if $d \in [\![d]\!]$, and $\delta_{\bullet,\Delta}d = id^*d = (0,d)$, then

$$\begin{array}{rl} lhs & = \mathbf{let}^*\ \bullet \otimes \Delta = q^{\bullet\otimes\Delta}d\ \mathbf{in}\ \mathbf{let}\ x = t\ \mathbf{in}\ u \\ & \equiv (\text{by } \otimes) \\ & \mathbf{let}^*\ \Delta = q^{\Delta}d\ \mathbf{in}\ \mathbf{let}\ x = t\ \mathbf{in}\ u \end{array}$$

$$\begin{array}{rl} rhs & = \mathbf{let}\ x = (\mathbf{let}^*\ \bullet = q^{\bullet}0\ \mathbf{in}\ t)\ \mathbf{in}\ \mathbf{let}^*\ \Delta = q^{\Delta}d\ \mathbf{in}\ u \\ & \equiv (\text{by } \mathbf{let}^*) \\ & \mathbf{let}\ x = t\ \mathbf{in}\ \mathbf{let}^*\ \Delta = q^{\Delta}d\ \mathbf{in}\ u \\ & \equiv (\text{by } Commuting) \\ & \mathbf{let}^*\ \Delta = q^{\Delta}d\ \mathbf{in}\ \mathbf{let}\ x = t\ \mathbf{in}\ u \end{array}$$

  - For *product*: if $d \in [\![d]\!]$, and $\delta_{\bullet,\Delta}d = id^*d = (0,d)$, then

$$lhs\ = \mathbf{let}^*\ \Delta = q^{\Delta}d\ \mathbf{in}\ (t,u)$$

$$\begin{array}{rl} rhs & = (\mathbf{let}^*\ \bullet = q^{\bullet}(0)\ \mathbf{in}\ t, \mathbf{let}^*\ \Delta = q^{\Delta}(d)\ \mathbf{in}\ u) \\ & \equiv (t, \mathbf{let}^*\ \Delta = q^{\Delta}(d)\ \mathbf{in}\ u) \\ & \equiv (\text{by } \mathbf{let} \text{ case above}) \\ & \mathbf{let}^*\ \Delta = q^{\Delta}d\ \mathbf{in}\ (t,u) \end{array}$$

- Induction hypothesis: assume the lemma holds for $(\Gamma \otimes \Delta)$, such that $e' \in [\![\Gamma \otimes \Delta]\!]$ and $g \in [\![\Gamma]\!],\ d \in [\![\Delta]\!]$, such that $\delta_{\Gamma,\Delta} e' = (g, d)$.

- Induction step 1: $(\Gamma, x : \sigma) \otimes \Delta = (\Gamma \otimes \Delta), x : \sigma$ if $x \notin dom(\Delta)$

  - For **let**: if $e \in [\![(\Gamma \otimes \Delta), x : \sigma]\!]$, and $\delta_{(\Gamma,x:\sigma),\Delta} e = ((g, s), d)$, then

$$lhs \ = \textbf{let}^*\ (\Gamma \otimes \Delta), x : \sigma = q^{(\Gamma\otimes\Delta),x:\sigma} e \ \textbf{in let}\ x_l = t\ \textbf{in}\ u.$$

$$\begin{aligned}
rhs \ &= \textbf{let}\ x_l = (\textbf{let}^*\ \Gamma, x : \sigma = q^{\Gamma,x:\sigma}(g, s)\ \textbf{in}\ t)\ \textbf{in let}^*\ \Delta = q^{\Delta} d\ \textbf{in}\ u \\
&\equiv \text{(by } \textbf{let}^*\text{)} \\
&\ \textbf{let}\ x_l = (\textbf{let}\ (x_r, x) = (q^{\Gamma} g, q^{\sigma} s)\ \textbf{in let}^*\ \Gamma = x_r\ \textbf{in}\ t) \\
&\ \textbf{in let}^*\ \Delta = q^{\Delta} d\ \textbf{in}\ u \\
&\equiv \text{(by } \beta \text{ equation)} \\
&\ \textbf{let}\ x_l = (\textbf{let}\ x = q^{\sigma} s\ \textbf{in let}^*\ \Gamma = q^{\Gamma} g\ \textbf{in}\ t) \\
&\ \textbf{in let}^*\ \Delta = q^{\Delta} d\ \textbf{in}\ u \\
&\equiv \text{(by commuting conversion)} \\
&\ \textbf{let}\ x_l = (\textbf{let}^*\ \Gamma = q^{\Gamma} g\ \textbf{in let}\ x = q^{\sigma} s\ \textbf{in}\ t) \\
&\ \textbf{in let}^*\ \Delta = q^{\Delta} d\ \textbf{in}\ u \\
&\equiv \text{(by hypothesis)} \\
&\textbf{let}^*\ (\Gamma \otimes \Delta) = q^{(\Gamma\otimes\Delta)} e' \ \textbf{in let}\ x_l = (\textbf{let}\ x = q^{\sigma} s\ \textbf{in}\ t) \\
&\textbf{in}\ u
\end{aligned}$$

  - For *product*: if $e \in [\![(\Gamma \otimes \Delta), x : \sigma]\!]$, and $\delta_{(\Gamma,x:\sigma),\Delta} e = ((g, s), d)$, then

$$lhs \ = \textbf{let}^*\ (\Gamma \otimes \Delta), x : \sigma = q^{(\Gamma\otimes\Delta),x:\sigma} e\ \textbf{in}\ (t, u)$$

$$\begin{aligned}
rhs \ &= (\textbf{let}^*\ \Gamma, x : \sigma = q^{\Gamma,x:\sigma}(g, s)\ \textbf{in}\ t,\ \textbf{let}^*\ \Delta = q^{\Delta} d\ \textbf{in}\ u) \\
&\equiv \text{(by } \textbf{let}^*\text{)} \\
&(\textbf{let}\ (x_r, x) = (q^{\Gamma} g, q^{\sigma} s)\ \textbf{in let}^*\ \Gamma = x_r\ \textbf{in}\ t,\ \textbf{let}^*\ \Delta = q^{\Delta} d\ \textbf{in}\ u) \\
&\equiv \text{(by } \beta \text{ equation)} \\
&(\textbf{let}\ x = q^{\sigma} s\ \textbf{in let}^*\ \Gamma = q^{\Gamma} g\ \textbf{in}\ t,\ \textbf{let}^*\ \Delta = q^{\Delta} d\ \textbf{in}\ u) \\
&\equiv \text{(by commuting conversion)} \\
&(\textbf{let}^*\ \Gamma = q^{\Gamma} g\ \textbf{in let}\ x = q^{\sigma} s\ \textbf{in}\ t,\ \textbf{let}^*\ \Delta = q^{\Delta} d\ \textbf{in}\ u) \\
&\equiv \text{(by hypothesis)} \\
&\textbf{let}^*\ \Gamma \otimes \Delta = q^{\Gamma\otimes\Delta} e' \ \textbf{in}\ (\textbf{let}\ x = q^{\sigma} s\ \textbf{in}\ t, u) \\
&\equiv \text{(by } \textbf{let}^*\text{)} \\
&\textbf{let}^*\ \Gamma \otimes \Delta = q^{\Gamma\otimes\Delta} e' \ \textbf{in}\ (\textbf{let}^*\ \bullet, x : \sigma = ((), q^{\sigma} s)\ \textbf{in}\ t, u) \\
&\equiv \text{(by let case)} \\
&\textbf{let}^*\ \bullet, x : \sigma \otimes (\Gamma \otimes \Delta) = q^{\bullet, x:\sigma\otimes(\Gamma\otimes\Delta)} e\ \textbf{in}\ (t, u) \\
&\equiv \text{(by } \otimes \text{ on contexts)} \\
&\textbf{let}^*(\Gamma \otimes \Delta), x : \sigma = q^{(\Gamma\otimes\Delta),x:\sigma} e\ \textbf{in}\ (t, u)
\end{aligned}$$

- Induction step 2: $(\Gamma, x : \sigma) \otimes (\Delta, x : \sigma) = (\Gamma \otimes \Delta), x : \sigma$.

  - For **let**: if $e \in [\![(\Gamma \otimes \Delta), x : \sigma]\!]$, and $\delta_{(\Gamma,x:\sigma),(\Delta,x:\sigma)} e = ((g, s), (d, s))$, then

$$lhs \;\; = \mathbf{let}^* \; (\Gamma \otimes \Delta), x : \sigma = q^{(\Gamma\otimes\Delta),x:\sigma} e \; \mathbf{in} \; \mathbf{let} \; x_l = t \; \mathbf{in} \; u.$$

$$\begin{aligned}
rhs \;\; &= \mathbf{let} \; x_l = (\mathbf{let}^* \; \Gamma, x : \sigma = q^{\Gamma,x:\sigma}(g, s) \; \mathbf{in} \; t) \\
&\quad \mathbf{in} \; \mathbf{let}^* \; \Delta, x : \sigma = q^{\Delta,x:\sigma}(d, s) \; \mathbf{in} \; u \\
&\equiv (\text{by } \mathbf{let}^*) \\
&\; \mathbf{let} \; x_l = (\mathbf{let} \; (x_r, x) = (q^\Gamma g, q^\sigma s) \; \mathbf{in} \; \mathbf{let}^* \; \Gamma = x_r \; \mathbf{in} \; t) \\
&\; \mathbf{in} \; \mathbf{let} \; (x_s, x) = (q^\Delta d, q^\sigma s) \; \mathbf{in} \; \mathbf{let}^* \; \Delta = x_s \; \mathbf{in} \; u \\
&\equiv (\text{by } \beta \text{ equation}) \\
&\; \mathbf{let} \; x_l = (\mathbf{let} \; x = q^\sigma s \; \mathbf{in} \; \mathbf{let}^* \; \Gamma = q^\Gamma g \; \mathbf{in} \; t) \\
&\; \mathbf{in} \; \mathbf{let} \; x = q^\sigma s \; \mathbf{in} \; \mathbf{let}^* \; \Delta = q^\Delta d \; \mathbf{in} \; u \\
&\equiv (\text{by commuting conversion}) \\
&\; \mathbf{let} \; x_l = (\mathbf{let}^* \; \Gamma = q^\Gamma g \; \mathbf{in} \; \mathbf{let} \; x = q^\sigma s \; \mathbf{in} \; t) \\
&\; \mathbf{in} \; \mathbf{let}^* \; \Delta = q^\Delta d \; \mathbf{in} \; \mathbf{let} \; x = q^\sigma s \; \mathbf{in} \; u \\
&\equiv (\text{by hypothesis}) \\
&\mathbf{let}^* \; (\Gamma \otimes \Delta) = q^{(\Gamma\otimes\Delta)} e' \; \mathbf{in} \; \mathbf{let} \; x_l = (\mathbf{let} \; x = q^\sigma s \; \mathbf{in} \; t) \\
&\; \mathbf{in} \; \mathbf{let} \; x = q^\sigma s \; \mathbf{in} \; u \\
&\equiv (\text{by } \mathbf{let}^*) \\
&\mathbf{let}^* \; (\Gamma \otimes \Delta) = q^{(\Gamma\otimes\Delta)} e' \; \mathbf{in} \; \mathbf{let} \; x_l = (\mathbf{let}^* \; \bullet, x : \sigma = ((), q^\sigma s) \; \mathbf{in} \; t) \\
&\; \mathbf{let}^* \; \bullet, x : \sigma = ((), q^\sigma s) \; \mathbf{in} \; u \\
&\equiv (\text{by commuting conversion}) \\
&\mathbf{let} \; x_l = (\mathbf{let}^* \; \bullet, x : \sigma = ((), q^\sigma s) \;\; \mathbf{in} \; t) \; \mathbf{in} \; \mathbf{let}^* \; (\Gamma \otimes \Delta) = q^{(\Gamma\otimes\Delta)} e' \\
&\; \mathbf{let}^* \; \bullet, x : \sigma = ((), q^\sigma s) \; \mathbf{in} \; u \\
&\equiv (\text{by base case}) \\
&\mathbf{let}^* \; \bullet, x : \sigma \otimes (\Gamma \otimes \Delta) = q^{\bullet,x:\sigma\otimes(\Gamma\otimes\Delta)} e \; \mathbf{in} \; \mathbf{let} \; x_l = t \; \mathbf{in} \; u \\
&\equiv (\text{by } \otimes \text{ on contexts}) \\
&\mathbf{let}^* \; (\Gamma \otimes \Delta), x : \sigma = q^{(\Gamma\otimes\Delta),x:\sigma} e \; \mathbf{in} \; \mathbf{let} \; x_l = t \; \mathbf{in} \; u
\end{aligned}$$

Lemma 4.2.4 (Adequacy) *The equation* $\vdash q^\sigma([\![\vdash t : \sigma]\!] \; 0) \equiv t : \sigma$ *is derivable.*

Proof. The proof is by induction over $\vdash \; t \; : \; \sigma$. Therefore, we need to generalize the statement, that is, during the proof we encounter open terms that must be closed before they are "quoted." First we have to extend q to contexts by identifying a context with the product of all its components, i.e., we use $q^{|\Gamma|}$. The idea we use in the proof is: given an open term $\Gamma \vdash t : \sigma$, and a semantic value representing an environment, $g \in [\![\Gamma]\!]$, we can generate a closed instance of the term by calculating $q^{|\Gamma|}(g)$ and then generating nested $\mathbf{let}$-expressions. For convenience, we use $\mathbf{let}^*$ which is defined in 4.2.3.

Note that we could have done this using iterated substitutions as follows: given $\Gamma = x_1 \; : \; \sigma_1, x_2 \; : \; \sigma_2, .., x_n \; : \; \sigma_n$ then forall $v_1 \in [\![\sigma_1]\!], v_2 \in [\![\sigma_2]\!], \ldots, v_n \in [\![\sigma_n]\!]$. Then $q^\sigma([\![t]\!](x_1 \mapsto v_1, \ldots, x_n \mapsto v_n)) = t[x_1 = q^{\sigma_1} v_1, \ldots, x_n = q^{\sigma_n} v_n]$ This would have worked in the classical case but not in the quantum case where there is a global *entangled* quantum state. Imagine what happens if $\Gamma = x : \mathcal{Q}_2, y : \mathcal{Q}_2$ with $\mathbf{let} \; (x, y) = (0, 0) + (1, 1) \; \mathbf{in}$.... The idea of $\mathbf{let}^*$ is to maintain the global state with "pointers" to specific values.

So in fact the statement to prove by induction is the following:

$$\text{If } g \in [\![\Gamma]\!] \text{ then } \; \vdash q^\sigma([\![\Gamma \vdash t : \sigma]\!] \; g) \equiv \mathbf{let}^* \; \Gamma = q^{|\Gamma|}(g) \mathbf{in} \; t : \sigma$$

1. $\bullet \vdash () : \mathcal{Q}_1$. Then, we need to show

$$q^{\mathcal{Q}_1}([\![\bullet \vdash () : \mathcal{Q}_1]\!]0) \equiv \textbf{let}^* \bullet = q^{\mathcal{Q}_1}(0) \textbf{ in} () : \mathcal{Q}_1.$$

The lhs is equal to:

$$\begin{array}{rcl} q^{\mathcal{Q}_1}([\![\bullet \vdash () : \mathcal{Q}_1]\!]0) & \equiv & (\text{by the meaning function in Figure } 4.2) \\ & & q^{\mathcal{Q}_1}(\textit{const}\ 0\ 0) \\ & \equiv & (\text{by } \textit{const}\ 0) \\ & & q^{\mathcal{Q}_1}(0) \\ & \equiv & (\text{by } q) \\ & & () \end{array}$$

The rhs is equal to:

$$\begin{array}{rcl} \textbf{let}^* \bullet = q^{\mathcal{Q}_1}(0) \textbf{ in} () : \mathcal{Q}_1 & \equiv & (\text{by } q) \\ & & \textbf{let}^* \bullet = () \textbf{ in } () : \mathcal{Q}_1 \\ & \equiv & (\text{by } \textbf{let}^*) \\ & & () \end{array}$$

2. $\bullet \vdash$ false $: \mathcal{Q}_2$. Then, we need to show

$$q^{\mathcal{Q}_2}([\![\bullet \vdash \text{false} : \mathcal{Q}_2]\!]0) \equiv \textbf{let}^* \bullet = q^{\mathcal{Q}_2}(0) \textbf{ in false} : \mathcal{Q}_2.$$

The lhs is equal to:

$$\begin{array}{rcl} q^{\mathcal{Q}_2}([\![\bullet \vdash \text{false} : \mathcal{Q}_2]\!]0) & \equiv & (\text{by the meaning function in Figure } 4.2) \\ & & q^{\mathcal{Q}_2}(\textit{const}\ 0\ 0) \\ & \equiv & (\text{by } \textit{const}\ 0) \\ & & q^{\mathcal{Q}_2}(0) \\ & \equiv & (\text{by } q) \\ & & \text{false} \end{array}$$

The rhs is equal to:

$$\begin{array}{rcl} \textbf{let}^* \bullet = q^{\mathcal{Q}_2}(0) \textbf{ in false } : \mathcal{Q}_2 & \equiv & (\text{by } q) \\ & & \textbf{let}^* \bullet = \text{ false } \textbf{in false} : \mathcal{Q}_2 \\ & \equiv & (\text{by } \textbf{let}^*) \\ & & \text{false} \end{array}$$

3. $\bullet \vdash$ true $: \mathcal{Q}_2$. Then, we need to show

$$q^{\mathcal{Q}_2}([\![\bullet \vdash \text{true } : \mathcal{Q}_2]\!]0) \equiv \textbf{let}^* \bullet = q^{\mathcal{Q}_2}(0) \textbf{ in true } : \mathcal{Q}_2.$$

The lhs is equal to:

$$
\begin{array}{rcl}
q^{\mathcal{Q}_2}([\![\bullet \vdash \text{true} \ : \mathcal{Q}_2]\!]0) & \equiv & \text{(by the meaning function in Figure } 4.2) \\
 & & q^{\mathcal{Q}_2}(\textit{const}\ 1\ 0) \\
 & \equiv & \text{(by } \textit{const}\ 1) \\
 & & q^{\mathcal{Q}_2}(1) \\
 & \equiv & \text{(by } q) \\
 & & \text{true}
\end{array}
$$

The rhs is equal to:

$$
\begin{array}{rcl}
\mathbf{let}^*\ \bullet = q^{\mathcal{Q}_2}(0)\ \mathbf{in}\ \text{true} \ : \mathcal{Q}_2 & \equiv & \text{(by } q) \\
 & & \mathbf{let}^*\ \bullet = \ \text{true}\ \mathbf{in}\ \text{true} \ : \mathcal{Q}_2 \\
 & \equiv & \text{(by } \mathbf{let}^*) \\
 & & \text{true}
\end{array}
$$

4. $x : \sigma \vdash x : \sigma$. Then, we need to show

$$q^{\sigma}([\![x : \sigma \vdash x : \sigma]\!]g) \equiv \mathbf{let}^*\ x : \sigma = q^{\sigma}(g)\ \mathbf{in}\ x : \sigma.$$

Actually, because the empty context is omitted if the context is non-empty, then we really need to show that if $g \in [\![\bullet, x : \sigma]\!] = \{0\} \times [\![\sigma]\!]$ then

$$q^{\sigma}([\![\bullet, x : \sigma \vdash x : \sigma]\!](0, s)) \equiv \mathbf{let}^*(\bullet, x : \sigma) = q^{\mathcal{Q}_1 \otimes \sigma}(0, s)\ \mathbf{in}\ x : \sigma.$$

$$
\begin{array}{rcl}
q^{\sigma}([\![\bullet, x : \sigma \vdash x : \sigma]\!](0, s)) & \equiv & \text{(by Figure } 4.2) \\
 & & q^{\sigma}(id_*\ (0, s)) \\
 & \equiv & q^{\sigma}(s)
\end{array}
$$

The rhs is:

$$
\begin{array}{rcl}
\mathbf{let}^*(\bullet, x : \ \sigma) = q^{\mathcal{Q}_1 \otimes \sigma}(0, s)\ \mathbf{in}\ x & \equiv & \text{(by } q) \\
\mathbf{let}^*(\bullet, x : \sigma) = ((), q^{\sigma} s)\ \mathbf{in}\ x & \equiv & \text{(by } \mathbf{let}^*) \\
\mathbf{let}(x_r, x) = ((), q^{\sigma} s)\ \mathbf{in}\ \mathbf{let}^*\ \bullet = x_r\ \mathbf{in}\ x & \equiv & \text{(by } \mathbf{let}^*) \\
\mathbf{let}(x_r, x) = ((), q^{\sigma} s)\ \mathbf{in}\ x & \equiv & \text{(by } \beta \text{ eq.)} \\
\mathbf{let}\ x_r = ()\ \mathbf{in}\ \mathbf{let}\ x = q^{\sigma} s\ \mathbf{in}\ x & \equiv & \text{(by } \mathbf{let}) \\
\mathbf{let}\ x = q^{\sigma} s\ \mathbf{in}\ x\& \equiv & \text{(by } \eta) & \\
q^{\sigma} s & &
\end{array}
$$

5. $\Gamma \otimes \Delta \vdash \texttt{let}\ x = t\ \texttt{in}\ u : \tau$. We want to show that: if $e \in [\![\Gamma \otimes \Delta]\!]$ then

$$
\begin{array}{l}
q^{\tau}([\![\Gamma \otimes \Delta \vdash \mathbf{let}\ x = t\ \mathbf{in}\ u : \tau]\!]e) \\
\equiv \\
\mathbf{let}^*\ \Gamma \otimes \Delta = q^{\Gamma \otimes \Delta} e\ \mathbf{in}\ \mathbf{let}\ x = t\ \mathbf{in}\ u.
\end{array}
$$

By induction hypothesis we have: if $g \in [\![\Gamma]\!]$, $s \in [\![\sigma]\!]$ and $d \in [\![\Delta]\!]$, such that $\delta_{\Gamma,\Delta} e = (g, d)$, then

(a) $q^{\sigma}([\![\Gamma \vdash t : \sigma]\!]g) = \mathbf{let}^*\ \Gamma = q^{\Gamma}(g)\ \mathbf{in}\ t$.

(b) $q^\tau([\![\Delta, x:\sigma \vdash u:\tau]\!](d,s) = \textbf{let}^*\ \Delta, x:\sigma = q^{\Delta,\sigma}(d,s)\ \textbf{in}\ u.$

Now developing the lhs:

$q^\tau([\![\Gamma \otimes \Delta \vdash \textbf{let}\ x = t\ \textbf{in}\ u:\tau]\!]e)$
$\equiv$ (by the meaning function in Figure 4.2)
$q^\tau([\![\Delta, x:\sigma \vdash u:\tau]\!] \circ ([\![\Gamma \vdash t:\sigma]\!] \times id) \circ \delta_{\Gamma,\Delta} e)$
$\equiv$ (by simplification)
$q^\tau([\![\Delta, x:\sigma \vdash u:\tau]\!](d, [\![\Gamma \vdash t:\sigma]\!]g))$
$\equiv$ (by hypothesis b.)
$\textbf{let}^*\ \Delta, x:\sigma = q^{\Delta,\sigma}(d, [\![\Gamma \vdash t:\sigma]\!]g)\ \textbf{in}\ u$
$\equiv$ (by q)
$\textbf{let}^*\ \Delta, x:\sigma = (q^\Delta d, q^\sigma[\![\Gamma \vdash t:\sigma]\!]g)\ \textbf{in}\ u$
$\equiv$ (by hypothesis a.)
$\textbf{let}^*\ \Delta, x:\sigma = (q^\Delta d, \textbf{let}^*\ \Gamma = q^\Gamma(g)\ \textbf{in}\ t)\ \textbf{in}\ u$
$\equiv$ (by $\textbf{let}^*$)
$\textbf{let}\ (x_r, x) = (q^\Delta d, \textbf{let}^*\ \Gamma = q^\Gamma(g)\ \textbf{in}\ t)\ \textbf{in}\ \textbf{let}^*\ \Delta = x_r\ \textbf{in}\ u$
$\equiv$ (by β equation)
$\textbf{let}\ x_r = q^\Delta\ d\ \textbf{in}\ \textbf{let}\ x = (\textbf{let}^*\ \Gamma = q^\Gamma(g)\ \textbf{in}\ t)\ \textbf{in}\ \textbf{let}^*\ \Delta = x_r\ \textbf{in}\ u$
$\equiv$ (by $\textbf{let}$ equation)
$\textbf{let}\ x = (\textbf{let}^*\ \Gamma = q^\Gamma(g)\ \textbf{in}\ t)\ \textbf{in}\ \textbf{let}^*\ \Delta = q^\Delta d\ \textbf{in}\ u$
$\equiv$ (by Lemma $4.2.3$)
$\textbf{let}^*\ \Gamma \otimes \Delta = q^{\Gamma\otimes\Delta} e\ \textbf{in}\ \textbf{let}\ x = t\ \textbf{in}\ u.$

6. $\Gamma \otimes \Delta \vdash (t,u) : \sigma \otimes \tau$. We want to show that: if $e \in [\![\Gamma \otimes \Delta]\!]$ then

$$q^{\sigma\otimes\tau}([\![\Gamma \otimes \Delta \vdash (t,u):\sigma\otimes\tau]\!]e) \equiv \textbf{let}^*\ \Gamma \otimes \Delta = q^{\Gamma\otimes\Delta} e\ \textbf{in}\ (t,u).$$

By induction hypothesis we have: if $g \in [\![\Gamma]\!]$, and $d \in [\![\Delta]\!]$, such that $\delta_{\Gamma,\Delta} e = (g,d)$, then

(a) $q^\sigma([\![\Gamma \vdash t:\sigma]\!]g) = \textbf{let}^*\ \Gamma = q^\Gamma(g)\ \textbf{in}\ t.$
(b) $q^\tau([\![\Delta \vdash u:\tau]\!]d) = \textbf{let}^*\ \Delta = q^\Delta(d)\ \textbf{in}\ u.$

Now developing the lhs:

$q^{\sigma\otimes\tau}([\![\Gamma \otimes \Delta \vdash (t,u):\sigma\otimes\tau]\!]e)$
$\equiv$ (by the meaning function in Figure 4.2)
$q^{\sigma\otimes\tau}(([\![\Gamma \vdash t:\sigma]\!] \times [\![\Delta \vdash u:\tau]\!]) \circ \delta_{\Gamma,\Delta} e)$
$\equiv$ (by simplification)
$q^{\sigma\otimes\tau}([\![\Gamma \vdash t:\sigma]\!]g, [\![\Delta \vdash u:\tau]\!]d)$
$\equiv$ (by q)
$(q^\sigma[\![\Gamma \vdash t:\sigma]\!]g, q^\tau[\![\Delta \vdash u:\tau]\!]d)$
$\equiv$ (by hypothesis a. and b.)
$(\textbf{let}^*\ \Gamma = q^\Gamma(g)\ \textbf{in}\ t, \textbf{let}^*\ \Delta = q^\Delta(d)\ \textbf{in}\ u)$
$\equiv$ (by Lemma $4.2.3$)
$\textbf{let}^*\ \Gamma \otimes \Delta = q^{\Gamma\otimes\Delta} e\ \textbf{in}\ (t,u).$

□

4.2.7.3 Inverting Evaluation

As explained earlier, the main ingredient of the proof of completeness is the function q^σ_Γ which inverts evaluation. To understand the basic idea of how the inverse of evaluation is defined, consider the following example. Let Γ be the environment $x : (\mathcal{Q}_2 \otimes \mathcal{Q}_2), y : \mathcal{Q}_2$ and let $f \in [\![\Gamma]\!] \to [\![\mathcal{Q}_2]\!]$. To find a syntactic term corresponding to f, we proceed as follows:

- flatten all the products by introducing intermediate names; this produces an updated environment $\Gamma' = x_1 : \mathcal{Q}_2, x_2 : \mathcal{Q}_2, y : \mathcal{Q}_2$, and an updated semantic function f' such that:
$$f' \ (((((), x_1), x_2), y) = f \ (((), (x_1, x_2)), y)$$
- enumerate all possible values for the variables, and apply f' to each enumeration to produce a result in the set $[\![\mathcal{Q}_2]\!]$. For example, it could be the case that $f \ (((), (1, 1)), 1) = 0$. The result of each enumeration can be inverted to a syntactic term using q^σ from Definition 4.2.3.
- Put things together using nested conditions representing all the possible values for the input variables. In the example we are considering, we get:

$$\begin{array}{l}
\textbf{let } (x_1, x_2) = x \\
\textbf{in if}^\circ \ x_1 \\
\quad \textbf{then if}^\circ \ x_2 \\
\qquad\quad \textbf{then if}^\circ \ y \ \textbf{then } \mathit{false} \\
\qquad\qquad\qquad\quad \textbf{else} \quad ... \\
\qquad\quad \textbf{else} \quad ... \\
\quad \textbf{else} \quad ...
\end{array}$$

The idea is formalised in the following definition.

Definition 4.2.4 *The function*

$$q^\sigma_\Gamma \in ([\![\Gamma]\!] \to [\![\sigma]\!]) \to \mathrm{Tm}\,\Gamma\,\sigma$$

for inverting evaluation *is defined by analysing the context:*

$$\begin{array}{rcl}
q^\sigma_\bullet(f) & = & q^\sigma \ (f \ (0)) \\
q^\sigma_{\Gamma, x:\mathcal{Q}_1}(f) & = & q^\sigma_\Gamma \ (h) \qquad \textit{where } h(g) = f(g, 0) \\
q^\sigma_{\Gamma, x:\mathcal{Q}_2}(f) & = & (\ \texttt{if}^\circ \ x \ \texttt{then} \ q^\sigma_\Gamma \ (h_1) \ \texttt{else} \ q^\sigma_\Gamma \ (h_0)) \\
 & & \textit{where } h_i(g) = f(g, i) \textit{ for } i \in \{0, 1\} \\
q^\sigma_{\Gamma, x:(\tau_1 \otimes \tau_2)}(f) & = & (\texttt{let} \ (x_1, x_2) = x \ \texttt{in} \ q^\sigma_{\Gamma, x_1:\tau_1, x_2:\tau_2}(h) \\
 & & \textit{where } h(g, x_1, x_2) = f(g, (x_1, x_2))
\end{array}$$

The base case is straightforward: the evaluation produces a closed value which can be inverted using the *"quote"* function of Definition 4.2.3. If the context includes a variable x of type $\mathcal{Q}_1$, then we supply the only possible value for that variable (0), and inductively construct the term with the variable x bound to $()$. The result is of the correct type because we can add or drop bindings of variables of type $\mathcal{Q}_1$ to the environment. If the context includes a variable x of type $\mathcal{Q}_2$, then we supply the two possible values for that variable

$$\frac{}{\bullet \vdash \overrightarrow{0} : \sigma}\ \text{z-intro} \qquad \frac{\Gamma \vdash t : \sigma}{\Gamma \vdash \kappa * t : \sigma}\ \text{prob} \qquad \frac{\Gamma \vdash t, u : \sigma}{\Gamma \vdash t + u : \sigma}\ \text{sup}$$

Figure 4.4: Typing quantum data (I)

0 and 1. A conditional is then used to select the correct branch depending on the actual value of x. Finally, if the context includes a variable of type $\tau_1 \otimes \tau_2$ then we simply flatten the product and proceed inductively. The function q^{σ}_{Γ} does indeed satisfy the inversion lemma.

4.3 Quantum Data and Control

The QML pure sublanguage terms consist of those presented in Section 4.2, extended with quantum data and quantum control. The full language also includes quantum measurement, which we do not consider in this work. The syntax of the quantum constructs is the following:

$$\begin{array}{lll} (\mathit{Prob.amplitudes}) & \kappa, \iota, ... \in & \mathbb{C} \\ (\mathit{Terms}) & t, u & ::= ... \mid \\ & & \overrightarrow{0} \mid \kappa * t \mid t + u \end{array}$$

Quantum data is modelled using the constructs $\kappa * t$, $\overrightarrow{0}$, and $t + u$. The term $\kappa * t$ where κ is a complex number associates the *probability amplitude* κ with the term t. It is convenient to have a special constant $\overrightarrow{0}$ for terms with probability amplitude zero. The term $t + u$ is a quantum *superposition* of t and u. Quantum superpositions are first-class values: when used as the first subexpression of a conditional, they turn the conditional into a *quantum control* construct. For example, $\mathbf{if}^{\circ}\ (\mathit{true} + \mathit{false})\ \mathbf{then}\ t\ \mathbf{else}\ u$ evaluates both t and u and combines their results in a quantum superposition.

We develop the typing rules and semantics of the quantum fragment of QML in two stages. First we extend the judgements $\Gamma \vdash t : \sigma$ and the semantics of Section 4.2.4 to handle quantum data in a straightforward manner. This simple treatment is only however an intermediate step in the development as it admits quantum programs that are not realisable on a quantum computer. We then refine both the type system and the semantics to identify exactly the realisable quantum programs.

4.3.1 The Category $\mathbf{Vec}$

As a first approximation to a type system for QML programs, we consider the type system of Figure 4.1 extended with the rules in Figure 4.4.

Unlike the classical case, a judgement $\Gamma \vdash t : \sigma$ is *not* interpreted as a function in $[\![\Gamma]\!] \to [\![\sigma]\!]$. Rather, because we now have superpositions of terms with complex probability amplitudes, we interpret such judgements as functions in $[\![\Gamma]\!] \to [\![\sigma]\!]^{\mathrm{Q}}$ where $[\![\sigma]\!]^{\mathrm{Q}}$ represents the complex vectors over the base set $[\![\sigma]\!]$. In other words, $[\![\sigma]\!]^{\mathrm{Q}}$ is defined to be $[\![\sigma]\!] \to \mathbb{C}$ which is sometimes denoted $[\![\vec{\sigma}]\!]$. We call the structure described above the category $\mathbf{Vec}$.

Naturally this change requires that we revisit the semantics of the classical terms given in Figure 4.2 so that each denotation returns a complex vector. For example, we should have:

$$[\![\bullet \vdash \mathrm{false} : \mathcal{Q}_2]\!]^{\mathrm{Q}} = \mathit{const}\ v \quad \text{where } v\ 0 = 1 \text{ and } v\ 1 = 0$$

Instead of mapping the value representing the empty context to the denotation of false, we

$$
\begin{array}{rcll}
[\![\bullet \vdash \overrightarrow{0} : \sigma]\!]^{\mathrm{Q}} & = & \mathit{const}\ v & \text{where } \forall a \in [\![\sigma]\!].v\ a = 0 \\
[\![\Gamma \vdash \kappa * t : \sigma]\!]^{\mathrm{Q}} & = & g & \text{where } g\ a = \kappa * (f a) \\
 & & & \qquad\quad f = [\![\Gamma \vdash t : \sigma]\!]^{\mathrm{Q}} \\
[\![\Gamma \vdash t + u : \sigma]\!]^{\mathrm{Q}} & = & h & \text{where } h\ a = f\ a + g\ a \\
 & & & \qquad\quad f = [\![\Gamma \vdash t : \sigma]\!]^{\mathrm{Q}} \\
 & & & \qquad\quad g = [\![\Gamma \vdash u : \sigma]\!]^{\mathrm{Q}}
\end{array}
$$

Figure 4.5: Meaning function for quantum data

now return a vector v which associates the denotation of false with probability amplitude 1 and the denotation of true with probability amplitude 0.

This change can be done systematically by noticing that it corresponds to a monad whose unit and lift operation are defined below:

$$
\begin{array}{rcl}
\mathit{return}\ a\ (b) & = & 1 \text{ if } a = b \text{ and } 0 \text{ otherwise} \\
f^*(v) & = & \Sigma a.(v\ a) * (f\ a)
\end{array}
$$

More precisely every value that is returned in Figure 4.2 is explicitly tagged with the monadic *return* and when two functions are composed in Figure 4.2 using $f \circ g$, the composition is replaced by $f^* \circ g$.

The meaning of the new constructs for quantum data is given in Figure 4.5.

4.3.2 Orthogonality

The type system presented so far does indeed correctly track the uses of variables and prevents variables from being weakened; yet the situation is more subtle. It turns out that the type system accepts terms which implicitly perform measurements and as a consequence accepts programs which are not realisable as quantum computations.

Consider the expression $\mathbf{if}^\circ\ x\ \mathbf{then}\ \mathit{true}\ \mathbf{else}\ \mathit{true}$: this expression appears, syntactically at least, to use x. However given the semantics of $\mathbf{if}^\circ$, which returns a superposition of the branches, the expression happens to return true without really *using* any information about x. In order to maintain the invariant that all measurements are explicit, the type system should reject the above expression as well.

More precisely, the expression $\mathbf{if}^\circ\ x\ \mathbf{then}\ t\ \mathbf{else}\ u$ should only be accepted if t and u are *orthogonal* quantum values ($t \perp u$). This notion intuitively ensures that the conditional operator does not implicitly discard any information about x during the evaluation. Because of a similar concern, the two branches of a superposition should also be orthogonal.

The typing rules for conditionals and superpositions are modified as in Figure 4.6. This modification also achieves that programs are normalised, *i.e.*, the sum of the probabilities of a superposition add up to 1.

In Figure 4.7 we define the inner product of terms, which to any pair of terms $\Gamma \vdash t, u : \sigma$ assigns $\langle t|u\rangle \in \mathbb{C} \cup \{?\}$. This is used to define orthogonality: $t \perp u$ holds if $\langle t|u\rangle = 0$.

The judgement $\vdash^\circ$ is not automatically closed under the equality judgement, hence we add the rule (subst). Our philosophy is that we allow equivalent representations of QML programs which do not satisfy the orthogonality criteria locally, as long as the program as a whole is equivalent to one which does satisfy the criteria.

$$\frac{\Gamma \vdash^{\circ} c : \mathcal{Q}_2 \qquad \Delta \vdash^{\circ} t, u : \sigma \qquad t \perp u}{\Gamma \otimes \Delta \vdash^{\circ} \texttt{if}^{\circ}\ c\ \texttt{then}\ t\ \texttt{else}\ u : \sigma}\ \texttt{if}^{\circ}$$

$$\frac{\Gamma \vdash^{\circ} t, u : \sigma \qquad t \perp u \qquad ||\lambda||^2 + ||\kappa||^2 = 1}{\Gamma \vdash^{\circ} \lambda * t + \kappa * u : \sigma}\ \sup^{\circ}$$

$$\frac{\Gamma \vdash^{\circ} t : \sigma \qquad \Gamma \vdash t \equiv u : \sigma}{\Gamma \vdash^{\circ} u : \sigma}\ \text{subst}$$

Figure 4.6: Typing quantum data (II)

$$\begin{array}{rcl@{\qquad}rcl}
\langle \mathrm{t}|\mathrm{t}\rangle & = & 1 & \langle \lambda * t + \lambda' * t' \mid u\rangle & = & \lambda^* * \langle t|u\rangle + \lambda'^* * \langle t'|u\rangle \\
\langle \mathrm{false}|\mathrm{true}\rangle & = & 0 & \langle t \mid \kappa * u + \kappa' * u'\rangle & = & \kappa * \langle t|u\rangle + \kappa' * \langle t|u'\rangle \\
\langle \mathrm{true}|\mathrm{false}\rangle & = & 0 & & & \\
 & & & \langle \lambda * t|u\rangle & = & \lambda^* \langle t|u\rangle \\
\langle \overrightarrow{0}|\mathrm{true}\rangle & = & 0 = \langle \mathrm{true}|\overrightarrow{0}\rangle & \langle t|\lambda * u\rangle & = & \lambda\langle t|u\rangle \\
\langle \overrightarrow{0}|\mathrm{false}\rangle & = & 0 = \langle \mathrm{false}|\overrightarrow{0}\rangle & \langle t + t'|u\rangle & = & \langle t|u\rangle + \langle t'|u\rangle \\
\langle \overrightarrow{0}|\mathrm{x}\rangle & = & 0 = \langle \mathrm{x}|\overrightarrow{0}\rangle & \langle t|u + u'\rangle & = & \langle t|u\rangle + \langle t|u'\rangle \\
 & & & & & \\
\langle (t,t') \mid (u,u')\rangle & = & \langle t|u\rangle * \langle t'|u'\rangle & \langle t|u\rangle & = & ? \qquad \text{otherwise}
\end{array}$$

Figure 4.7: Inner products and orthogonality

4.3.3 The Category $\mathbf{Q}^{\circ}$

The restriction of the set of typable terms requires a similar semantic restriction. All we need to do is to restrict the morphisms in the category of complex vectors to satisfy the following two conditions:

- Linearity: If $f \in \vec{A} \to \vec{B}$, $\alpha \in \mathbb{C}$, and $v, v_1, v_2 \in \vec{A}$, then $f(v_1 + v_2) = f(v_1) + f(v_2)$ and $f(\alpha v) = \alpha(f\ v)$.
- Isometry: If $f \in \vec{A} \to \vec{B}$ and $v_1, v_2 \in \vec{A}$, then $\langle v_1|v_2\rangle = \langle f\ v_1|f\ v_2\rangle$. (In other words, f preserves inner products of vectors.)

Two morphisms $f, g \in A \to B$ are *orthogonal* if for all vector $v \in \vec{A}$, we have $\langle f\ v|g\ v\rangle = 0$. We call the resulting category, the category $\mathbf{Q}^{\circ}$ of strict quantum computations. The homset of morphisms in $[\![\Gamma]\!] \to [\![\sigma]\!]^{\mathrm{Q}}$ satisfying the above conditions is called $\mathbf{Q}^{\circ}\ [\![\Gamma]\!]\ [\![\sigma]\!]^{\mathrm{Q}}$.

The meaning function is given as before but with the maps interpreted in the category $\mathbf{Q}^{\circ}$, *i.e.*, the meaning of a derivation $\Gamma \vdash t : \sigma$ is a morphism $[\![\Gamma \vdash t : \sigma]\!]^{\mathrm{Q}} \in \mathbf{Q}^{\circ}\ [\![\Gamma]\!]\ [\![\sigma]\!]^{\mathrm{Q}}$. The requirement for orthogonality in the type system is reflected semantically: for isometries f, g, we have that $f|g$ is an isometry, if f and g are orthogonal.

4.3.4 Quantum Equational Theory

The equational theory for the quantum language inherits all the equations for the classical case. This can be informally verified by noting that the meaning function in the case of the quantum language is essentially identical to the classical case. Formally, the proof technique explained in Section 4.2.7 applies equally well to the quantum case and yields

the same equations for the classical core plus additional equations to deal with quantum data.

Definition 4.3.1 *The* quantum equations *are:*

(if$^\circ$)

$$\begin{array}{lll} & \mathbf{if}^\circ\ (\lambda * t_0 + \kappa * t_1)\ \mathbf{then}\ u_0\ \mathbf{else}\ u_1 \\ \equiv & \lambda * (\mathbf{if}^\circ\ t_0\ \mathbf{then}\ u_0\ \mathbf{else}\ u_1) + \kappa * (\mathbf{if}^\circ\ t_1\ \mathbf{then}\ u_0\ \mathbf{else}\ u_1) \end{array}$$

(superpositions)

$$\begin{array}{lcl} t + u & \equiv & u + t \\ t + \overrightarrow{0} & \equiv & t \\ t + (u + v) & \equiv & (t + u) + v \\ \lambda * (t + u) & \equiv & \lambda * t + \lambda * u \\ \lambda * t + \kappa * t & \equiv & (\lambda + \kappa) * t \\ 0 * t & \equiv & \overrightarrow{0} \end{array}$$

Lemma 4.3.1 (Soundness) *The equational theory is sound: if* $\Gamma \vdash t \equiv u : \sigma$ *then the isometries* $[\![\Gamma \vdash t : \sigma]\!]^{\mathrm{Q}}$ *and* $[\![\Gamma \vdash u : \sigma]\!]^{\mathrm{Q}}$ *are extensionally equal.*

The additional equations are used to prove equality between different quantum values. Semantically, two quantum values are the same if they denote the same vector, which is the case if the sum of the paths to each classical value is the same. For example, to find a simplified quantum value equivalent to:

$$(\mathit{false} + \mathit{true}) + (\mathit{false} + (-1) * \mathit{true})$$

we first normalise to:

$$(1 \,/\, \sqrt{2}) * ((1 \,/\, \sqrt{2}) * \mathit{false} + (1 \,/\, \sqrt{2}) * \mathit{true}) +$$
$$(1 \,/\, \sqrt{2}) * ((1 \,/\, \sqrt{2}) * \mathit{false} + (-1 \,/\, \sqrt{2}) * \mathit{true})$$

This term has two paths to false; along each of them the product of the amplitudes is $(1 \,/\, \sqrt{2}) * (1 \,/\, \sqrt{2})$ which is $1 \,/\, 2$. The sum of all the paths to false is 1, and the sum of all the paths to true is 0. In other words, the entire term is equivalent to simply false. The above calculation proves that the Hadamard operation is self-inverse, as discussed in the introduction.

4.3.5 Quoting quantum values

We will now adapt the techniques developed in section 4.2.7 to the quantum case. A classical value $v \in \mathrm{Val}^{\mathrm{C}}\sigma$ is simply a term representing an element in $[\![\sigma]\!]$. A quantum value represents a vector in $[\![\sigma]\!]^{\mathrm{Q}}$, hence we have to close values under superpositions. We define $\mathrm{Val}^{\mathrm{Q}}\,\sigma \subseteq \mathrm{Tm}\ \sigma$ inductively as a subset of closed terms of type σ:

- $$\frac{v \in \mathrm{Val}^{\mathrm{C}}\,\sigma}{\mathrm{val}\,v \in \mathrm{Val}^{\mathrm{Q}}\,\sigma}$$
- $0 \in \mathrm{Val}^{\mathrm{Q}}\,\sigma$
- $$\frac{v, w \in \mathrm{Val}^{\mathrm{Q}}\,\sigma}{v + w \in \mathrm{Val}^{\mathrm{Q}}\,\sigma}$$

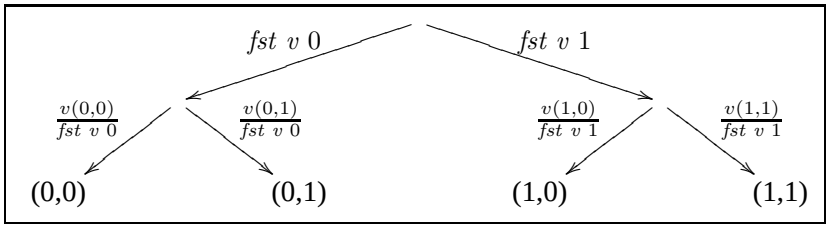


Figure 4.8: Value tree for $\mathcal{Q}_2 \otimes \mathcal{Q}_2$

- $$\frac{v \in \mathrm{Val}^{\mathrm{Q}}\,\sigma}{\kappa * v \in \mathrm{Val}^{\mathrm{Q}}\,\sigma}$$

We write $\mathrm{Val}^{\mathrm{Q}}_{\circ}\sigma$ for isometric quantum values which satisfy the restrictions introduced in Figure 4.6.

We have already seen that there is a monadic structure on $\vec{A} = A \to \mathbb{C}$. Correspondingly, we have a Kleisli structure on $\mathrm{Val}^{\mathrm{Q}}$; $\mathrm{val} \in \mathrm{Val}^{\mathrm{C}}\sigma \to \mathrm{Val}^{\mathrm{Q}}\sigma$ is the return and bind is defined as given $v \in \mathrm{Val}^{\mathrm{Q}}\sigma$ and $f \in \mathrm{Val}^{\mathrm{C}}\sigma \to \mathrm{Val}^{\mathrm{Q}}\tau$, we define $v \gg\!= f \in \mathrm{Val}^{\mathrm{Q}}\,\tau$ by induction over v:

$$\begin{array}{ll} (val\ x) & \gg\!= f = f\ x \\ 0 & \gg\!= f = 0 \\ v + w & \gg\!= f = (v \gg\!= f) + (w \gg\!= f) \\ \kappa * v & \gg\!= f = \kappa * (v \gg\!= f) \end{array}$$

Lemma 4.3.2 $(\mathrm{Val}^{\mathrm{C}}, \mathrm{Val}^{\mathrm{Q}}, val, (\gg\!=))$ *is a Kleisli structure, i.e. it satisfies the following equations:*

1. $val\ x \gg\!= f \equiv f\ x$
2. $v \gg\!= \lambda x.val\ x \equiv v$
3. $v \gg\!= \lambda x.(f\ x) \gg\!= g \equiv (v \gg\!= f) \gg\!= g$

Proof. Case (i) follows from the definition. Cases (ii) and (iii) can be shown by induction over the structure of v.

While the classical definition of q^{σ} (def. 4.2.3) was completely straightforward, its quantum counterpart is a bit more subtle, in particular in the case of tensor products. As a special case consider $q^{\mathcal{Q}_2 \otimes \mathcal{Q}_2}$, given an element

$$\overrightarrow{v} \in [\![\mathcal{Q}_2 \otimes \mathcal{Q}_2]\!]^{\mathrm{Q}} = [\![\mathcal{Q}_2]\!] \times [\![\mathcal{Q}_2]\!] \to \mathbb{C}$$

we have to construct a value $q^{\mathcal{Q}_2 \otimes \mathcal{Q}_2}\ \overrightarrow{v} \in \mathrm{Val}^{\mathrm{Q}}\ \mathcal{Q}_2 \otimes \mathcal{Q}_2$. This can be done by calculating the probabilities that the first qubit is i, $\mathrm{fst}\ \overrightarrow{v}\ i \in \mathbb{R}^+$, given by

$$\mathrm{fst}\ \overrightarrow{v}\ i = \sqrt{|\overrightarrow{v}(i,0)|^2 + |\overrightarrow{v}(i,1)|^2}$$

creating the first level of the value as a tree, and then for the second level normalising the amplitudes wrt. the probabilities of the previous level, see figure 4.8 for the corresponding tree. We write $[\![\sigma]\!]^{\mathrm{P}} = [\![\sigma]\!] \to \mathbb{R}^+$ for the set of probability distributions, obviously we have $[\![\sigma]\!]^{\mathrm{P}} \subseteq [\![\sigma]\!]^{\mathrm{Q}}$. We observe that $\mathrm{fst}\ \overrightarrow{v} \in [\![\sigma]\!]^{\mathrm{P}}$. Generalising the idea given above we arrive at the following definition of quote:

Definition 4.3.2 *The* syntactic representations of denotations *is given by*

$$q^{\sigma} \in [\![\sigma]\!]^{\mathrm{Q}} \to \mathrm{Val}^{\mathrm{Q}}\,\sigma$$

defined by induction over σ:

$$\begin{array}{rcl} q^{\mathcal{Q}_1}\,\overrightarrow{v} & = & (\overrightarrow{v}\,0) * () \\ q^{\mathcal{Q}_2}\,\overrightarrow{v} & = & (\overrightarrow{v}\,1) * \mathrm{true} + (\overrightarrow{v}\,0) * \mathrm{false} \\ q^{\sigma\otimes\tau}\,\overrightarrow{v} & = & q^{\sigma}(\mathrm{fst}\ \overrightarrow{v}) \\ & & \gg\!= \lambda x \in [\![\sigma]\!].(1/(\mathrm{fst}\ \overrightarrow{v})\,x) * q^{\tau}(\lambda y.\overrightarrow{v}(x,y)) \\ & & \gg\!= \lambda y.val\ (x,y) \end{array}$$

where:

$$\begin{array}{rcl} \mathrm{fst} & \in & [\![\sigma\otimes\tau]\!]^{\mathrm{Q}} \to [\![\sigma]\!]^{\mathrm{P}} \\ \mathrm{fst}\ \overrightarrow{v}\ x & = & \sqrt{\Sigma y.|\overrightarrow{v}(x,y)|^2} \end{array}$$

$$\begin{array}{rcl} 1/- & \in & [\![\sigma]\!]^{\mathrm{P}} \to [\![\sigma]\!]^{\mathrm{P}} \\ 1/\overrightarrow{v}\ x & = & \lambda x.\mathbf{if}\ p\ x \equiv 0\ \mathbf{then}\ 0\ \mathbf{else}\ 1\ /\ (p\ x) \end{array}$$

To show adequacy we have to establish a number of properties of q^{σ}: we have to show that it is linear and isometric and that it preserves tensor products. This is summarised in the following proposition:

Proposition 4.3.1

1. $q^{\sigma}\,(\kappa * \overrightarrow{v}) \equiv \kappa * (q^{\sigma}\ \overrightarrow{v})$
2. $q^{\sigma}\,(\overrightarrow{v} + \overrightarrow{w}) \equiv (q^{\sigma}\ \overrightarrow{v}) + (q^{\sigma}\ \overrightarrow{w})$
3. $\langle \overrightarrow{v}\,|\,\overrightarrow{w}\rangle = \langle q^{\sigma}\ \overrightarrow{v}\,|q^{\sigma}\ \overrightarrow{w}\rangle$
4. $q^{\sigma\otimes\tau}\,(\overrightarrow{v} \otimes \overrightarrow{w}) \equiv (q^{\sigma}\ \overrightarrow{v}, q^{\tau}\ \overrightarrow{w})$

The proof of the above proposition again isn't completely straightforward, e.g. linearity cannot just be proven by induction over σ. It is essential that we first establish some properties of renormalising a vector wrt. a probability distribution. We define the product of a probability distribution $p \in [\![\sigma]\!]^{\mathrm{P}}$ and a vector $\overrightarrow{v} \in [\![\sigma]\!]^{\mathrm{Q}}$ as:

$$\begin{array}{rcl} p * \overrightarrow{v} & \in & [\![\sigma]\!]^{\mathrm{Q}} \\ p * \overrightarrow{v} & = & \lambda x \in [\![\sigma]\!].(px) * (\overrightarrow{v}\,x) \end{array}$$

It is not hard to see that an analogous operation can be defined on values, given $v \in \mathrm{Val}^{\mathrm{Q}}\,\sigma$ and $p \in [\![\sigma]\!]^{\mathrm{P}}$ as above, we define:

$$\begin{array}{rcl} p * v & \in & \mathrm{Val}^{\mathrm{Q}}\,\sigma \\ p * v & = & v \gg\!= \lambda x \in [\![\sigma]\!].(px) * (\mathrm{val}\,x) \end{array}$$

The key property we establish is

Lemma 4.3.3 *Given $p \in [\![\sigma]\!]^{P}$ and $\overrightarrow{v} \in [\![\sigma]\!]^{\mathrm{Q}}$*

$$p * (q^{\sigma}\ \overrightarrow{v}) \equiv q^{\sigma}\ (p * \overrightarrow{v})$$

which can be verified by induction over σ and observing that while $1/-$ isn't a proper inverse, it nevertheless satisfies the following property

$$1/(p+q) * (p+q) = (1/p) * p$$

Using the fact that q^σ is isometric we can show that it produces values satisfying the orthogonality constraints:

Proposition 4.3.2 *Given* $v \in [\![\sigma]\!]^{\mathrm{Q}}$

$$\vdash^\circ q^\sigma\ v : \sigma$$

4.3.5.1 Adequacy

We define a syntactic counterpart to:

$$\delta_{\Gamma,\Delta} \in \mathbf{Q}^\circ\ [\![\Gamma \otimes \Delta]\!]\ ([\![\Gamma]\!]^{\mathrm{Q}} \otimes [\![\Delta]\!]^{\mathrm{Q}})$$

as:

$$\hat{\delta}_{\Gamma,\Delta} \in \mathrm{Tm}\ (\Gamma \otimes \Delta)\ (|\Gamma| \otimes |\Delta|)$$

by:

$$\hat{\delta}_{\Gamma,\Delta} = \begin{cases} \mathbf{let}\ (g,d) = \delta_{\Gamma',\Delta'} \mathbf{in}\ ((g,x),(d,x)) & \text{if } \Gamma = \Gamma', x:\sigma \\ \qquad \text{and } \Delta = \Delta', x:\sigma & \\ \mathbf{let}\ (g,d) = \delta_{\Gamma',\Delta} \mathbf{in}\ ((g,x),d) & \text{if } \Gamma = \Gamma', x:\sigma \\ \qquad \text{and } x \notin \text{dom } \Delta & \\ 1_\Delta & \text{if } \Gamma = \bullet \end{cases}$$

To establish that q^σ commutes with the context operations we have to show that contraction corresponds to $\delta \in \mathbf{Q}^\circ\ [\![\sigma]\!]\ ([\![\sigma]\!]^{\mathrm{Q}} \otimes [\![\sigma]\!]^{\mathrm{Q}})$.

Lemma 4.3.4 *Given* $v \in [\![\sigma]\!]^{\mathrm{Q}}$ *we have*

$$\mathbf{let}\ x = q^\sigma\ v\ \mathbf{in}\ (x,x) \equiv q^{\sigma\otimes\sigma}\ v$$

Proof. By induction on σ.

Exploiting this property we can show that the context operations commute with quote:

Lemma 4.3.5 *Given* $\overrightarrow{v} \in [\![\Gamma \otimes \Delta]\!]^{\mathrm{Q}}$

$$q^{|\Gamma|\otimes|\Delta|}\ (\delta_{\Gamma,\Delta}\ \overrightarrow{v}) \equiv \hat{\delta}_{\Gamma,\Delta}\ q^{|\Gamma\otimes\Delta|}\ \overrightarrow{v}$$

Theorem 4.3.3 *If* $\Gamma \vdash t : \sigma$ *and* $g \in [\![\Gamma]\!]^{\mathrm{Q}}$ *then*

$$\vdash q^\sigma([\![\Gamma \vdash t : \sigma]\!]^{\mathrm{Q}} g) \equiv \textbf{\textit{let}}^*\ \Gamma = q^\Gamma\ g\ \mathbf{in}\ t : \sigma.$$

Proof. By induction over the derivation of $\Gamma \vdash t : \sigma$, as an example consider the case for let:

$$\begin{array}{l} q^\rho\ ([\![\Gamma \otimes \Delta \vdash \mathtt{let}\ x = t\ \mathtt{in}\ u : \rho]\!]^{\mathrm{Q}}) \\ \equiv \{\text{definition of } [\![\ldots]\!]^{\mathrm{Q}}\} \\ q^\rho\ ([\![u]\!]^{\mathrm{Q}} \circ ([\![t]\!]^{\mathrm{Q}} \otimes id) \circ \delta_{\Gamma,\Delta}) \\ \equiv \{\text{induction hypothesis for } u \text{ and } t\} \\ u \circ (t \circ q^\Gamma \otimes q^\Delta) \circ \delta_{\Gamma,\Delta}) \\ \equiv \{\ \text{lemma 4.3.5}\ \}\} \\ u \circ (t \otimes id) \circ \hat{\delta}_{\Gamma,\Delta} \circ q^{|\Gamma\otimes\Delta|} \\ \equiv \\ (\mathtt{let}\ x = t\ \mathtt{in}\ u) \circ q^{|\Gamma\otimes\Delta|} \end{array}$$

The other cases use the same style of reasoning to deal with the structural properties and exploit proposition 4.3.1. Note that the case for $\mathbf{if}^\circ$ can be reduced to linearity.

Corollary 4.3.4 (Adequacy) *If* $\vdash t : \sigma$ *then* $\vdash q^\sigma([\![\vdash t : \sigma]\!]^{\mathrm{Q}}) \equiv t : \sigma$

4.3.5.2 Completeness and normalisation

The development here follows closely the one in the classical case as presented in Section 4.2.7.3.

Definition 4.3.3 *The function:*

$$q^\sigma_\Gamma \in \mathbf{Q}^\circ\,[\![\Gamma]\!]\,[\![\sigma]\!]^{\mathrm{Q}} \to \mathrm{Tm}\,\Gamma\,\sigma$$

for inverting evaluation *is defined by analysing the context:*

$$\begin{array}{rcl}
q^\sigma_\bullet(f) & = & q^\sigma\ (f\ (\textit{return}\ 0)) \\
q^\sigma_{\Gamma,x:\mathcal{Q}_1}(f) & = & \phi^{-1}_{\Gamma,x:\mathcal{Q}_1} \circ (q^\rho_\Gamma) \circ \Phi_{\Gamma,x:\mathcal{Q}_1} \\
q^\sigma_{\Gamma,x:\mathcal{Q}_2}(f) & = & \phi^{-1}_{\Gamma,x:\mathcal{Q}_2} \circ (q^\sigma_\Gamma \times q^\sigma_\Gamma) \circ \Phi_{\Gamma,x:\mathcal{Q}_2} \\
q^\sigma_{\Gamma,x:(\tau_1\otimes\tau_2)}(f) & = & \phi^{-1}_{\Gamma,x:\tau_1\otimes\tau_2} \circ q^\sigma_{\Gamma,x_1:\tau_1,x_2:\tau_2} \circ \Phi_{\Gamma,x:\tau_1\otimes\tau_2}
\end{array}$$

The auxiliary isomorphisms are defined as follows:

$$\begin{array}{rcl}
\phi_{\Gamma,x:\mathcal{Q}_1} & \in & \mathrm{Tm}\,(\Gamma, x:\mathcal{Q}_1)\,\sigma \to \mathrm{Tm}\,\Gamma\,\sigma \\
\phi_{\Gamma,x:\mathcal{Q}_1} t & = & \mathtt{let}\ x = ()\ \mathtt{in}\ t \\
\phi_\Gamma t & = & t
\end{array}$$

$$\begin{array}{rcl}
\phi_{\Gamma,x:\mathcal{Q}_2} & \in & \mathrm{Tm}\,(\Gamma, x:\mathcal{Q}_2\,\sigma) \to \{(t_0,t_1) \in (\mathrm{Tm}\,\Gamma\,\sigma)^2 \mid t_0 \perp t_1\} \\
\phi_{x:\mathcal{Q}_2}\, t & = & (\mathtt{let}\ x = \mathrm{false}\ \mathtt{in}\ t, \mathtt{let}\ x = \mathrm{true}\ \mathtt{in}\ t) \\
\phi^{-1}_{\Gamma,x:\mathcal{Q}_2}(t,u) & = & \mathtt{if}^\circ\ x\ \mathtt{then}\ t\ \mathtt{else}\ u
\end{array}$$

$$\begin{array}{rcl}
\phi_{\Gamma,x:\tau_1\otimes\tau_2} & \in & \mathrm{Tm}\,(\Gamma, x:\tau_1\otimes\tau_2)\,\rho \to \mathrm{Tm}\,(\Gamma, x_1:\tau_1, x_2:\tau_2) \\
\phi_{\Gamma,x:\tau_1\otimes\tau_2}\, t & = & \mathtt{let}\ x = (x_1,x_2)\ \mathtt{in}\ t \\
\phi^{-1}_{\Gamma,x:\tau_1\otimes\tau_2}(t) & = & \mathtt{let}\ (x_1,x_2) = x\ \mathtt{in}\ t
\end{array}$$

The semantic map corresponding to each ϕ *is written* Φ.

For the inversion proof we only need the provability of one side of the isomorphisms which follows from the η-equalities.

Lemma 4.3.6 *The following family of equalities is derivable*

$$\phi^{-1}_\Gamma(\phi_\Gamma t) \equiv t$$

Definition 4.3.4 *The* normal form *of* t *is given by* $nf^\sigma_\Gamma(t) = q^\sigma_\Gamma([\![\Gamma \vdash t : \sigma]\!]^{\mathrm{Q}})$.

Lemma 4.3.7 (Inversion) *The equation* $\Gamma \vdash nf^\sigma_\Gamma(t) \equiv t$ *is derivable.*

Proof. By induction over the definition of q^σ_Γ. In the case of $\Gamma = \bullet$ the result follows from adequacy, Corollary 4.3.4. In all the other cases we exploit Lemma 4.3.6.

Since all our definitions are effective nf indeed gives rise to a normalisation algorithm. As a consequence, our equational theory is decidable, modulo deciding equalities of the complex number terms which occur in our programs. We also note that as in the classical case, our theory is complete:

Proposition 4.3.5 (Completeness) *If* $[\![\Gamma \vdash t : \sigma]\!]^{\mathrm{Q}}$ *and* $[\![\Gamma \vdash u : \sigma]\!]^{\mathrm{Q}}$ *are extensionally equal, then we can derive* $\Gamma \vdash t \equiv u : \sigma$.

4.4 Summary

In this chapter we presented a sound and complete equational theory for a subset of QML excluding measurements. This enables syntactic reasoning about quantum programs using standard classical tools from semantics of classical programming languages. A next step would be generalise this approach to the full QML including measurements. In next chapters we structure a model for general and complete quantum computations using arrows. We hope to integrate the results of next chapters with a quantum programming language like QML.

5 MODELLING QUANTUM EFFECTS I: STATE VECTORS AS INDEXED MONADS

The traditional model of quantum computing is based on vector spaces, with *normalized vectors* to model computational states and *unitary transformations* to model physically realizable quantum computations. The idea is that information processing is physically realized via a *closed quantum system*.

In a closed quantum system, the evolution is *reversible* (also called *strict* or *pure*), that is, it is only given by means of unitary gates; measurements, which model the *interaction* with external world, are not considered. Therefore, in this context, the quantum computational process is considered like a black box, where information can be input and then read at the end of the process.

As explained in Chapter 2 there are some intrinsic differences between classical and quantum programming due to the nature of quantum states and operations acting on these states. Using the traditional model we can emphasize two main characteristics of quantum programming:

- quantum parallelism, which is caused by the quantum superposition phenomenon and expressed by *vector* states.
- *global* (possible entangled) quantum state, which is why not all composed vectors, that model a quantum state, can be decomposed into their subparts. In this way, each operation is global, yet in quantum circuits this global action is hidden. Abstractly, the application of a specific operation to a specific *subspace* of the vector space is achieved by the application of an operation to the whole space which carries the identity to the remaining subspaces. The semantics of any quantum programming language needs to take care of that.

In this chapter we present a monadic approach for quantum programming in Haskell. We show how to structure quantum state vectors using monads in Haskell, in such a way that the application of unitary transformations to state vectors is modelled by the monadic *bind* operation.

5.1 Vectors

Given a set a representing observable (classical) values, i.e. a *basis* set, a pure quantum state is a vector $a \to \mathbb{C}$ which associates each basis element with a complex probability amplitude.

In Haskell, a finite set a can be represented as an instance of the class $Basis$, shown below, which has the constructor $basis \in [a]$ explicitly listing the basis elements. The

basis elements must be distinguishable from each other, which explains the constraint $Eq\ a$ on the type of elements below:

$$\textbf{class}\ Eq\ a \Rightarrow Basis\ a\ \textbf{where}\ basis \in [a]$$
$$\textbf{type}\ K = \mathbb{C}\ Double$$
$$\textbf{type}\ Vec\ a = a \to K$$

The type K (notation from the base field) is the type of probability amplitudes.

As we saw in Chapter 3, from a programming perspective, a monad (or Kleisli triple) is a way to structure *computations* in terms of values and sequences of computations using those values.

Because we can only build vectors over a *set*, which is a basis (that is, a set of observable values), our computations have the additional constraint that they are indexed by $Basis$. Therefore, the type constructor Vec corresponds to a *Kleisli structure* (ALTENKIRCH; REUS, 1999) or, to an *indexed monad* (see Section 3.1.4).

Recall the class for indexed monads:

$$\textbf{class}\ IMonad\ m\ \textbf{where}$$
$$return \in F\ a \Rightarrow a \to m\ a$$
$$(\gg\!=) \in (F\ a, F\ b) \Rightarrow m\ a \to (a \to m\ b) \to m\ b$$

Therefore to make our vectors an instance of $CMonad$ class we need to define:

$$\textbf{instance}\ IMonad\ Vec\ \textbf{where}$$
$$return \in Basis\ a \Rightarrow a \to Vec\ a$$
$$return\ a\ b = \textbf{if}\ a \equiv b\ \textbf{then}\ 1.0\ \textbf{else}\ 0.0$$
$$(\gg\!=) \in (Basis\ a, Basis\ b) \Rightarrow Vec\ a \to (a \to Vec\ b) \to Vec\ b$$
$$va \gg\!= f = \lambda b \to sum\ [(va\ a) * (f\ a\ b) \mid a \leftarrow basis]$$

$return$ just lifts values to vectors, and $bind$, given a linear operator represented as a function $a \to Vec\ b$, and given a $Vec\ a$, returns a $Vec\ b$. Using the functional representation for vectors and the definition above for $bind$ we can easily extend any linear operator to act in a bigger space as we explain in detail in next section.

Proposition 5.1.1 *The indexed monad Vec satisfies the required equations for monads (see Section 3.1.1).*

Proof.

- First monad law: $(return\ x) \gg\!= f = f\ x$

$$\begin{aligned}(return\ x) \gg\!= f &= \lambda b \to sum\ [(return\ x\ a) * (f\ a\ b) \mid a \leftarrow basis] \\ &= \lambda b \to sum\ [(\textbf{if}\ x \equiv a\ \textbf{then}\ 1.0\ \textbf{else}\ 0.0) * (f\ a\ b) \mid \\ &\qquad a \leftarrow basis] \\ &= \lambda b \to f\ x\ b \\ &= f\ x\end{aligned}$$

- Second monad law: $m \gg\!= return = m$

$$\begin{aligned}m \gg\!= return &= \lambda b \to s\ um\ [(m\ a) * (return\ a\ b) \mid a \leftarrow basis] \\ &= \lambda b \to sum\ [(m\ a) * (\textbf{if}\ a \equiv b\ \textbf{then}\ 1.0\ \textbf{else}\ 0.0) \mid \\ &\qquad a \leftarrow basis] \\ &= \lambda b \to m\ b \\ &= m\end{aligned}$$

- Third monad law: $(m \gg\!= f) \gg\!= g = m \gg\!= (\lambda\, x\, .\, f\ x \gg\!= g)$

$$
\begin{aligned}
(m \gg\!= f) \gg\!= g &= (\lambda b \to sum\,[(m\ a) * (f\ a\ b) \mid a \leftarrow basis]) \gg\!= g \\
&= \lambda c \to sum\,[(sum\,[(m\ a) * (f\ a\ b) \mid a \leftarrow basis]) \\
&\qquad\qquad * (g\ b\ c) \mid b \leftarrow basis] \\
&= \lambda c \to sum\,[(m\ a) * (f\ a\ b) * (g\ b\ c) \\
&\qquad\qquad a \leftarrow basis, b \leftarrow basis]
\end{aligned}
$$

$$
\begin{aligned}
m \gg\!= (\lambda x \to f\ x \gg\!= g) &= \lambda c \to sum\,[(m\ a) * ((f\ a \gg\!= g)\ c) \mid \\
&\qquad\qquad a \leftarrow basis] \\
&= \lambda c \to sum\,[(m\ a) * (sum\,[(f\ a\ b) * (g\ b\ c) \mid \\
&\qquad\qquad b \leftarrow basis]) \mid a \leftarrow basis] \\
&= \lambda c \to sum\,[(m\ a) * (f\ a\ b) * (g\ b\ c) \mid \\
&\qquad\qquad a \leftarrow basis, b \leftarrow basis]
\end{aligned}
$$

□

The indexed monads have additional properties abstracted in the indexed version of the Haskell class $MonadPlus$:

class $IMonad\ m \Rightarrow IMonadPlus\ m$ **where**
$mzero \in F\ a \Rightarrow m\ a$
$mplus \in F\ a \Rightarrow m\ a \to m\ a \to m\ a$

Instances of this class support two additional methods: $mzero$ and $mplus$ which provide a "zero" computation and an operation to "add" computations:

instance $IMonadPlus\ Vec$ **where**
$mzero \in Basis\ a \Rightarrow Vec\ a$
$mzero = const\ 0.0$
$mplus \in Basis\ a \Rightarrow Vec\ a \to Vec\ a \to Vec\ a$
$mplus\ v_1\ v_2\ a = v_1\ a + v_2\ a$

where $const \in t \to t_1 \to t$ is a Haskell's polymorphic function. Analogously, we can also define $mminus$:

$mminus \in Basis\ a \Rightarrow Vec\ a \to Vec\ a \to Vec\ a$
$mminus\ v_1\ v_2\ a = v_1\ a - v_2\ a$

As we are modelling vector spaces in Haskell, we would like to define *products* over vectors: the *scalar* product $\$*$, the *tensor* product $\langle * \rangle$, and the *dot* product $\langle \cdot \rangle$:

$(\$*) \in K \to Vec\ a \to Vec\ a$
$pa\ \$*\ v = \lambda a \to pa * (v\ a)$
$(\langle * \rangle) \in Vec\ a \to Vec\ b \to Vec\ (a, b)$
$v_1 \langle * \rangle v_2 = \lambda(a, b) \to (v_1\ a) * (v_2\ b)$
$(\langle . \rangle) \in Basis\ a \Rightarrow Vec\ a \to Vec\ a \to K$
$v_1 \langle . \rangle v_2 = sum\,[conjugate\ (v_1\ a) * (v_2\ a) \mid a \leftarrow basis]$

Examples of vectors over the set of booleans may be defined as follows:

instance $Basis\ Bool$ **where**
$basis = [False, True]$
$qFalse, qTrue, qFT, qFmT \in Vec\ Bool$
$qFalse = return\ False$
$qTrue = return\ True$
$qFT = (1\ /\ \sqrt{2})\ \$*\ (qFalse\ `mplus`\ qTrue)$
$qFmT = (1\ /\ \sqrt{2})\ \$*\ (qFalse\ `mminus`\ qTrue)$

The first two are unit vectors corresponding to basis elements; the last two represent states which are in equal superpositions of $False$ and $True$. In the Dirac notation, these

vectors would be respectively written as $|\mathit{False}\rangle$, $|\mathit{True}\rangle$, $\frac{1}{\sqrt{2}}(|\mathit{False}\rangle + |\mathit{True}\rangle)$, and $\frac{1}{\sqrt{2}}(|\mathit{False}\rangle - |\mathit{True}\rangle)$.

Multidimensional vectors can be easily described using the tensor product on vectors or the Cartesian product on the underlying bases:

$$\begin{aligned}
&\textbf{instance}\ (\mathit{Basis}\ a, \mathit{Basis}\ b) \Rightarrow \mathit{Basis}\ (a, b)\ \textbf{where}\\
&\quad \mathit{basis} = [(a, b) \mid a \leftarrow \mathit{basis}, b \leftarrow \mathit{basis}]\\
&p_1, p_2, p_3 \in \mathit{Vec}\ (\mathit{Bool}, \mathit{Bool})\\
&p_1 = \mathit{qFT}\langle * \rangle \mathit{qFalse}\\
&p_2 = \mathit{qFalse}\langle * \rangle \mathit{qFT}\\
&p_3 = \mathit{qFT}\langle * \rangle \mathit{qFT}\\
&\mathit{epr} \in \mathit{Vec}\ (\mathit{Bool}, \mathit{Bool})\\
&\mathit{epr}\ (\mathit{False}, \mathit{False}) = 1 \,/\, \sqrt{2}\\
&\mathit{epr}\ (\mathit{True}, \mathit{True}) = 1 \,/\, \sqrt{2}\\
&\mathit{epr}\ _ = 0
\end{aligned}$$

In contrast to the first three vectors, the last vector describes an *entangled* quantum state which cannot be separated into the product of independent quantum states.

5.2 Linear Operators

Given two base sets A and B a linear operator $f \in A \multimap B$ is a function mapping vectors over A to vectors over B. We represent such operators as functions mapping values to vectors which is similar to the representation used by Karczmarczuk (2003) and which mirrors a *matrix*:

$$\begin{aligned}
&\textbf{type}\ \mathit{Lin}\ a\ b = a \to \mathit{Vec}\ b\\
&\mathit{fun2lin} \in (\mathit{Basis}\ a, \mathit{Basis}\ b) \Rightarrow (a \to b) \to \mathit{Lin}\ a\ b\\
&\mathit{fun2lin}\ f\ a = \mathit{return}\ (f\ a)
\end{aligned}$$

The function $\mathit{fun2lin}$ converts a classical (reversible) function to a linear operator. For example, the quantum version of the boolean negation is:

$$\begin{aligned}
&\mathit{qnot} \in \mathit{Lin}\ \mathit{Bool}\ \mathit{Bool}\\
&\mathit{qnot} = \mathit{fun2lin}\ \mathit{not}
\end{aligned}$$

Linear operations can also be defined directly, for example:

$$\begin{aligned}
&\mathit{phase} \in \mathit{Lin}\ \mathit{Bool}\ \mathit{Bool}\\
&\mathit{phase}\ \mathit{False} = \mathit{return}\ \mathit{False}\\
&\mathit{phase}\ \mathit{True} = (0 :+ 1)\ \$*\ (\mathit{return}\ \mathit{True})\\
&\mathit{hadamard} \in \mathit{Lin}\ \mathit{Bool}\ \mathit{Bool}\\
&\mathit{hadamard}\ \mathit{False} = \mathit{qFT}\\
&\mathit{hadamard}\ \mathit{True} = \mathit{qFmT}\\
&\mathit{zgate} \in \mathit{Lin}\ \mathit{Bool}\ \mathit{Bool}\\
&\mathit{zgate}\ \mathit{False} = \mathit{qFalse}\\
&\mathit{zgate}\ \mathit{True} = -1\ \$*\ \mathit{qTrue}
\end{aligned}$$

The definition of a linear operation specifies its action on each individual element of the basis, as a matrix. To apply a linear operation f to a vector v, we use the *bind* operation to calculate $v \gg\!= f$. For example $(\mathit{qFT} \gg\!= \mathit{hadamard})$ applies the operation $\mathit{hadamard}$ to the vector qFT, which can be calculated as follows:

$$\begin{aligned}
&\mathit{qFT} \gg\!= \mathit{hadamard}\\
&= \lambda b \to \mathit{sum}\ [(\mathit{qFT}\ a)\ (\mathit{hadamard}\ a\ b) \mid a \leftarrow [\mathit{False}, \mathit{True}]]\\
&= \lambda b \to \textbf{if}\ b \equiv \mathit{False}\ \textbf{then}\ \mathit{sum}\ [(\mathit{qFT}\ \mathit{False})\ (\mathit{hadamard}\ \mathit{False}\ \mathit{False}) +
\end{aligned}$$

$$
\begin{aligned}
&\qquad\qquad\qquad\qquad (qFT\ True)\ (hadamard\ True\ False)] \\
&\quad \textbf{else}\ sum\ [(qFT\ False)\ (hadamard\ False\ True) + \\
&\qquad\qquad (qFT\ True)\ (hadamard\ True\ True)] \\
&= \lambda b \rightarrow \textbf{if}\ b \equiv False\ \textbf{then}\ 1 \\
&\quad \textbf{else}\ 0
\end{aligned}
$$

that is, it produces the vector $qFalse$ as a result.

It is possible to write higher-order functions which consume linear operators and produce new linear operators. A very importante example of such functions extends the space of action of a linear operator:

$$extend \in (Basis\ a, Basis\ b) \Rightarrow Lin\ a\ a \rightarrow Lin\ (a, b)\ (a, b)$$
$$extend\ f = \lambda(a_1, b_1) \rightarrow (f\ a_1 \gg\!= \lambda a_2 \rightarrow return\ (a_2, b_1))$$

The definition of $bind$ gives us this possibility of easily extending a linear operator to act in a bigger space. A function similar to that is used in the **do**-notation for the implementation of the circuit for the Toffoli gate in next section.

Another example produces the so-called *controlled operations*:

$$controlled \in Basis\ a \Rightarrow Lin\ a\ a \rightarrow Lin\ (Bool, a)\ (Bool, a)$$
$$controlled\ f\ (b, a) = (return\ b)\langle * \rangle(\textbf{if}\ b\ \textbf{then}\ f\ a\ \textbf{else}\ return\ a)$$

The linear operator f is transformed to a new linear operator controlled by a quantum boolean value. The modified operator returns a pair whose first component is the input control value. The second input is passed to f only if the control value is true, and is otherwise left unchanged. For example, $(qFT\ \langle * \rangle\ qFalse) \gg\!= (controlled\ qnot)$ applies the familiar *controlled-not* gate to a vector over two values: the control value is a superposition of $False$ and $True$ and the data value is $False$. As one may calculate, the result of this application is the epr vector.

Linear operations can be combined and transformed in several ways which we list below. The function $\rangle * \langle$ produces the linear operator corresponding to the *outer product* of two vectors. The functions $linplus$ and $lintens$ are the functions corresponding to the sum and tensor product on vectors. Finally the function o composes two linear operators.

$$adjoint \in (Basis\ a, Basis\ b) \Rightarrow Lin\ a\ b \rightarrow Lin\ b\ a$$
$$adjoint\ f\ b\ a = conjugate\ (f\ a\ b)$$
$$(\rangle * \langle) \in (Basis\ a, Basis\ b) \Rightarrow Vec\ a \rightarrow Vec\ b \rightarrow Lin\ a\ b$$
$$(v_1\rangle * \langle v_2)\ a\ b = (v_1\ a) * (conjugate\ (v_2\ b))$$
$$linplus \in (Basis\ a, Basis\ b) \Rightarrow Lin\ a\ b \rightarrow Lin\ a\ b \rightarrow Lin\ a\ b$$
$$linplus\ f\ g\ a = f\ a\ `mplus`\ g\ a$$
$$lintens \in (Basis\ a, Basis\ b, Basis\ c, Basis\ d) \Rightarrow$$
$$\quad Lin\ a\ b \rightarrow Lin\ c\ d \rightarrow Lin\ (a, c)\ (b, d)$$
$$lintens\ f\ g\ (a, c) = f\ a\langle * \rangle g\ c$$
$$o \in (Basis\ a, Basis\ b, Basis\ c) \Rightarrow Lin\ a\ b \rightarrow Lin\ b\ c \rightarrow Lin\ a\ c$$
$$o\ f\ g\ a = (f\ a \gg\!= g)$$

5.3 Example: A Circuit for the Toffoli Gate

Modelling state vectors as monads we can define quite elegant quantum programs using monads' **do**-notation. For instance, consider we want to program a circuit for the Toffoli gate, as in Figure 5.1. The circuit diagram uses the de-facto standard notation for specifying quantum computations. Each line carries one quantum bit (*qubit*); we refer to the three qubits in the circuit as top, $middle$, and $bottom$. The values flow from left to

right in steps corresponding to the alignment of the boxes, which represent quantum gates. The gates labeled H, V, VT, and Not represent the quantum operations $hadamard$, $phase$, $adjoint\ phase$, and $qnot$ respectively. Gates connected via a bullet to another wire are *controlled* operations.

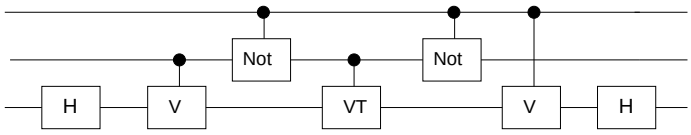


Figure 5.1: A Circuit for the Toffoli Gate.

In general all three qubits in the circuit may be entangled and hence the state vector representing them cannot be separated into individual state vectors. This means that, despite the appearance to the contrary, it is not possible to operate on any of the lines individually. Instead the circuit defines a linear operation on the entire state. However, as one can observe below, using **do**-notation we can elegantly program the circuit, *hiding* rewiring and the global state action of each quantum operator:

$$
\begin{aligned}
&toffoli \in Lin\ (Bool, Bool, Bool)\ (Bool, Bool, Bool)\\
&toffoli\ (top, middle, bottom) =\\
&\quad \mathbf{do}\ b_1 \leftarrow hadamard\ bottom\\
&\qquad (m_1, b_2) \leftarrow controlled\ phase\ (middle, b_1)\\
&\qquad (t_1, m_2) \leftarrow controlled\ qnot\ (top, m_1)\\
&\qquad (m_3, b_3) \leftarrow controlled\ (adjoint\ phase)\ (m_2, b_2)\\
&\qquad (t_2, m4) \leftarrow controlled\ qnot\ (t_1, m_3)\\
&\qquad (t_3, b_4) \leftarrow controlled\ phase\ (t_2, b_3)\\
&\qquad b_5 \leftarrow hadamard\ b_4\\
&\qquad return\ (t_3, m4, b_5)
\end{aligned}
$$

Note that we are using indices in the value variables. This is because monads' **do**-notation simulates an imperative routine. The evolution of the indices in the circuit can be analysed in the figure below:

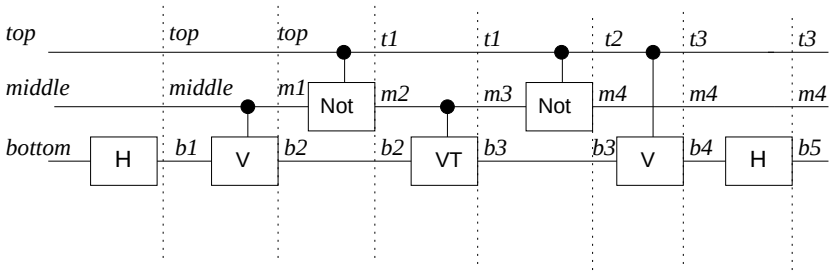


Figure 5.2: The evolution of values in the circuit for the Toffoli gate.

Running $toffoli$, which behaves like a controlled-controlled not as explained in Section 2.3.2, applied to the following entangled state [1]:

$$
\begin{aligned}
&emt \in Vec\ (Bool, Bool, Bool)\\
&emt = (1\ /\ \sqrt{2}\ \$*\ return\ (True, True, True))\text{`}mplus\text{'}\\
&\qquad (1\ /\ \sqrt{2}\ \$*\ return\ (False, False, False))
\end{aligned}
$$

[1] In Dirac's notation: $1/\sqrt{2}(|000\rangle + |111\rangle)$.

produces [2]:

$$haskell\rangle\, emt \gg\!= toffoli$$
$$[((False, False, False), 0.5 :+ 0.0),$$
$$((True, True, False), 0.5 :+ 0.0)]$$

5.4 Summary

We have shown the use of monads to structure the probability effects of quantum state vectors. The approach reveals an elegant underlying structure for quantum computations. This structure can be studied in the context of category theory and exploited in the design of calculi for quantum computation (TONDER, 2003, 2004; VALIRON, 2004; ALTENKIRCH; GRATTAGE, 2005).

Unfortunately in the monadic model of quantum computing we have used so far, it is difficult or impossible to deal formally with another class of quantum system, which present effects including measurements, decoherence, or noise, say to be *open quantum systems*. In the next chapter we consider density matrices and superoperators as a model for general quantum computations.

[2] In Dirac's notation: $1/\sqrt{2}(|000\rangle + |110\rangle)$.

6 MODELLING QUANTUM EFFECTS II: SUPEROPERATORS AS INDEXED ARROWS

While the state vector model of quantum computing is still widely considered as a convenient formalism to describe quantum algorithms, using measurements to deal with decoherence or noise, to make quantum computing an *interactive* process, and even to steer quantum computations has been considered a novel alternative, for instance see (AHARONOV; KITAEV; NISAN, 1998; RAUSSENDORF; BROWNE; BRIEGEL, 2001, 2003; KASHEFI; PANANGADEN; DANOS, 2004; DANOS et al., 2005; GAY; NAGARAJAN, 2006).

In this chapter we review the general model of quantum computations, including measurements, based on density matrices and superoperators. After expressing this more general model in Haskell, we establish that the superoperators used to express all quantum computations and measurements are an instance of the concept of *indexed arrows*, a generalisation of monads (see Chapter 3). The material presented on this chapter has been published in (VIZZOTTO; ALTENKIRCH; SABRY, 2006).

6.1 Density Matrices and Superoperators

We review, using Haskell, a generalised model of quantum computation where the state of computation is represented using a *density matrix* and the operations are represented using *superoperators* (AHARONOV; KITAEV; NISAN, 1998). Using these notions, the *projections* necessary to express measurements become expressible within the model.

6.1.1 Density Matrices

Intuitively, density matrices can be understood as a statistical perspective of the state vector. In the density matrix formalism, a quantum state that used to be modelled by a vector v (as presented in Section 5.1) is now modelled by its outer product in such a way that the *amplitudes of the state vector turn into a kind of probability distributions of state vectors* [1].

$$\begin{aligned}
&\mathbf{type}\ Dens\ a = Vec\ (a, a)\\
&pureD \in Basis\ a \Rightarrow Vec\ a \rightarrow Dens\ a\\
&pureD\ v = lin2vec\ (v\rangle{*}\langle v)\\
&lin2vec \in (a \rightarrow Vec\ b) \rightarrow Vec\ (a, b)\\
&lin2vec = uncurry
\end{aligned}$$

[1] The construction in this chapter is build on the construction in Chapter 5.

The function $pureD$ embeds a state vector in its density matrix representation. For convenience, we uncurry the arguments to the density matrix so that it looks more like a "matrix." For example, the density matrices corresponding to the vectors $qFalse$, $qTrue$, and qFT presented in Section 5.1 can be visually represented as follows:

$$\begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} \qquad \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} \qquad \begin{pmatrix} 1/2 & 1/2 \\ 1/2 & 1/2 \end{pmatrix}$$

and written as:

$qFalseD, qTrueD, qFTD \in Dens\ Bool$
$qFalseD = pureD\ qFalse$
$qTrueD = pureD\ qTrue$
$qFTD = pureD\ qFT$

In Haskell, we use the a following pretty printing for those matrices:

$[((False, False), 1.0 :+ 0.0)]$
$[((True, True), 1.0 :+ 0.0)\ ]$
$[((False, False), 0.5 :+ 0.0),$
$((False, True), 0.5 :+ 0.0),$
$((True, False), 0.5 :+ 0.0),$
$((True, True), 0.5 :+ 0.0)\ ]$

The appeal of density matrices is that they can represent states other than the pure ones above. In particular if we perform a measurement on the state represented by qFT, we should get $False$ with probability $1/2$ or $True$ with probability $1/2$. This information, which cannot be expressed using vectors, can be represented by the following density matrix:

$$\begin{pmatrix} 1/2 & 0 \\ 0 & 0 \end{pmatrix} + \begin{pmatrix} 0 & 0 \\ 0 & 1/2 \end{pmatrix} = \begin{pmatrix} 1/2 & 0 \\ 0 & 1/2 \end{pmatrix}$$

Such a density matrix represents a *mixed state* which corresponds to the sum (and then normalisation) of the density matrices for the two results of the observation.

6.1.2 Superoperators

Operations mapping density matrices to density matrices are called *superoperators*:

$\mathbf{type}\ Super\ a\ b = (a, a) \to Dens\ b$
$lin2super \in (Basis\ a, Basis\ b) \Rightarrow Lin\ a\ b \to Super\ a\ b$
$lin2super\ f\ (a_1, a_2) = lin2vec\ (f\ a_1\rangle * \langle f\ a_2)$

As we have done for unitary operators in Section 5.2, we represent a superoperator mirroring a big matrix, so mapping values to density matrices (that is, $Super\ a\ b \equiv (a, a) \to (b, b) \to K$). The function $lin2super$ constructs a superoperator from a linear operator on vectors. For instance:

$hadamardS \in Super\ Bool\ Bool$
$hadamardS = lin2super\ hadamard$

lifts the unitary operator $hadamard$ to a superoperator.

6.1.3 Tracing and Measurement

In contrast to the situation with the state vector model of quantum computing, it is possible to define a superoperator which "forgets," *projects*, or *traces out* part of a quantum state. Essentially, this corresponds to turn the dimension of the state space in consideration smaller. To do such an operation, we need first to understand how to *measure* part of

the quantum state which we would like to trace out. Measuring corresponds to setting the secondary diagonal of the density matrix to zero leaving only the classical probabilities corresponding to the possible measurement outputs in the main diagonal (same indexes).

$\mathbf{meas} \in Basis\ a \Rightarrow Super\ a\ (a, a)$

$\mathbf{meas}\ (a_1, a_2) = \mathbf{if}\ a_1 \equiv a_2\ \mathbf{then}\ return\ ((a_1, a_1), (a_1, a_1))\ \mathbf{else}\ mzero$

Note that we are considering *projective* measurements which are described by a set of projections onto mutually orthogonal subspaces. This kind of measurement returns a classical value and a post-measurement state of the quantum system. The operation $\mathbf{meas}$ is defined in such a way that it can encompass both results. Using the fact that a classical value m can be represented by the density matrix $|m\rangle\langle m|$ the superoperator $\mathbf{meas}$ returns the output of the measurement attached to the post-measurement state.

Now, it is easy to understand the operation below which forgets part of the quantum state. Before forgetting we measure:

$trL \in (Basis\ a, Basis\ b) \Rightarrow Super\ (a, b)\ b$

$trL\ ((a_1, b_1), (a_2, b_2)) = \mathbf{if}\ a_1 \equiv a_2\ \mathbf{then}\ return\ (b_1, b_2)\ \mathbf{else}\ mzero$

For example, the sequence:

$pureD\ qFT \gg\!= \mathbf{meas} \gg\!= trL$

first performs a measurement on the pure density matrix representing the vector qFT. This measurement produces a vector with two components: the first is the resulting collapsed quantum state and the second is the classical observed value. The last operation forgets about the collapsed quantum state and returns the result of the classical measurement. As explained earlier the resulting density matrix is:

$$\begin{pmatrix} 1/2 & 0 \\ 0 & 1/2 \end{pmatrix}$$

6.2 Why Density Matrices are not Monads?

Remember that for a type to be a monad, it would support the definition of

$\mathbf{class}\ Monad\ m\ \mathbf{where}$

$return \in forall\ a.a \to m\ a$

$(\gg\!=) \in forall\ a\ b.m\ a \to (a \to m\ b) \to m\ b$

But, as also explained for vectors is Section 5.1, because of the basis constraint over the types with which we build density matrices, the type constructor $Dens$ would corresponds in fact to a *indexed monad*, where:

$return \in (Basis\ a) \Rightarrow a \to Dens\ a$

$return = pureD.return$

However, the monadic bind operation needed to model the application of a superoperator to a density matrix can only be achieved with an operation instantiated to the following type:

$(\gg\!=) \in (Basis\ a, Basis\ b) \Rightarrow Dens\ a \to ((a, a) \to Dens\ b) \to Dens\ b$

$da \gg\!= s = \lambda(b_1, b_2) \to sum\ [(da\ (a_1, a_2)) * (s\ (a_1, a_2)\ (b_1, b_2))$
$\mid a_1 \leftarrow basis, a_2 \leftarrow basis]$

As one can observe, this type does not correspond to the required type for computations like s, which *consume multiple input values* (in this case (a, a)). This observation is reminiscent of Hughes's motivation for generalising monads to *arrows* (HUGHES, 2000) (see Section 3.2). Indeed, in addition to defining a notion of procedure which may perform computational effects, arrows may have a static component independent of the input, or may accept more than one input.

6.3 Superoperators as Indexed Arrows

Just as the probability effect associated with vectors is modelled by an *indexed monad* because of the $Basis$ constraint, the type $Super$ is modelled by an *indexed arrow*, as the following types include the additional constraint requiring the elements to form a set of observables:

$$
\begin{aligned}
&\textbf{instance } \mathit{IArrow}\ \mathit{Super} \textbf{ where}\\
&\quad \mathit{arr} \in (\mathit{Basis}\ b, \mathit{Basis}\ c) \Rightarrow (b \to c) \to \mathit{Super}\ b\ c\\
&\quad \mathit{arr}\ f = \mathit{fun2lin}\ (\lambda(b_1, b_2) \to (f\ b_1, f\ b_2))\\
&\quad (\ggg) \in (\mathit{Basis}\ b, \mathit{Basis}\ c, \mathit{Basis}\ d) \Rightarrow \mathit{Super}\ b\ c \to \mathit{Super}\ c\ d \to \mathit{Super}\ b\ d\\
&\quad (\ggg) = o\\
&\quad \mathit{first} \in (\mathit{Basis}\ b, \mathit{Basis}\ c, \mathit{Basis}\ d) \Rightarrow \mathit{Super}\ b\ c \to \mathit{Super}\ (b, d)\ (c, d)\\
&\quad \mathit{first}\ f\ ((b_1, d_1), (b_2, d_2)) = \mathit{permute}\ ((f\ (b_1, b_2))\langle * \rangle(\mathit{return}\ (d_1, d_2)))\\
&\qquad \textbf{where } \mathit{permute}\ v\ ((b_1, b_2), (d_1, d_2)) = v\ ((b_1, d_1), (b_2, d_2))
\end{aligned}
$$

The function *arr* constructs a superoperator from a pure function by applying the function to both the vector and its dual. The composition of arrows is simply the composition of linear operators (the operation o is defined in Section 5.2). The function *first* applies the superoperator f to the first component (and its dual) and leaves the second component unchanged. The definition calculates each part separately and then permutes the results to match the required type.

Proposition 6.3.1 *The indexed arrow $Super$ satisfies the required equations for arrows presented in Section 3.2.1.*

Proof. See Appendix C.

The proposition implies that we can use the arrow combinators to structure our quantum computations. For instance, the first few steps of the circuit for the Toffoli gate of Section 5.3 would now look like:

$$
\begin{aligned}
&\mathit{toffoli} \in \mathit{Super}\ (\mathit{Bool}, \mathit{Bool}, \mathit{Bool})\ (\mathit{Bool}, \mathit{Bool}, \mathit{Bool})\\
&\mathit{toffoli} = \textbf{let } \mathit{hadS} = \mathit{lin2super}\ \mathit{hadamard}\\
&\qquad\qquad \mathit{cphaseS} = \mathit{lin2super}\ (\mathit{controlled}\ \mathit{phase})\\
&\qquad\qquad \mathit{cnotS} = \mathit{lin2super}\ (\mathit{controlled}\ \mathit{qnot})\\
&\qquad\ \textbf{in } \mathit{arr}\ (\lambda(a_0, b_0, c_0) \to (c_0, (a_0, b_0))) \ggg\\
&\qquad\qquad (\mathit{first}\ \mathit{hadS} \ggg \mathit{arr}\ (\lambda(c_1, (a_0, b_0)) \to ((b_0, c_1), a_0))) \ggg\\
&\qquad\qquad (\mathit{first}\ \mathit{cphaseS} \ggg \mathit{arr}\ (\lambda((b_1, c_2), a_0) \to ((a_0, b_1), c_2))) \ggg\\
&\qquad\qquad (\mathit{first}\ \mathit{cnotS} \ggg \mathit{arr}\ (\lambda((a_1, b_2), c_2) \to ((b_2, c_2), a_1))) \ggg \ldots
\end{aligned}
$$

Clearly this notation is awkward as it forces us to explicitly manipulate the entire state and to manually permute the values. However, all the tedious code can be generated automatically as we explain next.

6.4 Examples: Toffoli and Teleportation

Using the arrow notation presented in Section 3.2.2, we express two well known quantum algorithms elegantly.

6.4.1 Toffoli

The following code mirrors the structure of the circuit and the structure of the monadic computation expressed earlier in Section 5.3:

$$
\begin{aligned}
&\mathit{toffoli} \in \mathit{Super}\ (\mathit{Bool}, \mathit{Bool}, \mathit{Bool})\ (\mathit{Bool}, \mathit{Bool}, \mathit{Bool})\\
&\mathit{toffoli} = \mathbf{let}\ \mathit{hadS} = \mathit{lin2super}\ \mathit{hadamard}\\
&\qquad\qquad \mathit{cnotS} = \mathit{lin2super}\ (\mathit{controlled}\ \mathit{qnot})\\
&\qquad\qquad \mathit{cphaseS} = \mathit{lin2super}\ (\mathit{controlled}\ \mathit{phase})\\
&\qquad\qquad \mathit{caphaseS} = \mathit{lin2super}\ (\mathit{controlled}\ (\mathit{adjoint}\ \mathit{phase}))\\
&\qquad \mathbf{in}\ \mathit{proc}\ (a_0, b_0, c_0) \rightarrow \mathbf{do}\\
&\qquad\qquad\qquad c_1 \leftarrow \mathit{hadS} \prec c_0\\
&\qquad\qquad\qquad (b_1, c_2) \leftarrow \mathit{cphaseS} \prec (b_0, c_1)\\
&\qquad\qquad\qquad (a_1, b_2) \leftarrow \mathit{cnotS} \prec (a_0, b_1)\\
&\qquad\qquad\qquad (b_3, c_3) \leftarrow \mathit{caphaseS} \prec (b_2, c_2)\\
&\qquad\qquad\qquad (a_2, b_4) \leftarrow \mathit{cnotS} \prec (a_1, b_3)\\
&\qquad\qquad\qquad (a_3, c_4) \leftarrow \mathit{cphaseS} \prec (a_2, c_3)\\
&\qquad\qquad\qquad c_5 \leftarrow \mathit{hadS} \prec c_4\\
&\qquad\qquad\qquad \mathit{returnA} \prec (a_3, b_4, c_5)
\end{aligned}
$$

Lets run *toffoli*, applied to the vector emt from Section 5.3 lifted to a density matrix:

$$
\begin{aligned}
&\mathit{demt} \in \mathit{Dens}\ (\mathit{Bool}, \mathit{Bool}, \mathit{Bool})\\
&\mathit{demt} = \mathit{pureD}\ \mathit{emt}\\
&\mathit{haskell}\rangle\, d \gg\!= \mathit{toffoli}
\end{aligned}
$$

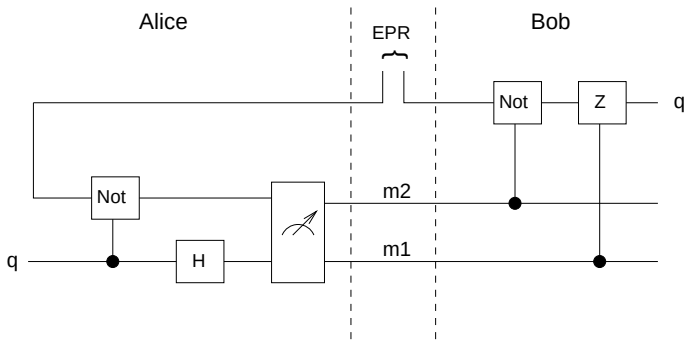
As expected, this produces the matrix:

$$
\begin{aligned}
&[(((\mathit{False}, \mathit{False}, \mathit{False}), (\mathit{False}, \mathit{False}, \mathit{False})), 0.5 :+ 0.0),\\
&\ (((\mathit{False}, \mathit{False}, \mathit{False}), (\mathit{True}, \mathit{True}, \mathit{False})), 0.5 :+ 0.0),\\
&\ (((\mathit{True}, \mathit{True}, \mathit{False}), (\mathit{False}, \mathit{False}, \mathit{False})), 0.5 :+ 0.0),\\
&\ (((\mathit{True}, \mathit{True}, \mathit{False}), (\mathit{True}, \mathit{True}, \mathit{False})), 0.5 :+ 0.0)]
\end{aligned}
$$

6.4.2 Quantum Teleportation

The idea of quantum teleportation is to make disappear a quantum information (quantum state) in one place making a perfect replica of it somewhere else. Indeed quantum teleportation (BENNETT et al., 1993) enables the transmission, *using a classical communication channel*, of an unknown quantum state via a previously shared epr pair.

In the following diagram, Alice and Bob initially have access to one of the qubits of an entangled epr pair, and Alice aims to teleport an unknown qubit q to Bob:



The calculation proceeds as follows. First Alice interacts with the unknown qubit q and her half of the epr state. Then Alice performs a measurement collapsing her quantum state and getting two classical bits m_1 and m_2 that she transmits to Bob using a classical channel of communication.

Upon receiving the two classical bits of information, Bob interacts with his half of the epr state with gates controlled by the classical bits. The circuit in the figure can be shown to re-create the quantum state q which existed at Alice's site before the experiment.

Our main interest in this circuit is that it is naturally expressed using a sequence of operations on quantum values which include a non-unitary *measurement* in the middle.

We use the machinery we have developed to express the teleportation circuit. We break the algorithm in two individual procedures, $alice$ and bob. Besides the use of the arrows notation to express the action of superoperators on specific qubits, we incorporate the measurement in Alice's procedure, and trace out the irrelevant qubits from the answer returned by Bob.

$$
\begin{aligned}
&alice \in Super\ (Bool, Bool)\ (Bool, Bool)\\
&alice = proc\ (eprL, q) \rightarrow \mathbf{do}\\
&\qquad (q_1, e_1) \leftarrow (lin2super\ (controlled\ qnot)) \prec (q, eprL)\\
&\qquad q_2 \leftarrow (lin2super\ hadamard) \prec q_1\\
&\qquad ((q_3, e_2), (m_1, n_1)) \leftarrow \mathbf{meas} \prec (q_2, e_1)\\
&\qquad (m_2, n_2) \leftarrow trL\ ((q_3, e_2), (m_1, n_1))\\
&\qquad returnA \prec (m_2, n_2)\\
&bob \in Super\ (Bool, Bool, Bool)\ Bool\\
&bob = proc\ (eprR, m_1, n_1) \rightarrow \mathbf{do}\\
&\qquad (n_2, e_1) \leftarrow (lin2super\ (controlled\ qnot)) \prec (n_1, eprR)\\
&\qquad (m_2, e_2) \leftarrow (lin2super\ (controlled\ zgate)) \prec (m_1, e_1)\\
&\qquad q' \leftarrow trL \prec ((m_2, n_2), e_2)\\
&\qquad returnA \prec q'\\
&teleport \in Super\ (Bool, Bool, Bool)\ Bool\\
&teleport = proc\ (eprL, eprR, q) \rightarrow \mathbf{do}\\
&\qquad (m_1, n_1) \leftarrow alice \prec (eprL, q)\\
&\qquad q' \leftarrow bob \prec (eprR, m_1, n_1)\\
&\qquad returnA \prec q'
\end{aligned}
$$

As an example, suppose we want to teleport the state $1/\sqrt{2}(|0\rangle + |1\rangle)$. Then the three qubits state to be passed to the procedure $teleport$ is $1/\sqrt{2}(|0\rangle+|1\rangle)\otimes 1/\sqrt{2}(|00\rangle+|11\rangle) = 1/2(|000\rangle + |011\rangle + |100\rangle + |111\rangle)$, pretty printed in Haskell as:

$$
\begin{aligned}
&gtele \in Vec\ (Bool, Bool, Bool)\\
&gtele = [((True, False, False), 1\ /\ 2),\\
&\qquad ((True, True, True), 1\ /\ 2),\\
&\qquad ((False, False, False), 1\ /\ 2),\\
&\qquad ((False, True, True), 1\ /\ 2)]\\
&dgtete \in Dens\ (Bool, Bool, Bool)\\
&dgtele = pureD\ gtele
\end{aligned}
$$

The application of the procedure recreates the density matrix for the qubit one wants to teleport:

$$
\begin{aligned}
&haskell\rangle dgtele \ggg teleport\\
&[((False, False), 0.5 :+ 0.0),\\
&\ ((False, True), 0.5 :+ 0.0),\\
&\ ((True, False), 0.5 :+ 0.0),\\
&\ ((True, True), 0.5 :+ 0.0)]
\end{aligned}
$$

6.5 Typing Rules

The category of superoperators is considered to be an adequate model of non-reversible quantum computation (SELINGER, 2004). Our construction presented so far seems to suggest that this category corresponds to a functional language with arrows, and so that we can accurately express quantum computation in such a framework. But as we explain below, this is not quite the whole story.

First consider the well-known "non-cloning" property of quantum states (NIELSEN; CHUANG, 2000). The arrow notation allows us to reuse variables more than once, and we are free to define the following operator:

$$copy \in Super\ Bool\ (Bool, Bool)$$
$$copy = arr\ (\lambda x \rightarrow (x, x))$$

But can this superoperator be used to clone a qubit? The answer, as explained in Section 1.3.5 of the classic book on quantum computing (NIELSEN; CHUANG, 2000), is no. The superoperator $copy$ can be used to copy classical information encoded in quantum data, but when applied to an arbitrary quantum state, for example qFT, the superoperator does not make two copies of the state qFT but rather it produces the epr state which is the correct and desired behaviour. Thus, in this aspect the semantics of arrows is coherent with quantum computation, *i.e.*, the use of variables more than once models entanglement, not cloning.

In contrast, in our model there is nothing to prevent the definition of:

$$weaken \in Super\ (Bool, Bool)\ Bool$$
$$weaken = arr\ (\lambda(x, y) \rightarrow y)$$

This operator is however not physically realizable. Applying $weaken$ to epr gives qFT. Physically forgetting about x corresponds to a measurement: if we measure the left qubit of epr we should get $qFalse$ or $qTrue$ or the mixed state of both measurements, but never qFT.

This suggests that we need something more than only the arrow combinators to make our quantum computations compatible with Quantum Mechanics. We propose a simple type system for the arrow combinators. First, suppose we have choosen a univeral set of unitary operations $\mathcal{U}$.

$$\frac{u : Lin\ b\ c\ \in \mathcal{U}}{lin2super\ u : Super\ b\ c}\ \textbf{Lift}$$

$$\frac{b \in Basis \quad c \in Basis \quad f : b \rightarrow c}{arr\ f : Super\ b\ c}\ arr \quad \textbf{if } \dim(b) \leqslant \dim(c)$$

$$\frac{f : Super\ b\ c \quad g : Super\ c\ d}{f \ggg g : Super\ b\ d}\ \ggg$$

$$\frac{f : Super\ b\ c \quad d \in Basis}{first\ f : Super\ (b, d)\ (c, d)}\ \text{first}$$

Figure 6.1: Typing arrow combinators for quantum computations

The critical typing rule is arr which selects only a class of *basic functions* to be lifted to superoperators, i.e., those ones which do not forget variables. That is imposed by the restriction $dim(b) \leqslant dim(c)$.

6.6 Summary

We have argued that a realistic model for quantum computations should accommodate both unitary operations and measurements, and we have shown that such general quantum computations can be modelled using *indexed arrows*. This is an extension of the previously-known observation that one can model pure quantum probabilities using indexed monads. Establishing such connections between quantum computations and monads and arrows enables elegant embeddings in current classical languages, and exposes connections to well-understood concepts from the semantics of (classical) programming languages.

We also have demonstrated the use of indexed arrows to elegantly model two examples in Haskell, including the teleportation experiment which interleaves measurements with unitary operations. However, for the case of teleportation, this model is not faithful. The procedure $alice$ is implemented such that at the end of its processing there is a projective measurement of her two qubits. Everything is very well except the fact *there is no classical information explicitly being communicated* between Alice and Bob. That is m_1 and n_1 *are classical values represented by the density matrices* $|m1\rangle\langle m_1|$ and $|n_1\rangle\langle n_1|$, respectively. That is, they are classical information represented in a form typically used to represent quantum information. Yet as noted by (GAY; NAGARAJAN, 2005; UNRUH, 2005) a *complete* model for expressing quantum algorithms should accommodate both measurements and combined interactions of quantum and *classical data*. In the next two chapters we propose to structure two alternative general (involving measurements) and complete (involving both quantum and classical data) approaches for *combined* quantum and classical computations as arrows.

7 MODELLING QUANTUM EFFECTS III: MIXED PRO - GRAMS WITH DENSITY OPERATORS AND CLASSICAL OUTPUTS AS INDEXED ARROWS

The model presented in last chapter is purely quantum. However, various quantum algorithms are explained in terms of the *interchanging* of quantum and classical information [1]. For instance, quantum teleportation is a traditional example of an algorithm which is based on two quantum process communicating via *classical data*. There is interest to consider a *mixed* model for quantum computations involving *measurements* and the *information flow* between quantum and classical processes (for instance, see (RAUSSENDORF; BROWNE; BRIEGEL, 2003; KASHEFI; PANANGADEN; DANOS, 2004; NIELSEN, 2003; GAY; NAGARAJAN, 2005; UNRUH, 2005)).

On the other hand, the finding of a representation that is suitable for representing both the results of unitary transformations and measurement operations should also be put into perspective.

That is, we would like that the same representational framework be able to take care of both: (1) the task of representing the *quantum state* resulting from a unitary operation applied to a given quantum state, and (2) the task of representing the pair of information coming out from a measurement, namely: (2a) that corresponding to the *measurement value* produced by the measurement (one of the eigen-values of the measurement operator), and (2b) the *quantum state* that results from the projection imposed on the original quantum state by the measurement (one of the eigen-vectors of the measurement operator).

The main problem introduced by the need of that uniformity is that measurement results (both value and state results) are of a probabilistic kind, needing *sets of possible results* for their representation. The usual alternative solution to such problem is the density matrix formalism.

Hence, in this Chapter we present a model for *mixed* or *combined* quantum computations based on a measurement approach over density matrices. We call mixed or combined quantum computation any computation transforming a combined state, with classical and quantum data. Essentially, the idea is to have a density operator representing the (global) quantum part, and a probability distribution of classical values representing the classical part of the state. A quantum program acting on this combined state is interpreted by a special *tracing superoperator*, which in the general case traces out part of the state, returning a classical output, and leaving the system in a new state (possibly in a space with reduced dimension).

[1] By interchanging we mean, for instance, a measurement in the middle of the computation.

7.1 Mixed Programs with Density Matrices

Because the tracing superoperator in general *forgets* part of the state, we define a relation between bases which we call Dec (from *decomposition*):

$\mathbf{class}\ (Basis\ a, Basis\ b) \Rightarrow Dec\ a\ b\ \mathbf{where}$
$\quad dec \in [a] \rightarrow [b]$

specifying that a can be decomposed in a part b. Then, a quantum program from a to b, parameterized by i, the type of the input classical probability distribution, and o, the part to be measured, is represented by a superoperator from a to b, delivering a classical probability distribution over o, the part of a which is measured.

$\mathbf{type}\ DProb\ c = [(c, Prob)]$
$\mathbf{type}\ QProgram\ i\ o\ a\ b = (DProb\ i, (a, a)) \rightarrow (DProb\ o, Dens\ b)$

Note that our quantum programs should satisfy the restriction $Dec\ a\ o$, and that $DProb\ i$ is used in classical operations or quantum operations controlled by classical data.

We can lift density matrices to combined states by:

$dens2combst \in (Basis\ a) \Rightarrow Dens\ a \rightarrow (DProb\ (), Dens\ a)$
$dens2combst\ d = ([\,], d)$

Suppose $qFTD$ as presented in Section 6.1.1, then $dens2combst\ qFTD$ produces:

$([\,], [(((False, False), (False, False)), 0.5 :+ 0.0),$
$\quad (((False, False), (True, True)), 0.5 :+ 0.0),$
$\quad (((True, True), (False, False)), 0.5 :+ 0.0),$
$\quad (((True, True), (True, True)), 0.5 :+ 0.0)])$

which is a combined state with an empty classical part.

As any type can be decomposed into the *unit* $()$, and can be decomposed into itself, and also can be decomposed into one of its parts, we have the following instances:

$\mathbf{instance}\ (Basis\ a) \Rightarrow Dec\ a\ ()\ \mathbf{where}$
$\quad dec\ \_ = [\,]$
$\mathbf{instance}\ (Basis\ a) \Rightarrow Dec\ a\ a\ \mathbf{where}$
$\quad dec\ l = l$
$\mathbf{instance}\ (Basis\ a, Basis\ b) \Rightarrow Dec\ (a, b)\ b\ \mathbf{where}$
$\quad dec\ [\,] = [\,]$
$\quad dec\ ((x, y) : l) = y : dec\ l$

Any unitary operator, as represented in Section 5.2, can be lifted to a quantum program which traces out $()$.

$uni2qprog \in (Basis\ a, Basis\ b, Basis\ i, Sub\ a\ ()) \Rightarrow$
$\qquad\qquad\qquad\qquad\qquad\qquad Lin\ a\ b \rightarrow QProgram\ i\ ()\ a\ b$
$uni2qprog\ f\ (dp, (a_1, a_2)) = \mathbf{let}\ d = lin2vec\ (f\ a_1)\rangle\!*\!\langle(f\ a_2)$
$\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathbf{in}\ (d, [\,])$

The function $uni2qprog$ constructs a mixed quantum program from a function representing a unitary operator. The idea is to apply the default construction to build a superoperator from a unitary transformation (see Section 6.1.2). Note that the classical input is ignored and the classical output is empty: there is no interaction with the classical world when considering unitary transformations.

For instance:

$hadamardP \in QProgram\ i\ ()\ Bool\ Bool$
$hadamardP = uni2qprog\ hadamard$

lifts the unitary operator $hadamard$ to a quantum program acting on a combined state.

Given a quantum state over a basis set (a, b), the quantum program trR forgets the *right* component, returning a new state over b. The subspace is measured before being discarded outputting a classical probability distribution over the basis which forms that subspace. In this case, the input classical data is just ignored.

$$
\begin{aligned}
&trR \in (Basis\ a, Basis\ b, Basis\ i) \Rightarrow QProgram\ i\ b\ (a, b)\ a \\
&trR\ (dp, ((a_1, b_1), (a_2, b_2))) = \mathbf{let}\ d = \mathbf{if}\ b_1 \equiv b_2\ \mathbf{then}\ vreturn\ (a_1, a_2) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{else}\ vzero \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad p = [(b_1, 1) \mid b_1 \equiv b_2] \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{in}\ (p, d) \\
&trA \in (Basis\ a, Basis\ i) \Rightarrow QProgram\ i\ a\ a\ () \\
&trA\ (dp, (a_1, a_2)) = \mathbf{let}\ d = \mathbf{if}\ a_1 \equiv a_2\ \mathbf{then}\ vreturn\ ((), ())\ \mathbf{else}\ vzero \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad p = [(a_1, 1) \mid a_1 \equiv a_2] \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{in}\ (p, d)
\end{aligned}
$$

Similarly, the program trA forgets (measures) all quantum state returning only a classical probability distribution as the result. To construe the classical probability distribution we consider that any value from the type being measured *can* appear in the output quantum state. Hence each value from the basis is attached to the probability 1. The real probability to appear in the final state is calculated by the function app below, which given a *mixed* program and a *combined* state calculates the new density matrix and the classical result (if there is some).

$$
\begin{aligned}
&app \in (Basis\ a, Basis\ b, Basis\ i, Basis\ o, Sub\ a\ o) \Rightarrow \\
&\qquad\qquad\qquad ((DProb\ i, (a, a)) \rightarrow (DProb\ o, Dens\ b)) \rightarrow \\
&\qquad\qquad\qquad (DProb\ i, Dens\ a) \rightarrow (DProb\ o, Dens\ b) \\
&app\ p\ (d_i, da) = \mathbf{let}\ fdb = [(b, sum\ [\mathbf{let}\ (po, db) = p\ (d_i, a) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad p_2 = vlookup\ b\ db \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad p_1 = vlookup\ a\ da \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{in}\ p_1 * p_2 \mid a \leftarrow basis]) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mid b \leftarrow basis] \\
&\qquad\qquad\qquad\qquad\qquad\qquad l = map\ (\lambda a \rightarrow\ \mathbf{let}\ pp = vlookup\ (a, a)\ da \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{in\ if}\ pp \not\equiv 0\ \mathbf{then}\ p\ (d_i, (a, a)) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathbf{else}\ ([\,], [\,]))\ basis \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (lp, ld) = unzip\ l \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad flp = concat\ (filter\ (\lambda a \rightarrow \mathbf{if}\ (a \equiv [\,]) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathbf{then}\ False \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathbf{else}\ True)\ lp) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{in}\ (flp, dbf)
\end{aligned}
$$

The output density matrix fdb (from *final density* of type b) is calculated by simple matrix multiplication: the superoperator matrix by the input density matrix. Note that the overall operation may depend on the classical state. Then, the final probability distribution of classical values flp is calculated by analysing the inputted density matrix d_i and by applying p to the observables in d_i.

7.2 Mixed Programs with Density Matrices as Indexed Arrows

We define the three functions, arr, $\ggg$, and $first$, over $QProgram\ i\ o$ as follows:

$$
\begin{aligned}
&arr \in (Basis\ b, Basis\ c, Sub\ b\ ()) \Rightarrow (b \rightarrow c) \rightarrow QProgram\ i\ ()\ b\ c \\
&arr = uni2qprog.fun2lin \\
&(\ggg) \in (Basis\ a, Basis\ b, Basis\ c, Basis\ i, Basis\ o,
\end{aligned}
$$

$$
\begin{aligned}
&\qquad\qquad\qquad\qquad Basis\ o_2, Sub\ a\ o, Sub\ b\ o_2) \Rightarrow \\
&\quad QProgram\ i\ o\ a\ b \to QProgram\ o\ o_2\ b\ c \to QProgram\ i\ o_2\ a\ c \\
&(f \ggg g)\ (dpi, (a_1, a_2)) = app\ g\ (f\ (dpi, (a_1, a_2))) \\
&first \in (Basis\ a, Basis\ b, Basis\ c, Basis\ i, Basis\ o, Sub\ a\ o, Sub\ (a, c)\ o) \Rightarrow \\
&\qquad\qquad QProgram\ i\ o\ a\ b \to QProgram\ i\ o\ (a, c)\ (b, c) \\
&first\ p\ (pi, ((a_1, c_1), (a_2, c_2))) = \\
&\quad \mathbf{let}\ (po, db) = p\ (pi, (a_1, a_2)) \\
&\qquad vcc = vreturn\ (c_1, c_2) \\
&\qquad dbc = [(((b_1, c_1), (b_2, c_2)), db\ (b_1, b_2) * vcc\ (c_1, c_2)) \mid \\
&\qquad\qquad\qquad\qquad\qquad\qquad ((b_1, c_1), (b_2, c_2)) \leftarrow basis] \\
&\quad \mathbf{in}\ (po, dbc)
\end{aligned}
$$

The function arr just lifts basic functions to *mixed* programs. The classical input is ignored and the classical output is empty. The function $\ggg$ composes two mixed programs, and *first* augments the state space of the quantum part of a mixed program by c. The action of the new augmented mixed program is defined as the original program over a with identity over c.

Proposition 7.2.1 *The indexed arrow $QProgram\ i\ o$ satisfies the required equations for arrows.*

7.3 Example: Teleportation

Recall the teleportation circuit in Section 6.4.2. Remember that the original algorithm involves a measurement in the middle of the computation and communication of classical information (the classical result of the measurement) between Alice (the sender) and Bob (the receiver). Using this model of mixed programs acting on combined states structured as arrows we can faithfully express the teleportation algorithm as follows. The main procedure receives no classical data and three entangled qubits; then passes a qubit of the epr pair and the qubit to be teleported to Alice, which realizes some quantum operations and measures its two qubits, returning only classical values to the main procedure, which will be communicated to Bob.

$$
\begin{aligned}
&teleportation \in QProgram\ ()\ ()\ (Bool, Bool, Bool)\ Bool \\
&teleportation = proc\ (eprL, eprR, q) \to \mathbf{do} \\
&\qquad\qquad cs \leftarrow alice \prec (eprL, q) \\
&\qquad\qquad q' \leftarrow bob \prec (eprR, cs) \\
&\qquad\qquad returnA \prec q' \\
&alice \in QProgram\ ()\ (Bool, Bool)\ (Bool, Bool)\ () \\
&alice = proc\ (eprL, q) \to \mathbf{do} \\
&\qquad (q_1, e_1) \leftarrow qcnotP \prec (q, eprL) \\
&\qquad q_2 \leftarrow hadamardP \prec q_1 \\
&\qquad cs \leftarrow trA \prec (e_1, q_2) \\
&\qquad returnA \prec cs
\end{aligned}
$$

where $qcnotP$ is the mixed version of the linear operator $controlled\ \ qnot$ defined in Section 5.2.

$$
\begin{aligned}
&qcnotP \in QProgram\ ()\ ()\ (Bool, Bool)\ (Bool, Bool) \\
&qcnotP = uni2qprog\ (controlled\ qnot)
\end{aligned}
$$

Bob is a procedure which receives a classical data over $(Bool, Bool)$ and a qubit. The procedure analyses the classical data and depending on its value applies or not a certain

quantum operation to the input qubit.

$$
\begin{aligned}
&bob \in QProgram\ (Bool, Bool)\ ()\ ((), Bool)\ Bool\\
&bob = \lambda(pbb, db) \rightarrow \mathbf{let}\ (p_1, d_1) = \mathbf{if}\ (lookup\ True\ (unzipL\ pbb)\ pbb > 0)\\
&\qquad\qquad\qquad\qquad\qquad\quad \mathbf{then}\ (qnotP\ ([((), 1)], db))\\
&\qquad\qquad\qquad\qquad\qquad\quad \mathbf{else}\ ([((), 1)], vreturn\ db)\\
&\qquad\qquad\qquad\quad (p_2, d_2) = \ \mathbf{if}\ (lookup\ True\ (unzipR\ pbb) > 0)\\
&\qquad\qquad\qquad\qquad\qquad\quad \mathbf{then}\ (zgateP\ ([((), 1)], db))\\
&\qquad\qquad\qquad\qquad\qquad\quad \mathbf{else}\ st_1\\
&\qquad\qquad\quad \mathbf{in}\ (p_2, d_2)
\end{aligned}
$$

Again we are using a mixed version of a linear operator defineded in Section 5.2

$$
\begin{aligned}
&zgateP \in QProgram\ ()\ ()\ Bool\ Bool\\
&zgateP = uni2prog\ zgate
\end{aligned}
$$

The functions $unzipL$ and $unzipR$ take a list of tuples and return a list with the left elements of the tuples and a list with the right elements of the tuples, respectively.

$$
\begin{aligned}
&unzipL \in [((a, b), p)] \rightarrow [a]\\
&unzipL\ l = \mathbf{let}\ (lb, lp) = unzip\ l\\
&\qquad\qquad\quad (las, lbs) = unzip\ lb\\
&\qquad\qquad \mathbf{in}\ las
\end{aligned}
$$

$$
\begin{aligned}
&unzipR \in [((a, b), p)] \rightarrow [b]\\
&unzipR\ l = \mathbf{let}\ (lb, lp) = unzip\ l\\
&\qquad\qquad\quad (las, lbs) = unzip\ lb\\
&\qquad\qquad \mathbf{in}\ lbs
\end{aligned}
$$

7.4 Summary

In this Chapter we introduced a model for mixed quantum computations acting on a combined state with a quantum and a classical part. The quantum part of the state is represented by a density matrix which can efficiently express the probabilistic distribution of quantum states resulting from measurements. We justified the importance of using mixed programs and combined states based on the structure of some important quantum algorithms like teleportation.

However, there is a (possibly not minor) conceptual problem in the adoption of the density matrix formalism, namely: a density matrix is supposed to represent a set (*ensemble*) of quantum systems whose probability distribution of states the density matrix represents; however, from a programming theoretic point of view, one usually thinks of a quantum algorithm as being performed by one single quantum system, not an ensemble of quantum systems each possibly behaving in a different way according to a probability distribution.

We feel that the quantum programmer's intuition of programming one single quantum system at a time, while elaborating his algorithms, may happen to be not appropriately captured by the density matrix formalism. We feel (but we have no definite argument) that a representation modelled on the usual set-theoretic representation of states of non-deterministic machines, adjusted to explicitly represent the probability of occurrence of each deterministic state, may happen to capture in a better way the quantum programmer's intuition.

So, in next Chapter we introduce another way of dealing with *mixed* quantum computations, which is based on explicit probability distributions over sets of quantum states.

8 MODELLING QUANTUM EFFECTS IV: MIXED PRO - GRAMS WITH PROBABILITY DISTRIBUTIONS OF QUAN- TUM VECTORS STATES AS ARROWS

As motivated in the previous chapter before, we present in this chapter another way of representing combined states. Basically, the quantum part of the combined state is represented by an explicit probability distribution over quantum states.

The idea is to have a combined state, where the classical part is as before (i.e. a probability distribution of classical values), and the quantum part is represented by this explicity probability distribution over quantum states. A mixed program acting on this combined state can act on the quantum part, on the classical part, or on both parts.

Mixed programs acting only on quantum data are of two kinds: i) the unitary transformations, which reversibly transform the state vector and nothing happens to the classical probability; and ii) measurements, which probabilistically yield one of the *eigenvalues* of the observable being measured, and *throws* the system into the correspondent *eigenstate*. Yet one can have quantum operations controlled by classical values as well as purely classical operations.

8.1 Mixed Programs with Probability Distributions

The probabilistic quantum programming model that we define is based on data type to represent *probability distributions of quantum state vectors*:

$\mathbf{type}\ EV = Double$
$\mathbf{type}\ Prob = Double$
$\mathbf{newtype}\ PDQst\ a = PDQ\{\, unPDQ \in [([EV],\ Vec\ a, Prob)]\}$

More specifically, a probability distribution over a basis set a is represented by a pair formed by: a list of real values EV, the eigenvalues which are the outputs of previously performed measurements, and a state vector, $Vec\ a$. We chose to keep a list of eigenvalues EV to maintain a history of measurements. For now this list does not include information about the source of eigenvalues, i.e., about the position of the qubit which was measured in the global state.

An example of simple distribution over a basic vector may be defined as follows:

$return \in (Basis\ a) \Rightarrow a \rightarrow PDQst\ a$
$return\ a = PDQ\ [([\,], return\ a, 1)]$
$qdFalse \in PDQst\ Bool$
$qdFalse = return\ False$

Also we can define basic distributions over simple superpositions and over n-dimensional vectors:

$qdFT \in PDQst\ Bool$
$qdFT = PDQ\ [([\,], qFT, 1)]$
$eprd \in PDQst\ (Bool, Bool)$
$eprd = PDQ\ [([\,], epr, 1)]$

Note that the list of eigenvalues is empty for basic distributions. This is because the eigenvalues start to appear in distributions only after we have measurements involved.

A *mixed quantum program* is represented by two kinds of transformations:

$$\begin{aligned}&\textbf{data}\ PDQTrans\ a\ b = Transform\ ((PDQst\ a) \to (PDQst\ b))\\ &\qquad\qquad\qquad\qquad\mid Meas\ ((PDQst\ a) \to (PDQst\ b))\end{aligned}$$

We made the difference explicit because the semantics of applying unitary transformations is different from the semantics of applying measurements.

A simple unitary transformation can be defined in such a way that the transformation is applied to all vectors in the distribution. The probability distribution over eigenvalues is preserved. For instance, a simple quantum unitary transformation as $hadamard$ can be defined as:

$$\begin{aligned}&hadamardD \in PDQTrans\ Bool\ Bool\\ &hadamardD =\\ &\quad Transform\ (\lambda x \to PDQ\ [(l2_1, v_2, p_1) \mid (l2_1, v_1, p_1) \leftarrow unPDQ\ x,\\ &\qquad\qquad\qquad\qquad\qquad\qquad\textbf{let}\ v_2 = v_1 \gg\!= hadamard])\end{aligned}$$

We can test the function above using an application operation:

$$\begin{aligned}&appD \in (Basis\ a, Basis\ b) \Rightarrow PDQTrans\ a\ b \to PDQst\ a \to PDQst\ b\\ &appD\ f = \lambda x \to f\ x\end{aligned}$$

Applying $hadamardD$ to $qdFT$ produces:

$$([\,], [(False, 1.0 :+ 0.0), (True, 0.0 :+ 0.0)]) \to 1.0$$

using a pretty printing for $PDQst$: on the left are the list of eigenvalues and the vector, and on the right, after the arrow, is the probability.

Measurements are the operations which produce eigenvalues as *classical outputs* and return a new classical probability distribution over eigenstates of the observable according to *each* vector in the distribution.

The measurement of a simple qubit realized by an observable, which has $qFalse$ ($|0\rangle$) and $qTrue$ ($|1\rangle$) as its eigenvectors, can be implemented as follows:

$$\begin{aligned}&measqD \in PDQTrans\ Bool\ Bool\\ &measqD = Meas\ (\lambda x \to PDQ\ [(evalue : l2_1, evector, p_2) \mid\\ &\qquad(l2_1, v_1, p_1) \leftarrow unPDQ\ x,\\ &\qquad(evalue, evector) \leftarrow [(0, qFalse), (1, qTrue)],\\ &\qquad\textbf{let}\ p_2 = \textbf{if}\ p_1 \not\equiv 0\\ &\qquad\qquad\textbf{then}\ (((magnitude\ (evector\langle.\rangle v_1)) **\ 2) * p_1)\ \textbf{else}\ 0])\end{aligned}$$

The two possible outputs are 0 collapsing the vector to $qFalse$ or 1 collapsing the vector to $qTrue$. Note that the new probability p_2 is calculated using the formula presented in Section 2.1.3 multiplied by the previous probability as it is a dependent event. This operation may augment the number of vectors in the distribution, for instance $appD\ measqD\ qdFT$ returns:

$$\begin{aligned}&([0.0], [(False, 1.0 :+ 0.0)]) \to 0.5\\ &([1.0], [(True, 1.0 :+ 0.0)]) \to 0.5\end{aligned}$$

Moreover we can define a function which discards a qubit,

$$discqD \in PDQTrans\ Bool\ ()$$

Of course, discarding a qubit physically corresponds to measuring it, returning a real value for the probability distribution. The definition of this function is similar to $measqD$

except for the fact that there is no vector returned.

8.2 $PDQTrans$ as Indexed Arrows

We define the three functions, arr, $\ggg$, and *first*, over $PDQTrans$ as follows:

$$
\begin{aligned}
&arr \in (Basis\ b_1, Basis\ b_2) \Rightarrow (b_1 \to b_2) \to PDQTrans\ b_1\ b_2\\
&arr\ f = Transform\ (\lambda x \to PDQ\ [(e_1, v_2, p) \mid (e_1, v_1, p) \leftarrow unPDQ\ x,\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{let}\ fv = fun2vecfun\ f,\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{let}\ v_2 = fv\ v_1])\\
&(\ggg) \in (Basis\ b_1, Basis\ b_2, Basis\ b_3) \Rightarrow\\
&\qquad PDQTrans\ b_1\ b_2 \to PDQTrans\ b_2\ b_3 \to PDQTrans\ b_1\ b_3\\
&(Transform\ f) \ggg (Transform\ g) = Transform\ (\lambda x \to \mathbf{let}\ d = f\ x\ \mathbf{in}\ g\ d)\\
&(Meas\ f) \ggg (Transform\ g) = Transform\ (\lambda x \to \mathbf{let}\ d = f\ x\ \mathbf{in}\ g\ d)\\
&(Transform\ f) \ggg (Meas\ g) = Meas\ (\lambda x \to \mathbf{let}\ d = f\ x\ \mathbf{in}\ g\ d)\\
&(Meas\ f) \ggg (Meas\ g) = Meas\ (\lambda x \to \mathbf{let}\ d = f\ x\ \mathbf{in}\ g\ d)\\
&first \in (Basis\ b_1, Basis\ b_2, Basis\ b_3) \Rightarrow\\
&\qquad\qquad PDQTrans\ b_1\ b_2 \to PDQTrans\ (b_1, b_3)\ (b_2, b_3)\\
&first\ (Transform\ f) =\\
&Transform\ (\lambda x \to \mathbf{let}\ fg = getvbs\ (Transform\ f)\\
&\qquad\qquad\qquad\qquad fext = firstl\ fg\\
&\qquad\qquad\qquad \mathbf{in}\ PDQ\ [(le, v, p) \mid (l2_1, v_1, p_1) \leftarrow unPDQ\ x,\\
&\qquad\qquad\qquad\qquad\qquad \mathbf{let}\ (le, v, p) = \quad (l2_1, [((b, c), k_1 * k_2) \mid\\
&\qquad\qquad\qquad\qquad\qquad ((a, c), k_1) \leftarrow \quad v_1, \mathbf{let}\ d_2 = fext\ (a, c),\\
&\qquad\qquad\qquad\qquad\qquad (le_2, v_2, p_2) \leftarrow unPDQ\ d_2,\\
&\qquad\qquad\qquad\qquad\qquad ((b, c), k_2) \leftarrow \quad v_2], p_1)])\\
&first\ (Meas\ f) =\\
&Meas\ (\lambda x \to \mathbf{let} \quad fg = getvbs\ (Meas\ f)\\
&\qquad\qquad\qquad\qquad fext = firstl\ fg\\
&\qquad\qquad\qquad\quad \mathbf{in}\ zipqd\ (PDQ\ [(le, v, p) \mid (l2_1, v_1, p_1) \leftarrow unPDQ\ x,\\
&\qquad\qquad\qquad\qquad\qquad ((a, c), k_1) \leftarrow v_1, \mathbf{let}\ d_2 = fext\ (a, c),\\
&\qquad\qquad\qquad\qquad\qquad (le_2, v_2, p_2) \qquad\quad \leftarrow unPDQ\ d_2,\\
&\qquad\qquad\qquad\qquad\qquad \mathbf{let}\ (le, v) = (le_2 +\!\!+ l2_1,\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad [((b, c), k_1 * k_2) \mid ((b, c), k_2) \leftarrow v_2]),\\
&\qquad\qquad\qquad\qquad\qquad \mathbf{let}\ p = p_1 * p_2 * (((**2).magnitude)\ k_1)]))
\end{aligned}
$$

The first two functions are straightforward: arr constructs a ***reversible*** transformation from a basic function, where

$$
\begin{aligned}
&fun2vecfun \in (Basis\ a, Basis\ b) \Rightarrow (a \to b) \to (Vec\ a \to Vec\ b)\\
&fun2vecfun\ f\ va = \mathbf{let}\ fa = fun2lin\ f\\
&\qquad\qquad\qquad\quad \mathbf{in}\ va \gg\!= fa
\end{aligned}
$$

converts a "matrix" to a function mapping vectors to vectors, and $\ggg$ just composes two $PDQTrans$. The function *first* is a bit more subtle, the idea is to transform a function which acts in ***part*** of a quantum state (say $Vec\ b_1$) to a function which acts in the ***global*** state (say $Vec\ (b_1, b_3)$). The implementation is based in the following two functions [1]:

$$
\begin{aligned}
&getvbs \in PDQTrans\ a\ b \to (a \to PDQst\ b)\\
&getvbs\ (Transform\ f) = \lambda a \to \mathbf{let}\ d = dreturn\ a\ \mathbf{in}\ f\ d\\
&getvbs\ (Meas\ f) = \lambda a \to \mathbf{let}\ d = dreturn\ a\ \mathbf{in}\ f\ d
\end{aligned}
$$

[1] The function $vlk\ a\ v$ just lookups the amplitude probability of a in vector v.

$$
\begin{aligned}
&\mathit{firstbs} \in (a \to \mathit{PDQst}\ b) \to (a, c) \to \mathit{PDQst}\ (b, c)\\
&\mathit{firstl}\ f\ (a, c) = \mathbf{let}\ db = f\ a\\
&\qquad\qquad\qquad\qquad dc = \mathit{dreturn}\ c\\
&\qquad\qquad\qquad\quad \mathbf{in}\ \mathit{PDQ}\ [(le, v_2, p * q) \mid (le, vb, p) \leftarrow \mathit{unPDQ}\ db,\\
&\qquad\qquad\qquad\qquad (\_, vc, q) \leftarrow \mathit{unPDQ}\ dc,\\
&\qquad\qquad\qquad\qquad \mathbf{let}\ v_2 = [((b, c), \mathit{vlk}\ b\ vb * \mathit{vlk}\ c\ vc) \mid (b, c) \leftarrow \mathit{basis}]]
\end{aligned}
$$

Given a $PDQTrans$, $getvbs$ determines how that behaves for basic vectors. Then, given the basis' elements, $firstbs$ extends the transformation. Essentially, what $first$ does is to calculate the *extended* function for the input $PDQTrans$ using $firstbs$, and then to calculate the output, correctly applying the extended $PDQTrans$ to the inputed probability distribution of state vectors. The trick for $first$ is that we have made an explicit difference between measurements and unitary transformations. If the inputed function is *not* a measurement the calculation is standard, but if that *is* a measurement then the number of states vectors in the distribution is augmented and we need to use the function $zipqd$, which combines all state vectors that are tagged with the same eigenvalue.

Proposition 8.2.1 *The given implementation for arr, $\ggg$, and $first$ satisfy the required equations for arrows.*

We can use the arrow combinators to structure quantum computations modelled by *mixed computations* over combined states.

8.3 Example: Teleportation

Now we model the algorithm for teleportation using $PDQTrans$ as arrows.

$$
\begin{aligned}
&\mathit{alice} \in \mathit{PDQTrans}\ (\mathit{Bool}, \mathit{Bool})\ ()\\
&\mathit{alice} = \mathit{proc}\ (\mathit{eprL}, q) \to \mathbf{do}\\
&\qquad\qquad (q_1, e_1) \leftarrow \mathit{controlled\_notD} \prec (q, \mathit{eprL})\\
&\qquad\qquad q_2 \leftarrow \mathit{hadamardD} \prec q_1\\
&\qquad\qquad \mathit{u1} \leftarrow \mathit{discqD} \prec q_2\\
&\qquad\qquad e_2 \leftarrow \mathit{simplqD} \prec (\mathit{u1}, e_1)\\
&\qquad\qquad u_2 \leftarrow \quad \mathit{discqD} \prec e_2\\
&\qquad\qquad \mathit{returnA} \prec u_2\\
&\mathit{bob} \in \mathit{PDQTrans}\ \mathit{Bool}\ \mathit{Bool}\\
&\mathit{bob} = \mathit{PDQTrans}\ (\lambda x \to \mathit{PDQ}\ [((l2_1, v_3), p_1) \mid ((l2_1, v_1), p_1) \leftarrow \mathit{unPDQ}\ x,\\
&\qquad \mathbf{let}\ v_2 = \mathbf{if}\ ((\mathit{head}\ l2_1) \equiv 1)\ \mathbf{then}\ v_1 \gg\!= \mathit{qnot}\ \mathbf{else}\ v_1,\\
&\qquad \mathbf{let}\ v_3 = \mathbf{if}\ ((\mathit{head}\ (\mathit{tail}\ l2_1)) \equiv 1)\ \mathbf{then}\ v_2 \gg\!= z\ \mathbf{else}\ v_2])\\
&\mathit{teleportation} \in \mathit{PDQTrans}\ (\mathit{Bool}, \mathit{Bool}, \mathit{Bool})\ \mathit{Bool}\\
&\mathit{teleportation} = \mathit{proc}\ (\mathit{eprL}, \mathit{eprR}, q) \to \mathbf{do}\\
&\qquad\qquad\qquad\qquad \mathit{u1} \leftarrow \mathit{alice} \prec (\mathit{eprL}, q)\\
&\qquad\qquad\qquad\qquad q' \leftarrow \mathit{bob} \prec \mathit{eprR}\\
&\qquad\qquad\qquad\qquad \mathit{returnA} \leftarrow q'
\end{aligned}
$$

A running of $teleportation$ of the qubit $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ produces the following output

$$
\begin{aligned}
&([0.0, 0.0], [(\mathit{False}, 1\ /\ \sqrt{2}), (\mathit{True}, 1\ /\ \sqrt{2})]) \to 0.25\\
&([1.0, 0.0], [(\mathit{False}, 1\ /\ \sqrt{2}), (\mathit{True}, 1\ /\ \sqrt{2})]) \to 0.25\\
&([0.0, 1.0], [(\mathit{False}, 1\ /\ \sqrt{2}), (\mathit{True}, 1\ /\ \sqrt{2})]) \to 0.25\\
&([1.0, 1.0], [(\mathit{False}, 1\ /\ \sqrt{2}), (\mathit{True}, 1\ /\ \sqrt{2})]) \to 0.25
\end{aligned}
$$

that is, if we sum the probabilities the final state in Bob's site is 100% in a uniform superposition of $False$ and $True$. Note that at the end there is a list of all classical results of the measurements.

8.4 Summary

This Chapter presents an alternative model for mixed quantum computations acting on a combined state with quantum and classical data, such that the quantum part of the state is represented by an explicit probability distributions over quantum states.

Although we feel this model using explicit probability distributions of vectors is more intuitive for a programmer, it remains to be analysed how efficient, as the size of these states will in general greatly increase in the number of measurements performed and as they may represent, in different probabilities, vectors which are not observationally different.

9 CONCLUSION

In this thesis we have modelled and explained quantum programming using well established constructions of classical programming languages and semantics.

9.1 Contributions

9.1.1 High-level Languages for Quantum Computation

It took many years for a classical programming language to develop sophisticated programming abstractions, compiler technology, type systems, and connections to semantic constructions like monads. In this thesis we formalize some important connections between classical and quantum computations and we hope this would help much of the constructs and tools used for classical programming to be transfered to the domain of quantum programming languages.

In Section 4 we have developed a technique for reasoning about quantum programs, written in a pure subset of QML, using algebraic laws.

We have also noted that a general purpose quantum programming language that can faithfully express quantum algorithms would be *general*, with respect measurements, and *complete*, with respect the interchanging between quantum and classical data. Additionaly, we shown that two approaches for general and complete quantum computations can be structured using indexed arrows.

9.1.2 Main Differences between Quantum and Classical Programming

Previous work on quantum programming seems to declare that this new approach is completely disjoint with classical programming constructions. In some sense this is true for two reasons: (1) quantum computing is based on a kind of parallelism caused by the non-local character of quantum information which is qualitatively different from the classical notion of parallelism, and (2) quantum computing has a peculiar notion of observation in which the observed part of the quantum state and every other part that is entangled with it immediately lose their choerence.

Interestingly it seems that none of the other differences that are often cited between quantum and classical computing are actually relevant semantically. For example, even though we do not often think of classical computation as “reversible,” it is just as reversible as quantum computing. Both can be implemented by a set of reversible universal gates (see (NIELSEN; CHUANG, 2000), section 1.4.1), but in neither model should the user be required to reason about reversibility.

The two properties of quantum computing discussed above certainly go beyond “pure” classical programming. In this thesis we have established that quantum parallelism can

be elegantly structured using indexed monads, and that quantum measurement can be modelled using a generalisation of monads called indexed arrows. In summary, our construction relates “unusual” quantum features to well-founded semantic constructions and programming languages. We hope it will serve as a useful tool to further understand the nature and structure of quantum computation.

However as pointed in Section 6.5 it seems there is a tricky characteristic in quantum programming, which is related to forgetting variables (stopping to operate on them) in quantum programs, as the quantum state is global and possibly entangled, and forgetting a part of the quantum state corresponds to measure it and to possibly destroy entanglement.

9.1.3 High-level Executable Models of Quantum Computation

Developing executable models for quantum computation may help in a better understanding of quantum algorithms and may give inspiration for programmers to develop new quantum algorithms. Also, showing how quantum programming can be integrated to classical programming constructions, based on a sound mathematical semantics, may help the structuring of simulators for quantum computers.

9.2 Future Work

9.2.1 Quantum Haskell

As an obvious future work we plan to develop further the library for general and complete quantum computation structured as indexed arrows. Specially implementing a type system to control decoherence, adding quantum control and high-level data structures.

9.2.2 QML

We presented in Section 4.1 a functional quantum programming language, called QML developed by Altenkirch and Grattage (ALTENKIRCH; GRATTAGE, 2005). Essentially, QML is explained by translating functional programs with quantum effects into quantum circuits using additional registers for initial heap and final garbage of the computation. These circuits can be translated into superoperators, and this translation turns out to be full, *i.e.*, every superoperator is given by a computation. A QML compiler has been implemented by Grattage in Haskell (GRATTAGE; ALTENKIRCH, 2005), its output are quantum circuits which can be simulated using a standard simulator for quantum circuits. The present work is complementary: it provides a direct implementation of superoperators in Haskell, by passing the need to simulate circuits. The details of implementing QML using the library of superoperators presented here will be subject of further work.

REFERENCES

ABRAMSKY, S.; COECKE, B. A Categorical Semantics of Quantum Protocols. In: ANNUAL IEEE SYMPOSIUM ON LOGIC IN COMPUTER SCIENCE, LICS, 19., 2004, Chicago, USA. **Proceedings...** [S.l.]:IEEE Computer Society, 2004. Also arXiv:quant-ph/0402130.

ABRAMSKY, S.; DUNCAN, R. A Categorical Quantum Logic. In: INTERNATIONAL WORKSHOP ON QUANTUM PROGRAMMING LANGUAGES, 2., 2004, Turku, Finland. **Proceedings...** [S.l.: s.n.], 2004.

AHARONOV, D.; KITAEV, A.; NISAN, N. Quantum circuits with mixed states. In: ACM SYMPOSIUM ON THEORY OF COMPUTING, 1998. **Proceedings...** New York: ACM Press, 1998. p.20–30.

ALTENKIRCH, T.; GRATTAGE, J. A functional quantum programming language. In: ANNUAL IEEE SYMPOSIUM ON LOGIC IN COMPUTER SCIENCE, 20., 2005. **Proceedings...** [S.l.: s.n.], 2005.

ALTENKIRCH, T.; GRATTAGE, J. **QML**: quantum data and control. Submitted for publication, 2005.

ALTENKIRCH, T.; GRATTAGE, J.; VIZZOTTO, J. K.; SABRY, A. An Algebra of Pure Quantum Programming. In: INTERNATIONAL WORKSHOP ON QUANTUM PROGRAMMING LANGUAGES, 3., 2005. **Proceedings...** [S.l.: s.n.], 2005. To appear in ENTCS.

ALTENKIRCH, T.; REUS, B. Monadic presentations of lambda terms using generalized inductive types. In: COMPUTER SCIENCE LOGIC, 1999. **Proceedings...** [S.l.: s.n.], 1999.

ALTENKIRCH, T.; UUSTALU, T. Normalization by evaluation for $\lambda^{\rightarrow 2}$. In: FUNCTIONAL AND LOGIC PROGRAMMING, 2004. **Proceedings...** [S.l.: s.n.], 2004. p.260 – 275. (Lecture Notes in Computer Science, n. 2998).

ARRIGHI, P.; DOWEK, G. A Computational Definition of the Notion of Vectorial Space. **Electr. Notes Theor. Comput. Sci.**, [S.l.], v.117, p.249–261, 2005.

BELL, J. S. On The Einstein-Podolsky-Rosen Paradox. In: **Speakable and Unspeakable in Quantum Mechanics**. [S.l.]: Cambridge University Press, 1987. p.14–21.

BENNETT, C. H.; BRASSARD, G. Quantum Cryptography: public-key distribution and coin tossing. In: IEEE INTERNATIONAL CONFERENCE ON COMPUTER, SYSTEMS AND SIGNAL PROCESSING, 1984, Bangalore, India. **Proceedings...** [S.l.: s.n.], 1984. p.175–179.

BENNETT, C. H.; BRASSARD, G.; CREPEAU, C.; JOZSA, R.; PERES, A.; WOOTTERS, W. Teleporting an unknown quantum state via dual classical and EPR channels. **Phys Rev Lett**, [S.l.], p.1895–1899, 1993.

BETTELLI, S.; SERAFINI, L.; CALARCO, T. Toward an architecture for quantum programming. **EUR.PHYS.J.D**, [S.l.], v.25, p.181, 2003.

BONE, S.; CASTRO, M. **A Brief History of Quantum Computing**. Available at: <http://www.doc.ic.ac.uk/ nd/_nada>. Visited on January 2006.

BRIDGE, J. **Beginning Model Theory**: the completeness theorem and some consequences. [S.l.]: Oxford University Press, 1997.

BRIEGEL, H. J.; RAUSSENDORF, R. Persistent Entanglement in Arrays of Interacting Particles. **Phys. Rev. Lett.**, [S.l.], v.86, p.910–913, 2001.

CHURCH, A. An Unsolvable Problem of Elementary Number Theory. **J. Math.**, [S.l.], v.58, p.345–363, 1936.

COECKE, B. De-linearizing linearity I: projective quantum axiomatics from strong compact closure. In: INTERNATIONAL WORKSHOP ON QUANTUM PROGRAMMING LANGUAGES, 3., 2005, Chicago, USA. **Proceedings...** [S.l.]:Elsevier Science, 2005. (Electronic Notes in Theoretical Computer Science).

DANOS, V.; HONDT, E. D. .; KASHEFI, E.; PANANGADEN, P. Distributed measurement-based quantum computation. In: INTERNATIONAL WORKSHOP ON QUANTUM PROGRAMMING LANGUAGES, 3., 2005, Chicago, USA. **Proceedings...** [S.l.]:Elsevier Science, 2005. (Electronic Notes in Theoretical Computer Science).

DEUTSCH, D. Quantum Theory, the Church-Turing Principle and the Universal Quantum Computer. **Proc. Roy. Soc. London, Ser. A**, [S.l.], v.400, p.97–117, 1985.

EINSTEIN, A.; PODOLSKY, B.; ROSEN, N. Can Quantum-Mechanical Description of Physical Reality be Considered Complete? **Phys. Rev.**, [S.l.], v.47, p.777–780, 1935.

FEYNMAN, R. Simulating Physics with Computers. **International Journal of Theoretical Physics**, [S.l.], v.21, n.6&7, p.467–488, 1982.

GAY, S. Quantum Programming Languages: survey and bibliography. **Mathematical Structures in Computer Science**, [S.l.], v.16, n.4, p.581–600, 2006.

GAY, S. J.; NAGARAJAN, R. Communicating Quantum Processes. In: ACM SYMPOSIUM ON PRINCIPLES OF PROGRAMMING LANGUAGES, 32., 2005. **Proceedings...** [S.l.: s.n.], 2005.

GAY, S.; NAGARAJAN, R. Typechecking Communicating Quantum Processes. **Mathematical Structures in Computer Science**, [S.l.], v.16, n.3, p.375–406, 2006.

GRATTAGE, J.; ALTENKIRCH, T. **A compiler for a functional quantum programming language**. Submitted for publication. January, 2005.

GROVER, L. K. A fast quantum mechanical algorithm for database search. In: ANNUAL ACM SYMPOSIUM ON THEORY OF COMPUTING, 28., 1996. **Proceedings...** [S.l.: s.n.], 1996. p.212–219.

HUGHES, J. Generalising Monads to Arrows. **Science of Computer Programming**, [S.l.], v.37, p.67–111, May 2000.

KASHEFI, E.; PANANGADEN, P.; DANOS, V. **The Measurement Calculus**. Available at: <http://arxiv.org/abs/quant-ph/0412135>. Visited on January 2006.

KNILL, E. **Conventions for quantum pseudocode**. Technical Report LAUR-96-2724, Los Alamos National Laboratory. 1996.

LEUNG, D. W. Quantum computation by measurements. **J. of Quant. Comp.**, [S.l.], v.2, p.33–43, 2004.

MACLANE, S. **Categories for the Working Mathematician**. [S.l.]: Spinger Verlag, 1971.

MOGGI, E. Computational lambda-calculus and monads. In: ANNUAL SYMPOSIUM ON LOGIC IN COMPUTER SCIENCE, 4., 1989. **Proceedings...** [S.l.]:IEEE Press, 1989. p.14–23.

MOGGI, E. Notions of Computation and Monads. **Information and Computation**, [S.l.], v.93, n.1, p.55–92, 1991.

NIELSEN, M. A. Universal quantum computation using only projective measurement, quantum memory, and preparation of the 0 state. **Phys. Lett.**, [S.l.], v.A. 308, n.2–3, p.96–100, 2003.

NIELSEN, M. A.; CHUANG, I. L. **Quantum Computation and Quantum Information**. [S.l.]: Cambridge University Press, 2000.

ÖMER, B. **A Procedural Formalism for Quantum Computing**. 1998. Dissertação (Mestrado em Ciência da Computação) — Department of Theoretical Physics, Technical University of Vienna.

PATERSON, R. A New Notation for Arrows. In: INTERNATIONAL CONFERENCE ON FUNCTIONAL PROGRAMMING, 2001. **Proceedings...** [S.l.: s.n.], 2001. p.229–240.

POWER, J.; ROBINSON, E. Premonoidal Categories and Notions of Computation. **Mathematical Structures in Computer Science**, [S.l.], v.7, n.5, p.453–468, 1997.

PRESKILL, J. **Lecture notes for Physics 229, quantum computation**. Available at: <http://www.theory.caltech.edu/people/preskill/ph229/#lecture>. Visited on January 2006.

RAUSSENDORF, R.; BROWNE, D.; BRIEGEL, H. A One-Way Quantum Computer. **Phys. Rev.**, [S.l.], v.86, p.5188–5191, 2001.

RAUSSENDORF, R.; BROWNE, D.; BRIEGEL, H. Measurement-based quantum computation with cluster states. **Phys. Rev.**, [S.l.], v.A 68, 2003.

SABRY, A. A. **Rebindable syntax in GHC**. Personal Communication. 2006.

SABRY, A. Modeling quantum computing in Haskell. In: ACM SIGPLAN WORKSHOP ON HASKELL, 2003. **Proceedings...** New York: ACM Press, 2003. p.39–49.

TUAN, S. F. (Ed.). **Modern Quantum Mechanics**. [S.l.]: Addison-Wesley, 1994.

SANDERS, J. W.; ZULIANI, P. Quantum Programming. In: S.L.] MATHEMATICS OF PROGRAM CONSTRUCTION, 2000. **Proceedings...** [S.l.]:Springer-Verlag, 2000. p.80–99. (Lecture Notes in Computer Science, v.1837).

SELINGER, P. Towards a Quantum Programming Language. **Mathematical Structures in Computer Science**, [S.l.], v.14, n.4, p.527–586, 2004.

SELINGER, P.; VALIRON, B. A lambda calculus for quantum computation with classical control. **Mathematical Structures in Computer Science**, [S.l.], v.16, p.527–552, 2006.

SHANKAR, R. **Principles of Quantum Mechanics**. 2nd. [S.l.]: Springer, 1994.

SHENG LIANG, P. H.; JONES, M. Monad Transformers and Modular Interpreters. In: ACM SYMPOSIUM ON PRINCIPLES OF PROGRAMMING LANGUAGES, POPL, 22., 1995. **Proceedings...** [S.l.: s.n.], 1995.

SHOR, P. W. Algorithms for Quantum Computation: discrete logarithms and factoring. In: IEEE SYMPOSIUM ON FOUNDATIONS OF COMPUTER SCIENCE, 1994. **Proceedings...** [S.l.: s.n.], 1994. p.124–134.

SWIERSTRA, S. D.; DUPONCHEEL, L. Deterministic, Error-Correcting Combinator Parsers. In: ADVANCED FUNCTIONAL PROGRAMMING, 1996. **Proceedings...** [S.l.]:Springer-Verlag, 1996. p.184–207. (Lecture Notes in Computer Science, v.1129).

TARLECKI, A.; BURSTALL, R.; GOGUEN, J. A. Some Fundamental Algebraic Tools For The Semantics Of Computation, Part 3: indexed categories. **Theoretical Computer Science**, [S.l.], v.91, p.239–264, 1991.

TONDER, A. van. **Quantum Computation, Categorical Semantics and Linear Logic**. Available at: <http://arxiv.org/abs/quant-ph/0312174>. Visited on January 2006.

TONDER, A. van. A Lambda Calculus for Quantum Computation. **SIAM Journal on Computing**, [S.l.], v.33, n.5, p.1109–1135, 2004.

TURING, A. M. On Computable Numbers, with an Application to the Entscheidungsproblem. **Proceedings of the London Mathematical Society**, [S.l.], v.42, n.2, p.230–265, 1936.

UNRUH, D. Quantum Programs with Classical Output Streams. **Electronic Notes in Theoretical Computer Science**, [S.l.], 2005. 3rd International Workshop on Quantum Programming Languages, to be published.

VALIRON, B. Quantum typing. In: INTERNATIONAL WORKSHOP ON QUANTUM PROGRAMMING LANGUAGES, 2., 2004, Turku, Finland. **Proceedings...** [S.l.: s.n.], 2004.

VIZZOTTO, J. K.; ALTENKIRCH, T.; SABRY, A. Structuring Quantum Effects: superoperators as arrows. **Journal of Mathematical Structures in Computer Science: special issue in quantum programming languages**, [S.l.], v.16, p.453–468, 2006.

VIZZOTTO, J. K.; COSTA, A. C. R. Concurrent Quantum Programming in Haskell. In: CONGRESSO BRASILEIRO DE REDES NEURAIS, 7., SESSÃO DE COMPUTAçãO QUÂNTICA, 2005. **Anais...** [S.l.: s.n.], 2005. p.1–6.

VIZZOTTO, J. K.; COSTA, A. C. R.; SABRY, A. Quantum Arrows in Haskell. In: INTERNATIONAL WORKSHOP ON QUANTUM PROGRAMMING LANGUAGES, 4., 2006, Oxford. **Proceedings...** [S.l.: s.n.], 2006. (Electronic Notes in Theoretical Computer Science). To appear in ENTCS.

APPENDIX A LINEAR VECTOR SPACES

In this appendix, we review the essential notions on vector spaces needed to a basic understanding of the principles of quantum mechanics. This review is based on Chapter 1 from (SHANKAR, 1994).

A.1 Basics

Intuitively, a vector space is a very useful mathematical world to model scenarios from the real world - we can model the scenes and preview how they change. The cleverness of vectors spaces is that they may abstract an appropriate set of general properties of the scenarios.

Definition A.1.1 (Linear Vector Space) *A linear vector space* $\mathbb{V}$ *is a collection of objects* [1] $|1\rangle, |2\rangle, \ldots, |V\rangle, \ldots, |W\rangle, \ldots$*, called vectors, for which there exists:*

1. *A definite rule for forming the vector sum, denoted by* $|V\rangle + |W\rangle$
2. *A definite rule for multiplication by scalars* $a, b, \ldots$*, denoted by* $a|V\rangle$ *with the following features:*

- *The result of these operations is another element of the space, a feature called* closure*:* $|V\rangle + |W\rangle \in \mathbb{V}$.
- *Scalar multiplication is* distributive *in the vectors:* $a(|V\rangle + |W\rangle) = a|V\rangle + a|W\rangle$.
- *Scalar multiplication is* distributive *in the scalars:* $(a + b)|V\rangle = a|V\rangle + b|V\rangle$.
- *Scalar multiplication is* associative*:* $a(b|V\rangle) = ab|V\rangle$.
- *Addition is* commutative*:* $|V\rangle + |W\rangle = |W\rangle + |V\rangle$.
- *Addition is* associative*:* $|V\rangle(|W\rangle + |Z\rangle) = (|V\rangle + |W\rangle) + |Z\rangle$.
- *There exists a* null *vector* $|0\rangle$ *obeying* $|V\rangle + |0\rangle = |V\rangle$.
- *For every vector* $|V\rangle$ *there exists an* inverse *under addition,* $|-V\rangle$*, such that* $|V\rangle + |-V\rangle = 0$.

[1] Here we are using the *braket* Dirac's notation. The symbol $|V\rangle$ is called a *ket* and denotes a generic vector.

The numbers $a, b, \ldots$ are called the *field* over which the vector space is defined. If the fiel consists of real numbers, we have *real vector spaces*, if they are complex, we have a *complex vector space*.

By one hand, one can, of course, associate an object $|V\rangle$ with an arrow-like object such that addition of two arrows corresponds to put the tail of the second arrow on the tip of the first. Scalar multiplication corresponds to stretching the vector by a factor a. This is a real vector space since stretching by a complex number makes no sense. Since these operations acting on arrows give more arrows, we have a closure. The null vector is the arrow of zero length, while the inverse of a vector is the vector reversed in direction. Hence one can think the objects of a vector space are necessarily arrows. However, no reference has been made to magnitude or direction. The point is that while the arrows have these qualities, members of a vector space need not. For instance, consider the set of all 2×2 matrices. We know how to add them, multiply them by scalars, anf that the corresponding rules obey closure. In other words, they constitute a genuine vector space, which do not have an obvious length or direction associated with them.

Now consider linear dependence of vectors:

Definition A.1.2 (Linear Dependence) *A set of vectors is said to be linearly independent if the only linear relation such that*

$$\sum_{i=1}^{n} a_i |i\rangle = |0\rangle$$

is the one with all $a_i = 0$. *If the set of vector is not linear independent, we say they are linearly dependent.*

The definition tells that it is not possible to write any member of the linearly independent set in terms of the others. For instance, consider two non-parallel vectors $|1\rangle$ and $|2\rangle$ in a plane. These form a linearly independent set. There is no way to write one as a multiple of the other, or equivalently, no way to combine them to get the null vector. On the other hand if the vectors are parallel, we can clearly write one as multiple of the other or equivalently play them against each other to get $|0\rangle$.

Definition A.1.3 (Dimension of a Vector Space) *A vector space has dimension* n *if it can accommodate a maximum of* n *linearly independent vectors.*

For example, the plane is two-dimensional and the set of 2×2 matrices is a four-dimensional vector space, which can have the following linearly independent set of vectors:

$$|1\rangle = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \quad |2\rangle = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \quad |3\rangle = \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix} \quad |4\rangle = \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}$$

Note that it is impossible to form a linear combination of any three of them to give the fourth any three of them. So the space is at least four-dimensional, and it can not be bigger since any arbitrary 2×2 matrix can be written in terms of them:

$$\begin{bmatrix} a & b \\ c & b \end{bmatrix} = a|1\rangle + b|2\rangle + c|3\rangle + d|4\rangle.$$

If the scalars $a,\ b,\ c,\ d$ are real, we have a *real four-dimensional space*, if they are complex we have a *complex four-dimensional space*.

Definition A.1.4 (Basis) *a set of n linearly independent vectors in a n-dimensional space is called a* basis.

Thus we can write a *unique* expansion:

$$|V\rangle = \sum_{i=1}^{n} v_i |i\rangle$$

for any vector, such that the vectors $|i\rangle$ form a basis.

A.2 Inner Product Spaces

The 2×2 matrix example of a vector space in the section below has clarified that a vector space need not to have a preassigned length or direction for its elements. However, one can make up quantities that have the same properties that the lengths and angles do in the case of arrows.

Based on the definition of the *dot product* for arrows [2] there is a generalisation called the *inner product* between any two vectors $|V\rangle$ and $|W\rangle$ that is denoted by the symbol $\langle V|W\rangle$. The inner product is a number (generally complex) dependent on the two vectors and obey the following axioms:

- $\langle V|W\rangle = \langle W|V\rangle^*$ [3] (skew-symmetry)
- $\langle V|V\rangle \geqslant 0$ or 0 iff $|V\rangle = |0\rangle$ (positive semidefiniteness)
- $\langle V|(a|W\rangle + b|Z\rangle) \equiv \langle V|aW + bZ\rangle = a\langle V|W\rangle + b\langle V|Z\rangle$ (linearity in ket)

Definition A.2.1 (Inner Product Vector Space) *An* inner product space *is a vector space with an inner product.*

Notice that we have not yet presented an explicit rule for actually evaluating the inner product, we just posted that any rule must have these properties. Lets analyse the axioms closer. The first one ensures that $\langle V|V\rangle$ is real. The second axiom says that $\langle V|V\rangle$ is not just real but also positive semidefinite, vanishing only if the vector itself does. The last axiom expresses the linearity of the inner product when a linear superposition $a|W\rangle + b|Z\rangle \equiv |aW + bZ\rangle$ appears as the second vector.

Definition A.2.2 (Orthogonality) *Two vectors are* orthogonal *or perpendicular if their inner product vanishes.*

Definition A.2.3 (Norm) *The norm or length of a vector is defined as $\sqrt{\langle V|V\rangle} \equiv |V|$. A normalized vector has unit norm.*

Definition A.2.4 (Orthonormal Basis) *An orthonormal basis is a set of basis vectors all of unit norm which are pairwise orthogonal.*

---

[2] $\vec{A} \cdot \vec{B} = |A||B| \cos\theta$, which is defined in terms of the lengths of the arrows and the cosine of the angle between the arrows.

[3] Where $*$ is the conjugate complex.

We present now a concrete formula for the inner product (or also called the dot product). Given $|V\rangle$ and $|W\rangle$

$$|V\rangle = \sum_i v_i |i\rangle$$
$$|W\rangle = \sum_j v_j |j\rangle$$

Following the axioms obeyed by the inner product we have:

$$\langle V|W\rangle = \sum_i \sum_j v_i^* w_j \langle i|j\rangle.$$

Then, we have to know the inner product between basis vectors, $\langle i|j\rangle$. That depends on the details of the basis vectors and all we know for sure is that they are linearly independent. But note that if we use an orthonormal basis only diagonal terms like $\langle i|i\rangle$ will survive.

Theorem A.2.1 (Gram-Schmidt) *Given a linearly independent basis we can form linear combinations of the basis vectors to obtain an orthonormal basis.*

To verify the proof of the theorem see (NIELSEN; CHUANG, 2000), Section . Assuming that the procedure has been implemented and that the current basis is orthonormal:

$$\langle i|j\rangle = \begin{cases} 1 & \text{for } i = j \\ 0 & \text{for } i \neq j \end{cases}$$

we will use the following formula for the inner product:

$$\langle V|W\rangle = \sum_i v_i^* w_i.$$

Since the vector $|V\rangle$ is uniquely specified by its components in a given basis, we may, in this basis, write it as a column vector:

$$|V\rangle \rightarrow \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix}$$

in this basis. Likewise

$$|W\rangle \rightarrow \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{bmatrix}$$

in this basis. Hence, the inner product is given by the matrix product of the transpose conjugate of the column vector representing $|V\rangle$ with the column vector representing $|W\rangle$:

$$\langle V|W\rangle = [v_1^*, v_2^*, \ldots, v_n^*] \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{bmatrix}.$$

A.3 Dual Spaces and Dirac Notation

Column vectors are concrete manifestations of an abstract vector $|V\rangle$ or ket in a basis. We can also work backwards and go from column vectors to the abstract kets. But then it is similarly possible to work backward and associate a *row vector* with an abstract object $\langle W|$, called *bra-*W. Therefore, associated with every ket $|V\rangle$ is a column vector, and taking its *adjoint*, or transpose conjugate, we form a row vector, which is the abstract bra, $\langle V|$. Thus, there are two vector spaces, the space of kets and the dual space of bras, with a ket for every bra and vice-versa. There is a basis of vectors $|i\rangle$ for expanding kets and a similar basis $\langle i|$ for expanding bras. The basis ket $|i\rangle$ is represented in the basis we are using by a column vector with all zeros except for a 1 in the ith row, while the basis bra $\langle i|$ is a row vector with all zeros except for a 1 the the ith column.

All this may be summarized as follows:

$$|V\rangle \leftrightarrow \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix} \leftrightarrow [v_1^*, v_2^*, \ldots, v_n^*] \leftrightarrow \langle V|$$

where $\leftrightarrow$ means "within a basis".

A.4 Subspaces

Definition A.4.1 (Subspace) *Given a vector space $\mathbb{V}$, a subset of its elements that form a vector space among themselves, such that vector addition and scalar multiplication are defined in the same way in the subspace as in $\mathbb{V}$, is called a subspace.*

Remark A.4.1 *We denote a particular subspace i of dimensionality n_i by $\mathbb{V}_i^{n_i}$.*

Definition A.4.2 (Composite Spaces) *Given two subspaces $\mathbb{V}_i^{n_i}$ and $\mathbb{V}_j^{m_j}$, then $\mathbb{V}_i^{n_i} \otimes \mathbb{V}_j^{m_j}$ (read $\otimes$ as tensor) is a $n_i \times m_j$ dimensional vector space. The elements of $\mathbb{V}_i^{n_i} \otimes \mathbb{V}_j^{m_j}$ are linear combinations of tensor products $|v\rangle \otimes |w\rangle$ of elements $|v\rangle \in \mathbb{V}_i^{n_i}$ and $|w\rangle \in \mathbb{V}_j^{m_j}$. In particular, if $|i\rangle$ and $|j\rangle$ are orthonormal bases for the spaces $\mathbb{V}_i^{n_i}$ and $\mathbb{V}_j^{m_j}$ then $|i\rangle \otimes |j\rangle$ is a basis for $\mathbb{V}_i^{n_i} \otimes \mathbb{V}_j^{m_j}$.*

Remark A.4.2 *We often use abbreviated notations $|v\rangle|w\rangle$, $|v, w\rangle$ or even $|vw\rangle$ for the tensor product $|v\rangle \otimes |w\rangle$.*

A.5 Linear Operators

An operator Ω is an instruction for transforming any given vector $|V\rangle$ into another vector $|V'\rangle$. The action of the operator is represented as follows:

$$\Omega|V\rangle = |V'\rangle.$$

One says that the operator Ω has transformed the ket $|V\rangle$ into the ket $|V'\rangle$. We will restrict our attention throughout to operators Ω that do not take us out of the vector space, i.e., if $|V\rangle$ is an element of a space $\mathbb{V}$, so is $|V'\rangle = \Omega|V\rangle$.

Operators can also act on bras:

$$\langle V'|\Omega = \langle V''|$$

Linear operators are the operators which obey the following rules:

- $\Omega\alpha|V_i\rangle = \alpha\Omega|V_i\rangle$
- $\Omega\{\alpha|V_i\rangle + \beta|V_j\rangle\} = \alpha\Omega|V_i\rangle + \beta\Omega|V_i\rangle$
- $\langle V_i|\alpha\Omega = \langle V_i|\Omega\alpha$
- $(\langle V_i|\alpha + \langle V_j|\beta)\Omega = \alpha\langle V_i|\Omega + \beta\langle V_j|\Omega$.

The simplest operator is the identity operator, I, which carries the instruction:

$$I \rightarrow \text{leave the vector alone!}$$

Thus,

$$I|V\rangle = |V\rangle \text{ for all kets } |V\rangle$$

and

$$\langle V|I = \langle V| \text{ for all bras } \langle V|.$$

The nice feature of linear operators is that once their action on the basis vectors is know, their action on any vector in the space is determined. If

$$\Omega|i\rangle = |i'\rangle$$

for a basis $|1\rangle, |2\rangle, \ldots, |n\rangle$ in $\mathbb{V}^n$ (where n is the dimension of the space), then for nay $|V\rangle = \sum_i v_i|i\rangle$

$$\Omega|V\rangle = \sum_i \Omega v_i|i\rangle = \sum_i v_i\Omega|i\rangle = \sum_i v_i|i'\rangle.$$

A.5.1 Matrix Elements of Linear Operators

We are accustomed to the idea of an abstract vector being represented in a basis by an n-tuple of numbers, called its components, in terms of which all vector operations can be carried out. We shall now see that in the same manner a linear operator can be represented in a basis by a set of n^2 numbers, written as an $n \times n$ matrix, and called its *matrix elements* in that basis.

The start point is the observation made earlier, that the action of a linear operator is fully specified by its action on the basis vectors. If the basis vector suffer a change

$$\Omega|i\rangle = |i'\rangle$$

then any vector in this space undergoes a change that is readily calculable:

$$\Omega|V\rangle = \Omega\sum_i v_i|i\rangle = \sum_i v_i\Omega|i\rangle = \sum_i v_i|i'\rangle.$$

As the vector $|i'\rangle$ is know, its components in the original basis

$$\langle j|i'\rangle = \langle j|\Omega|i\rangle = \Omega_{j,i}$$

are know. The n^2 numbers, $\Omega_{i,j}$ are the *matrix* elements of Ω in this basis. If

$$\Omega|V\rangle = |V'\rangle$$

then the components of the transformed ket $|V'\rangle$ are expressible in terms of the components Ω_{ij} and $|V\rangle$:

$$\begin{aligned} v'_i &= \langle i|V'\rangle = \langle i|\Omega|V\rangle = \langle i|\Omega\Big(\textstyle\sum_j v_j|j\rangle\Big) \\ &= \textstyle\sum_j v_j\langle i|\Omega|j\rangle \\ &= \textstyle\sum_j \Omega_{ij}v_j. \end{aligned}$$

Summarizing, we can form the following matrix for Ω:

$$\begin{bmatrix} v'_1 \\ v'_2 \\ \vdots \\ v'_n \end{bmatrix} = \begin{bmatrix} \langle 1|\Omega|1\rangle & \langle 1|\Omega|2\rangle & \dots & \langle 1|\Omega|n\rangle \\ \langle 2|\Omega|1\rangle & \langle 2|\Omega|2\rangle & \dots & \langle 2|\Omega|n\rangle \\ \vdots & & \vdots & \\ \langle n|\Omega|1\rangle & \langle n|\Omega|2\rangle & \dots & \langle n|\Omega|n\rangle \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix}$$

Notice that the components of the first column are simply the components of the first transformed basis vector $|1'\rangle = \Omega|1\rangle$ in the given basis. Likewise, the components of the jth column represent the image of the jth basis vector after Ω acts on it.

Now we can have the feeling about what an object like $|i\rangle\langle i|$ is. Whereas $\langle V|V'\rangle = (1\times n \text{ matrix})\times(n\times 1 \text{ matrix}) = (1\times 1 \text{ matrix})$ is a scalar, $|V\rangle\langle V''| == (n\times 1 \text{ matrix})\times(1\times n \text{ matrix}) = (n\times n \text{ matrix})$ is an operator. The inner product $\langle V|V'\rangle$ represents a bra and ket that have found each other, while $|V\rangle\langle V'|$, sometimes called the *outer product*, has the two factors looking the other way.

A.5.2 The Adjoint of an Operator

Recall that given a ket $\alpha|V\rangle$ the corresponding bra is

$$\langle V|\alpha^*(\text{ not } \langle V|\alpha)$$

In the same way, given a ket

$$\Omega|V\rangle$$

the corresponding bra is

$$\langle V|\Omega^\dagger$$

which *defines* the operator $\Omega^\dagger$. In other words, if Ω turns a ket $|V\rangle$ to $|V'\rangle$, then $\Omega^\dagger$ turns the bra $\langle V|$ into $\langle V'|$. Just as α and α^*, $|V\rangle$ and $\langle V|$ are related but distinct objects, so are Ω and $\Omega^\dagger$. The relation between Ω and $\Omega^\dagger$, called the *adjoint* of Ω or "omega dagger", is best seen in a basis:

$$\begin{aligned} (\Omega^\dagger)_{ij} &= \langle i|\Omega^\dagger|j\rangle = \langle \Omega i|j\rangle \\ &= \langle j|\Omega i\rangle^* = \langle j|\Omega|i\rangle^* \end{aligned}$$

so

$$\Omega^\dagger_{ij} = \Omega^*_{ji}.$$

That means that the matrix representing $\Omega^\dagger$ is the transpose conjugate of the matrix representing Ω.

A.5.3 Hermitian, Anti-Hermitian and Unitary Operators

We now turn our attention to certain special classes of operators that will play a major role in quantum mechanics.

Definition A.5.1 (Hermitian) *An operator Ω is Hermitian if $\Omega^\dagger = \Omega$.*

Definition A.5.2 (Anti-Hermitian) *An operator Ω is anti-Hermitian if $\Omega^\dagger = -\Omega$.*

The adjoint is to an operator what the complex conjugate is to numbers. Hermitian and anti-Hermitian operators are like pure real and pure imaginary numbers.

Definition A.5.3 (Unitary) *An operator U is unitary if*

$$UU^\dagger = I.$$

The equation above tells us that U and $U^\dagger$ are inverses of each other. Consequently,

$$U^\dagger U = I.$$

Theorem A.5.1 *Unitary operators preserve the inner product between the vectors they act.*

A.5.4 The Eigenvalue Problem

Consider some linear operator Ω acting on an arbitrary *nonzero* ket $|V\rangle$:

$$\Omega|V\rangle = |V'\rangle.$$

Unless the operator happens to be a trivial one, such as the identity or its multiple, the vector will suffer a nontrivial change, i.e, $|V'\rangle$ will not be simply related to $|V\rangle$. Each operator, however, has certain kets of its own, called its *eigenkets*, on which its action is simply that of rescaling:

$$\Omega|V\rangle = \omega|V\rangle.$$

In this case we say that $|V\rangle$ is an *eigenket* of Ω with *eigenvalue* ω. Given an operator Ω we can systematically determine all its eigenvalues and eigenvectors.

For instance, consider the trivial case where $\Omega = I$. Since

$$I|V\rangle = |V\rangle$$

for all $|V\rangle$, we conclude that

1. the only eigenvalue of I is 1;
2. all vectors are its eigenvectors with this eigenvalue.

The solution of the **eigenvalue problem** is given by the following calculation. The equation

$$det(\Omega - \omega I) = 0$$

which is the condition for nonzero eigenvectors, will determine the eigenvalues ω.

The eigenvalues, which are the roots of the polynomial above, are basis independent. And because every n-order polynomial has n roots, not necessarily distinct and not necessarily real, every operator in $\mathbb{V}^n$ has n eigenvalues. Once the eigenvalues are know, the eigenvectors may be found, at least for Hermitian and Unitary operators.

Theorem A.5.2 *The eigenvalues of a Hermitian operator are real.*

Theorem A.5.3 *To every Hermitian operator Ω, there exists (at least) a basis consisting of its orthonormal eigenvectors. It is diagonal in this eigenbasis and has its eigenvalues as its diagonal entries.*

APPENDIX B A HASKELL PRIMER

We use Haskell as a precise mathematical (and executable) notation.

It is useful to think of a Haskell type as representing a mathematical set. Haskell includes several built-in types that we use: the type $Boolean$ whose only two elements are $False$ and $True$; the type $Complex\ Double$ whose elements are complex numbers written $a :+ b$ where both a and b are elements of the type $Double$ which approximates the real numbers. Given two types a and b, the type (a, b) is the type of ordered pairs whose elements are of the respective types; the type $a \to b$ is the type of functions mapping elements of a to elements of b; and the type $[a]$ is the type of sequences (lists) whose elements are of type a. For convenience, we often use the keyword $\mathbf{type}$ to introduce a new type abbreviation. For example:

$\mathbf{type}\ PA = \mathbb{C}\ Double$

introduces the new type PA as an abbreviation of the more verbose $Complex\ Double$. A family of types that supports related operations can be grouped in a Haskell $\mathbf{class}$. Individual types can then be made an $\mathbf{instance}$ of the class, and arbitrary code can require that a certain type be a member of a given class.

The syntax of Haskell expressions is usually self-explanatory except perhaps for the following points. A function can be written in at least two ways. Both the following definitions define a function which squares its argument:

$sq\ n = n * n$

$sq' = \lambda n \to n * n$

A function f can be applied to every element of a list using map or using *list comprehensions*. If xs is the list $[1, 2, 3, 4]$, then both the following:

$map\ sq\ xs$

$[sq\ x \mid x \leftarrow xs]$

evaluate to $[1, 4, 9, 16]$.

Usually, a function f is applied to an argument a, by writing $f\ a$. If the function expects two arguments, it can either be applied to both at once $f\ (a, b)$ or one at a time $f\ a\ b$ depending on its type. When convenient the function symbol can be placed between the arguments using back quotes $a\ `f`\ b$.

APPENDIX C PROOFS

C.1 Proof of Proposition 6.3.1

Proof.

- First arrow equation: $arr\ \mathrm{id} \ggg f = f$.

$$\begin{aligned}
arr\ \mathrm{id} \ggg f &= \mathit{fun2lin}\ (\lambda(a_1, a_2) \to (\mathrm{id}\ a_1, \mathrm{id}\ a_2))\ `o`\ f && (by\ arr\ and\ \ggg) \\
&= \mathit{fun2lin}\ \mathrm{id}\ `o`\ f && (by\ simp.) \\
&= \mathit{return}\ `o`\ f && (by\ \mathit{fun2lin}) \\
&= \lambda a \to \mathit{return}\ a \gg\!= f && (by `o`) \\
&= \lambda a \to f\ a && (by\ m.law\ 1.) \\
&= f
\end{aligned}$$

- Second arrow equation: $f \ggg arr\ \mathrm{id} = f$.

$$\begin{aligned}
f \ggg arr\ \mathrm{id} &= f\ `o`\ \mathit{fun2lin}\ (\lambda(b_1, b_2) \to (\mathrm{id}\ b_1, \mathrm{id}\ b_2)) && (by\ arr\ and\ \ggg) \\
&= f\ `o`\ \mathit{fun2lin}\ \mathrm{id} && (by\ simp.) \\
&= f\ `o`\ \mathit{return} && (by\ \mathit{fun2lin}) \\
&= \lambda a \to f\ a \gg\!= \mathit{return} && (by\ o) \\
&= \lambda a \to f\ a && (by\ m.law\ 2.) \\
&= f
\end{aligned}$$

- Third arrow equation: $(f \ggg g) \ggg h = f \ggg (g \ggg h)$.

$$\begin{aligned}
(f \ggg g) \ggg h &= (f\ `o`\ g)\ `o`\ h && (by \ggg) \\
&= \lambda b \to (\lambda a.f\ a \gg\!= g)\ b \gg\!= h && (by\ o) \\
&= \lambda b \to (f\ b \gg\!= g) \gg\!= h && (by\ \beta)
\end{aligned}$$

$$\begin{aligned}
f \ggg (g \ggg h) &= f\ `o`\ (g\ `o`\ h) && (by \gg\!=) \\
&= \lambda a \to f\ a \gg\!= (\lambda b \to g\ b \gg\!= h) && (by\ o) \\
&= \lambda a \to (f\ a \gg\!= g) \gg\!= h && (by\ m.law\ 3.)
\end{aligned}$$

- Fourth arrow equation: $arr\ (g.f) = arr\ f \ggg arr\ g$.

$$\begin{aligned}
arr\ (g.f) &= \mathit{fun2lin}\ (\lambda(b_1, b_2) \to ((g.f)\ b_1, (g.f)\ b_2)) && (by\ arr) \\
&= \mathit{return}.(\lambda(b_1, b_2) \to ((g.f)\ b_1, (g.f)\ b_2)) && (by\ \mathit{fun2lin}) \\
&= \lambda(b_1, b_2) \to \mathit{return}\ ((g.f)\ b_1, (g.f)\ b_2) && (simp.)
\end{aligned}$$

$$\begin{aligned}
arr\ f \ggg arr\ g = \mathit{fun2lin}\ &(\lambda(b_1, b_2) \to (f\ b_1, f\ b_2))\ `o` \\
&\mathit{fun2lin}\ (\lambda(b_1, b_2) \to (g\ b_1, g\ b_2)) \\
&- (by \gg\!= and\ arr)
\end{aligned}$$

$$
\begin{aligned}
&= \mathit{return}.(\lambda(b_1, b_2) \to (f\ b_1, f\ b_2))\ `o`\\
&\quad \mathit{return}.(\lambda(b_1, b_2) \to (g\ b_1, g\ b_2))\\
&- (\mathit{by\ fun2lin})\\
&= \lambda(b_1, b_2) \to \mathit{return}\ (f\ b_1, f\ b_2) \gg\!=\\
&\quad \lambda(b_1, b_2) \to \mathit{return}\ (g\ b_1, g\ b_2))\\
&- (\mathit{by}\ o)\\
&= \lambda(b_1, b_2) \to (\lambda(b_1, b_2) \to \mathit{return}\ (g\ b_1, g\ b_2))\ (f\ b_1, f\ b_2)\\
&- (\mathit{by}\ m.law\ 1.)\\
&= \lambda(b_1, b_2) \to \mathit{return}\ ((g.f)\ b_1, (g.f)\ b_2)\\
&- (\mathit{by}\ \beta)
\end{aligned}
$$

- Fifth arrow equation: $\mathit{first}\ (\mathit{arr}\ f) = \mathit{arr}\ (f \times \mathrm{id})$.

$$
\begin{aligned}
\mathit{first}\ (\mathit{arr}\ f) &= \mathit{first}\ (\mathit{fun2lin}\ (\lambda(b_1, b_2) \to (f\ b_1, f\ b_2)))\\
&- (\mathit{by}\ \mathit{arr})\\
&= \mathit{first}\ (\mathit{return}.(\lambda(b_1, b_2) \to (f\ b_1, f\ b_2)))\\
&- (\mathit{by\ fun2lin})\\
&= \mathit{first}\ (\lambda(b_1, b_2) \to \mathit{return}\ (f\ b_1, f\ b_2))\\
&- (\mathit{by\ simp.})\\
&= \lambda((b_1, d_1), (b_2, d_2)) \to \lambda((x, y), (w, z)) \to\\
&\quad \mathit{return}\ (f\ b_1, f\ b_2)\ (x, w) * \mathit{return}\ (d_1, d_2)\ (y, z)\\
&- (\mathit{by\ first})\\
&= \lambda((b_1, d_1), (b_2, d_2)) \to \lambda((x, y), (w, z)) \to\\
&\quad \mathbf{if}\ ((f\ b_1, f\ b_2), (d_1, d_2)) \equiv ((x, w), (y, z))\\
&\quad \mathbf{then}\ 1\ \mathbf{else}\ 0\\
&- (\mathit{by\ return})
\end{aligned}
$$

$$
\begin{aligned}
\mathit{arr}\ (f \times \mathrm{id}) &= \mathit{fun2lin}\ (\lambda((b_1, d_1), (b_2, d_2)) \to ((f\ b_1, d_1), (f\ b_2, d_2)))\\
&- (\mathit{by}\ \mathit{arr})\\
&= \mathit{return}.(\lambda((b_1, d_1), (b_2, d_2)) \to ((f\ b_1, d_1), (f\ b_2, d_2)))\\
&- (\mathit{by\ fun2lin})\\
&= \lambda((b_1, d_1), (b_2, d_2)) \to \mathit{return}\ ((f\ b_1, d_1), (f\ b_2, d_2))\\
&- (\mathit{by\ return})\\
&= \lambda((b_1, d_1), (b_2, d_2)) \to \lambda((x, y), (w, z)) \to\\
&\quad \mathbf{if}\ ((f\ b_1, d_1), (f\ b_2, d_2)) \equiv ((x, y), (w, z))\\
&\quad \mathbf{then}\ 1\ \mathbf{else}\ 0\\
&- (\mathit{by\ return})
\end{aligned}
$$

- Sixth arrow equation: $\mathit{first}\ (f \ggg g) = \mathit{first}\ f \ggg \mathit{first}\ g$. In the following proofs assume: $\mathit{ad1}\ ((b_1, d_1), (b_2, d_2)) = (b_1, b_2)$ and $\mathit{ad2}\ ((b_1, d_1), (b_2, d_2)) = (d_1, d_2)$.

$$
\begin{aligned}
\mathit{first}\ (f\ `o`\ g) &= \mathit{first}\ (\lambda a.f\ a \gg\!= g)\\
&- (\mathit{by}\ o)\\
&= \lambda b \to \lambda((x, y), (w, z)).(f\ (\mathit{ad1}\ b) \gg\!= g)\ (x, w) *\\
&\quad \mathit{return}\ (\mathit{ad2}\ b)\ (y, z)\\
&- (\mathit{by\ first})\\
&= \lambda b \to \lambda((x, y), (w, z)) \to (\lambda c \to \mathit{sum}\ [(f\ (\mathit{ad1}\ b))\ a *\\
&\quad g\ a\ c \mid a \leftarrow \mathit{basis}])(x, w) * \mathit{return}\ (\mathit{ad2}\ b)\ (y, z)\\
&- (\mathit{by} \gg\!=)\\
&= \lambda b \to \lambda((x, y), (w, z)) \to \mathit{sum}\ [(f\ (\mathit{ad1}\ b))\ a *
\end{aligned}
$$

$$
\begin{aligned}
& g\ a\ (x,w) \mid a \leftarrow basis] * return\ (ad2\ b)\ (y,z) \\
& - (by\ \beta)
\end{aligned}
$$

$$
\begin{aligned}
first\ f\ `o`\ first\ g &= \lambda a \rightarrow first\ f\ a \gg\!= \lambda b \rightarrow first\ g\ b \\
&- (by `o`) \\
&= \lambda a \rightarrow \lambda((x,y),(w,z)) \rightarrow f\ (ad1\ a)\ (x,w) * \\
&\quad return\ (ad2\ a)\ (y,z) \gg\!= \lambda b \rightarrow \lambda((x,y),(w,z)) \rightarrow \\
&\quad g\ (ad1\ b)\ (x,w) * return\ (ad2\ b)\ (y,z) \\
&- (by\ first) \\
&= \lambda a \rightarrow \lambda((x,y),(w,z)) \rightarrow sum\ [f\ (ad1\ a)\ (m,o) * \\
&\quad return\ (ad2\ a)\ (n,p) * (\lambda((x,y),(w,z)) \rightarrow \\
&\quad g\ (m,o)\ (x,w) * return\ (n,p)\ (y,z))\ ((x,y),(w,z)) \mid \\
&\quad ((m,n),(o,p)) \leftarrow basis] \\
&- (by \gg\!=) \\
&= \lambda a \rightarrow \lambda((x,y),(w,z)) \rightarrow sum\ [f\ (ad1\ a)\ (m,o) * \\
&\quad return\ (ad2\ a)\ (n,p) * g\ (m,o)\ (x,w) * \\
&\quad return\ (n,p)\ (y,z) \mid ((m,n),(o,p)) \leftarrow basis] \\
&= \lambda a \rightarrow \lambda((x,y),(w,z)) \rightarrow sum\ [f\ (ad1\ a)\ a_1 * \\
&\quad g\ a_1\ (x,w) * return\ (ad2\ a)\ a_2 * \\
&\quad return\ a_2\ (y,z) \mid a_1 \leftarrow basis, a_2 \leftarrow basis] \\
&- (by\ simp.) \\
&= \lambda a \rightarrow \lambda((x,y),(w,z)) \rightarrow sum\ [f\ (ad1\ a)\ a_1 * \\
&\quad g\ a_1\ (x,w) \mid a_1 \leftarrow basis] * return\ (ad2\ a)\ (y,z) \\
&- (by\ simp.)
\end{aligned}
$$

- Seventh arrow equation: $first\ f \ggg arr\ (\mathrm{id}\ \times\ g) = arr\ (\mathrm{id}\lambda \times\ g) \ggg first\ f$.

$$
lhs = first\ f\ `o`\ arr\ (\mathrm{id}\ \times\ g)
$$

$$
\begin{aligned}
lhs &= \lambda((a_1,b_1),(a_2,b_2)) \rightarrow first\ f\ ((a_1,b_1),(a_2,b_2)) \gg\!= \\
&\quad fun2lin\ (\lambda((a,b),(c,d)) \rightarrow ((a,g\ b),(c,g\ d))) \\
&- (by\ `o`\ and\ arr) \\
&= \lambda((a_1,b_1),(a_2,b_2)) \rightarrow first\ f\ ((a_1,b_1),(a_2,b_2)) \gg\!= \\
&\quad \lambda((a,b),(c,d)) \rightarrow return\ ((a,g\ b),(c,g\ d)) \\
&- (by\ fun2lin) \\
&= \lambda((a_1,b_1),(a_2,b_2)) \rightarrow \lambda((x,y),(w,z)) \rightarrow f\ (a_1,a_2)\ (x,w) * \\
&\quad return\ (b_1,b_2)\ (y,z) \gg\!= \lambda((a,b),(c,d)) \rightarrow \\
&\quad return\ ((a,g\ b),(c,g\ d)) \\
&- (by\ first) \\
&= \lambda((a_1,b_1),(a_2,b_2)) \rightarrow \lambda c \rightarrow sum\ [f\ (a_1,a_2)\ (m,o) * \\
&\quad return\ (b_1,b_2)\ (n,p) * return\ ((m,g\ n),(o,g\ p))\ c \mid \\
&\quad ((m,n),(o,p)) \leftarrow basis] \\
&- (by \gg\!=) \\
&= \lambda((a_1,b_1),(a_2,b_2)) \rightarrow \lambda((x,y),(w,z)) \rightarrow sum\ [\\
&\quad f\ (a_1,a_2)\ (m,o) * return\ (b_1,b_2)\ (n,p) * \\
&\quad return\ ((m,g\ n),(o,g\ p))\ ((x,y),(w,z)) \mid \\
&\quad ((m,n),(o,p)) \leftarrow basis] \\
&- (by\ simp.) \\
&= \lambda((a_1,b_1),(a_2,b_2)) \rightarrow \lambda((x,y),(w,z)) \rightarrow \\
&\quad sum\ [f\ (a_1,a_2)\ (m,o) * [\mathbf{if}\ (b_1,b_2) \equiv (n,p)\ \mathbf{then}\ 1\ \mathbf{else}\ 0] *
\end{aligned}
$$

$$
\begin{array}{l}
\quad [\textbf{if } ((m, g\ n), (o, g\ p)) \equiv ((x, y), (w, z)) \\
\quad \textbf{then } 1 \textbf{ else } 0] \mid ((m, n), (o, p)) \leftarrow \mathit{basis}] \\
\quad - (\mathit{by\ return}) \\
\quad = \lambda((a_1, b_1), (a_2, b_2)) \rightarrow \lambda((x, y), (w, z)) \rightarrow \\
\quad \textbf{if } (g\ b_1, g\ b_2) \equiv (y, z) \textbf{ then } \ f\ (a_1, a_2)\ (x, w) \\
\quad \textbf{else } 0
\end{array}
$$

$$
\mathit{rhs} = \mathit{arr}\ (\mathrm{id}\ \times\ g)\ `o`\ \mathit{first}\ f
$$

$$
\begin{array}{l}
\mathit{rhs} = \lambda((a_1, b_1), (a_2, b_2)) \rightarrow \mathit{fun2lin}\ (\lambda((a, b), (c, d)) \rightarrow \\
\quad ((a, g\ b), (c, g\ d)))((a_1, b_1), (a_2, b_2)) \gg\!= \mathit{first}\ f \\
\quad - (\mathit{by}\ `o`\ \mathit{and}\ \mathit{arr}) \\
\quad = \lambda((a_1, b_1), (a_2, b_2)) \rightarrow \mathit{return}\ ((a_1, g\ b_1), (a_2, g\ b_2)) \\
\quad\ \ \gg\!= \mathit{first}\ f \\
\quad - (\mathit{by\ fun2lin}) \\
\quad = \lambda((a_1, b_1), (a_2, b_2)) \rightarrow \mathit{first}\ f\ ((a_1, g\ b_1), (a_2, g\ b_2)) \\
\quad - (\mathit{by\ monad\ law\ 1.}) \\
\quad = \lambda l\ ((a_1, b_1), (a_2, b_2)) \rightarrow \lambda((x, y), (w, z)) \rightarrow \\
\quad f\ (a_1, a_2)\ (x, w) * \mathit{return}\ (g\ b_1, g\ b_2)\ (y, z) \\
\quad - (\mathit{by\ first}) \\
\quad = \lambda((a_1, b_1), (a_2, b_2)) \rightarrow \lambda((x, y), (w, z)) \rightarrow \\
\quad f\ (a_1, a_2)\ (x, w) * [\textbf{if } (g\ b_1, g\ b_2) \equiv (y, z) \\
\quad \textbf{then } 1 \textbf{ else } 0] \\
\quad - (\mathit{by\ return})
\end{array}
$$

- Eighth arrow equation: $\mathit{first}\ f \ggg \mathit{arr}\ \mathit{fst} = \mathit{arr}\ \mathit{fst} \ggg f$.

$$
\mathit{lhs} = \mathit{first}\ f\ `o`\ \mathit{arr}\ (\lambda(a, b) \rightarrow a)
$$

$$
\begin{array}{l}
\mathit{lhs} = \lambda((a_1, b_1), (a_2, b_2)) \rightarrow \mathit{first}\ f\ ((a_1, b_1), (a_2, b_2)) \gg\!= \\
\quad \mathit{arr}\lambda(a, b) \rightarrow a \\
\quad - (\mathit{by\ o}) \\
\quad = \lambda((a_1, b_1), (a_2, b_2)) \rightarrow \mathit{first}\ f\ ((a_1, b_1), (a_2, b_2)) \gg\!= \\
\quad \lambda((a, b), (c, d)) \rightarrow \mathit{return}\ (a, c) \\
\quad - (\mathit{by\ arr}) \\
\quad = \lambda((a_1, b_1), (a_2, b_2)) \rightarrow \lambda((x, y), (w, z)) \rightarrow f\ (a_1, a_2)\ (x, w) * \\
\quad \mathit{return}\ (b_1, b_2)\ (y, z) \gg\!= \lambda((a, b), (c, d)) \rightarrow \mathit{return}\ (a, c) \\
\quad - (\mathit{by\ first}) \\
\quad = \lambda((a_1, b_1), (a_2, b_2)) \rightarrow \lambda(c_1, c_2) \rightarrow \mathit{sum}\ [f\ (a_1, a_2)\ (m, o) * \\
\quad \mathit{return}\ (b_1, b_2)\ (n, p) * \mathit{return}\ (m, o)\ (c_1, c_2) \mid \\
\quad ((m, n), (o, p)) \leftarrow \mathit{basis}] \\
\quad - (\mathit{by} \gg\!=) \\
\quad = \lambda((a_1, b_1), (a_2, b_2)) \rightarrow \lambda(c_1, c_2) \rightarrow \mathit{sum}\ [f\ (a_1, a_2)\ (m, o) * \\
\quad [\textbf{if } (b_1, b_2) \equiv (n, p) \textbf{ then } 1 \textbf{ else } 0] * \\
\quad [\textbf{if } (m, o) \equiv (c_1, c_2) \textbf{ then } 1 \textbf{ else } 0] \mid \\
\quad ((m, n), (o, p)) \leftarrow \mathit{basis}] \\
\quad - (\mathit{by\ return}) \\
\quad = \lambda((a_1, b_1), (a_2, b_2)) \rightarrow \lambda(c_1, c_2) \rightarrow f\ (a_1, a_2)\ (c_1, c_2) \\
\quad - (\mathit{by\ simp.})
\end{array}
$$

$$
\mathit{rhs} = \mathit{arr}\ \mathit{fst}\ `o`\ f
$$

$$
\begin{aligned}
rhs &= \lambda((a,b),(c,d)) \rightarrow return\ (a,c)\ `o`\ f \\
&\quad -\ (by\ arr) \\
&\quad = \lambda((a_1,b_1),(a_2,b_2)) \rightarrow (\lambda\lambda\ ((a,b),(c,d)) \rightarrow \\
&\quad\quad return\ (a,c))((a_1,b_1),(a_2,b_2)) \ggg\!= f \\
&\quad -\ (by\ o) \\
&\quad = \lambda((a_1,b_1),(a_2,b_2)) \rightarrow f\ (a_1,a_2) \\
&\quad -\ (by\ monad\ law\ 1.) \\
&\quad = \lambda((a_1,b_1),(a_2,b_2)) \rightarrow \lambda(c_1,c_2) \rightarrow f\ (a_1,a_2)\ (c_1,c_2)
\end{aligned}
$$

- Ninth arrow equation: $first\ (first\ f) \ggg arr\ assoc = arr\ assoc \ggg first\ f$

$$
\begin{aligned}
lhs &= \lambda(((a_1,b_1),c_1),((a_2,b_2),c_2)) \rightarrow first\ (first\ f) \\
&\quad\quad (((a_1,b_1),c_1),((a_2,b_2),c_2)) \ggg\!= arr\ (\lambda((a,b),c) \rightarrow (a,(b,c)))
\end{aligned}
$$

$$
\begin{aligned}
lhs &= \lambda(((a_1,b_1),c_1),((a_2,b_2),c_2)) \rightarrow first\ (\lambda b \rightarrow \\
&\quad\quad \lambda((x,y),(w,z)) \rightarrow f\ (ad1\ b)\ (x,w) * return\ (ad2\ b)\ (y,z)) \\
&\quad\quad (((a_1,b_1),c_1),((a_2,b_2),c_2)) \ggg\!= \lambda(((a_1,b_1),c_1),((a_2,b_2),c_2)) \rightarrow \\
&\quad\quad return\ ((a_1,(b_1,c_1)),(a_2,(b_2,c_2))) \\
&\quad -\ (by\ first) \\
&\quad = \lambda(((a_1,b_1),c_1),((a_2,b_2),c_2)) \rightarrow \\
&\quad\quad \lambda((m_1,n_1),p_1)\ ((m_2,n_2),p_2) \rightarrow (\lambda b \rightarrow \lambda((x,y),(w,z)) \rightarrow \\
&\quad\quad f\ (ad1\ b)\ (x,w) * return\ (ad2\ b)\ (y,z))\ ((a_1,b_1),(a_2,b_2)) \\
&\quad\quad ((m_1,n_1),(m_2,n_2)) * return\ (c_1,c_2)\ (p_1,p_2) \ggg\!= \\
&\quad\quad \lambda(((a_1,b_1),c_1),((a_2,b_2),c_2)) \rightarrow \\
&\quad\quad return\ ((a_1,(b_1,c_1)),(a_2,(b_2,c_2))) \\
&\quad -\ (by\ first) \\
&\quad = \lambda(((a_1,b_1),c_1),((a_2,b_2),c_2)) \rightarrow \lambda((m_1,n_1),p_1)\ ((m_2,n_2),p_2) \rightarrow \\
&\quad\quad f\ (a_1,a_2)\ (m_1,m_2) * return\ (b_1,b_2)\ (n_1,n_2) * \\
&\quad\quad return\ (c_1,c_2)\ (p_1,p_2) \ggg\!= \\
&\quad\quad \lambda(((a_1,b_1),c_1),((a_2,b_2),c_2)) \rightarrow return\ ((a_1,(b_1,c_1)),(a_2,(b_2,c_2))) \\
&\quad -\ (by\ \beta) \\
&\quad = \lambda(((a_1,b_1),c_1),((a_2,b_2),c_2)) \rightarrow \\
&\quad\quad \lambda((x_1,(y_1,z_1)),(x_2,(y_2,z_2))) \rightarrow \\
&\quad\quad sum\ [f\ (a_1,a_2)\ (m_1,m_2) * return\ (b_1,b_2)\ (n_1,n_2) * \\
&\quad\quad return\ (c_1,c_2)\ (p_1,p_2) * return\ ((m_1,n_1),p_1)\ ((m_2,n_2),p_2) \\
&\quad\quad ((x_1,(y_1,z_1)),(x_2,(y_2,z_2))) \mid \\
&\quad\quad ((m_1,n_1),p_1)\ ((m_2,n_2),p_2) \leftarrow basis] \\
&\quad -\ (by \ggg\!=) \\
&\quad = \lambda(((a_1,b_1),c_1),((a_2,b_2),c_2)) \rightarrow \\
&\quad\quad \lambda((x_1,(y_1,z_1)),(x_2,(y_2,z_2))) \rightarrow sum\ [f\ (a_1,a_2)\ (m_1,m_2) * \\
&\quad\quad \mathbf{if}\ (b_1,b_2) \equiv (n_1,n_2)\ \mathbf{then}\ 1\ \mathbf{else}\ 0] * \\
&\quad\quad [\mathbf{if}\ (c_1,c_2) \equiv (p_1,p_2)\ \mathbf{then}\ 1\ \mathbf{else}\ 0] * \\
&\quad\quad [\mathbf{if}\ ((m_1,n_1),p_1)\ ((m_2,n_2),p_2) \equiv ((x_1,(y_1,z_1)),(x_2,(y_2,z_2))) \\
&\quad\quad \mathbf{then}\ 1\ \mathbf{else}\ 0] \mid ((m_1,n_1),p_1)\ ((m_2,n_2),p_2) \leftarrow basis] \\
&\quad -\ (by\ return) \\
&\quad = \lambda(((a_1,b_1),c_1),((a_2,b_2),c_2)) \rightarrow \\
&\quad\quad \lambda((x_1,(y_1,z_1)),(x_2,(y_2,z_2))) \rightarrow f\ (a_1,a_2)\ (x_1,x_2) * \\
&\quad\quad return\ ((b_1,c_1),(b_2,c_2))\ ((y_1,z_1),(y_2,z_2))
\end{aligned}
$$

$$
rhs = \lambda(((a_1,b_1),c_1),((a_2,b_2),c_2)) \rightarrow
$$

$$\quad \mathit{return}\ ((a_1,(b_1,c_1)),(a_2,(b_2,c_2)))\ `o`\ \mathit{first}\ f$$

$$\begin{aligned}
\mathit{rhs} &= \lambda(((a_1,b_1),c_1),((a_2,b_2),c_2)) \rightarrow \\
&\quad \mathit{return}\ ((a_1,(b_1,c_1)),(a_2,(b_2,c_2))) \ggg\!= \mathit{first}\ f \\
&\quad -\ (\mathit{by}\ o) \\
&= \lambda(((a_1,b_1),c_1),((a_2,b_2),c_2)) \rightarrow \mathit{first}\ f\ ((a_1,(b_1,c_1)),(a_2,(b_2,c_2))) \\
&\quad -\ (\mathit{by}\ \mathit{monad}\ \mathit{law}\ 1.) \\
&= \lambda(((a_1,b_1),c_1),((a_2,b_2),c_2)) \rightarrow \lambda((x_1,(y_1,z_1)),(x_2,(y_2,z_2))) \rightarrow \\
&\quad f\ (a_1,a_2)\ (x_1,x_2) * \mathit{return}\ ((b_1,c_1),(b_2,c_2))\ ((y_1,z_1),(y_2,z_2)) \\
&\quad -\ (\mathit{by}\ \mathit{first})
\end{aligned}$$

□

APPENDIX D ESTRUTURANDO COMPUTAÇÕES QUÂNTICAS VIA SETAS

Nesta tese, discutimos que um modelo realístico para computações quânticas deve ser *geral* e *completo*, considerando medidas e a comunicação entre o mundo quântico e o mundo clássico, respectivamente. Assim sendo, explicamos e estruturamos computações quânticas gerais e completas em Haskell utilizando construções conhecidas da área de semântica e linguagens de programação clássicas, como *mônadas* e *setas*.

Nessa seção apresentamos brevemente os três principais capítulos da tese mostrando o desenvolvimento incremental que levou as principais contribuições e conclusões finais.

D.1 Introdução

A *computação quântica* (NIELSEN; CHUANG, 2000) pode ser entendida como *processamento* da informação codificada fisicamente através de um sistema *físico quântico*. A idéia básica é codificar dados binários usando bits quânticos (qubits). Diferentemente do bit clássico, o bit quântico pode estar em uma *superposição* de estados básicos, tornando possível o "paralelismo quântico". O paralelismo quântico é uma característica importante da computação quântica, pois é um dos pontos responsáveis pelo possível aumento da eficiência informação codificada fisicamente através de um sistema *físico quântico*. A idéia básica é codificar dados binários usando bits quânticos (qubits). Diferentemente do bit clássico, o bit quântico pode estar em uma *superposição* de estados básicos, tornando possível o "paralelismo quântico". O paralelismo quântico é uma característica importante da computação quântica, pois é um dos pontos responsáveis pelo possível aumento da eficiência em relação ao tempo de processamento dos algoritmos quânticos. Entretanto, dados quânticos são computacionalmente interessantes não somente pela superposição de estados. Existem outras características ímpares como a *medida* e o *emaranhamento*.

Nesta tese, discutimos que um modelo realístico para computações quânticas deve ser *geral* e *completo*, considerando medidas e a comunicação entre o mundo quântico e o mundo clássico, respectivamente. Assim sendo, explicamos e estruturamos computações quânticas gerais e completas em Haskell utilizando construções conhecidas da área de semântica e linguagens de programação clássicas, como *mônadas* (MOGGI, 1989) e *setas* (HUGHES, 2000).

Em mais detalhes, este trabalho tem como foco as seguintes contribuições: i) *entendimento de efeitos quânticos utilizando construções conhecidas na área de semântica de linguagens de programação clássicas*. O paralelismo quântico, o emaranhamento e a medida são noções que certamente vão além dos conceitos conhecidos em lingua-

gens funcionais “puras”. Com este intuito, mostramos que o paralelismo quântico pode ser modelado utilizando uma generalização de mônadas chamada *mônadas indexadas*, ou *Estruturas Kleisli*. Além disso, mostramos que a medida quântica pode ser explicada através de uma generalização mais radical de mônadas chamadas *setas*, mais especificamente *setas indexadas*, conceito este definido nesta tese. Este resultado conecta efeitos quânticos, como a superposição e a medida, à construções semânticas de linguagens de programação clássicas. ii) *Uma interpretação computacional para a mecânica quântica*. Einsten, Podolsky, e Rosen demonstraram em (BELL, 1987) algumas propriedades não-intuitivas da mecânica quântica. A idéia básica discutida pelos autores é que duas partículas emaranhadas parecem sempre comunicar alguma informação mesmo quando elas estão separadas por uma distância arbitrária. Atualmente ainda existem sérios debates na comunidade física sobre esse tópico, mas é interessante notar que, como proposto por Amr Sabry (SABRY, 2003), o emaranhamento pode essencialmente ser modelado através de atribuições à variáveis globais. Nesta tese, discutimos sobre esse assunto e modelamos o emaranhamento usando noções gerais de efeitos computacionais expressados em mônadas e setas.

D.2 Modelando Efeitos Quânticos I: Vetores de Estado como Mônadas Indexadas

O modelo tradicional de computação quântica é baseado em espaços vetoriais, com *vetores normalizados* para modelar estados computacionais e *transformações unitárias* para modelar computações quânticas fisicamente realizáveis. A idéia é que o processamento da informação é fisicamente realizado via *sistemas quânticos fechados*.

Em um sistema quântico fechado, a evolução é *reversível* (também chamada *estrita* ou *pura*), isto é, ela somente acontece por meio de portas unitárias; a medida, a qual é uma operação que modela a *interação* do sistema com o *mundo*, não é considerada. Portanto, nesse contexto, o processo computacional quântico é considerado como uma caixa preta, que lê informação de entrada e ao final do processo a saída é retornada.

Devido a natureza dos estados quânticos e operações agindo em tais estados, existem algumas diferenças intrínsecas entre programação clássica e programação quântica. Podemos enfatizar duas características principais na programação quântica: i) paralelismo quântico, o qual é caracterizado pelo fenômeno da superposição de estados quânticos e expressado pelo *vetor* de estado; ii) estado quântico *global* (possivelmente emaranhado), o qual é caracterizado pelo fato de que nem todos os vetores compostos, que modelam o estado quântico, podem ser decompostos em suas subpartes. Cada operação quântica é sempre global. Em termos abstratos isto pode ser explicado pelo fato de que a aplicação de uma operação em um *subespaço* específico do espaço vetorial em questão é realizada através da aplicação de uma operação em todo o espaço de estados. A operação identidade é aplicada nos subespaços não atingidos pela transformação. Portanto, a semântica de linguagens de programação quântica precisa necessariamente considerar este fato.

Nesta seção, apresentamos uma abordagem baseada em mônadas para programação quântica em Haskell. Para tanto, mostramos como estruturar vetores de estado quântico usando mônadas. A idéia é que a aplicação de transformações unitárias a vetores de estado é modelada pela operação monádica *bind*.

D.2.1 Mônadas Indexadas

Mônadas são utilizadas para formular definições e estruturar *noções de computações* (possivelmente não-funcionais) em linguagens de programação. Neste contexto, um *programa*, o qual apresenta noções de computações (como efeitos colaterais por exemplo), pode ser visto como uma *função de valores para computações*. Por exemplo, um programa com exceções pode ser visto como uma função que recebe um valor e retorna uma *computação* que pode suceder ou falhar.

Em Haskell, uma mônada é representada utilizando-se um tipo construtor para computações m e duas funções:

$$return \in forall\ a.a \to m\ a$$
$$\gg\!= \in forall\ a\ b.m\ a \to (a \to m\ b) \to m\ b$$

A operação $\gg\!=$ (pronunciada "bind") especifica como sequencializar computações e $return$ especifica como *elevar* valores em computações. Os requerimentos $forall$ nas definições representam que o construtor é induzido por um *endofuntor* T em alguma categoria de valores $\mathcal{C}$. Então, m é um tipo construtor agindo em *todos os objetos* da categoria de valores.

Entretanto, algumas vezes precisamos *selecionar* alguns objetos (conjuntos) da categoria $\mathcal{C}$ para aplicar o construtor T. Esta noção é um pouco mais geral que mônadas e é capturada pela definição de *estrutura Kleisli* (**?**). Basicamente, para *mônadas indexadas* (ou estrutura Kleisli), a função T não precisa ser necessariamente um endofunctor na categoria $\mathcal{C}$. Em contraste, podemos selecionar alguns objetos de $\mathcal{C}$ para aplicar o construtor. Esta idéia representa exatamente a noção que precisamos para modelar vetores de estado quântico (função que associa cada estado básico com uma determinada amplitude de probabilidade). O construtor para um vetor quântico age somente sobre os tipos que podem constituir um conjunto de bases para o espaço vetorial.

Para mônadas indexadas, as definições de $return$ e $\gg\!=$ em Haskell podem ser reescritas como:

$$return \in forall\ a.F\ a \Rightarrow a \to m\ a$$
$$\gg\!= \in forall\ a\ b.F\ a, F\ b \Rightarrow m\ a \to (a \to m\ b) \to m\ b$$

isto é, para todo a o qual $F\ a$ vale podemos aplicar o construtor m, e para todo a e b para os quais $F\ a$ e $F\ b$ valem, podemos aplicar $\gg\!=$. Além disso, para formar uma mônada indexada, as funções $return$ e $\gg\!=$ devem safisfazer as leis monádicas (MOGGI, 1989):

$$\begin{array}{rcl} m \gg\!= return & = & m \\ (return\ x) \gg\!= f & = & fx \\ (m \gg\!= f) \gg\!= g & = & m \gg\!= (\lambda x \to fx \gg\!= g) \end{array}$$

D.2.2 Vetores

Dado um conjunto a representando valores clássicos de observáveis, i.e. um *conjunto de bases*, um estado quântico puro é um vetor $a \to \mathbb{C}$, que associa cado elemento do conjunto de bases com uma amplitude de probabilidade complexa. Em Haskell, um conjunto finito a pode ser representado como uma instância da classe $Basis$, como mostrado abaixo. Essa classe tem um construtor $basis \in [a]$, o qual lista explicitamente os elementos do conjunto. Os elementos da base devem ser diferenciáveis uns dos outros, por isso temos a restrição $Eq\ a$ sobre o tipo de elementos:

$$\textbf{class}\ Eq\ a \Rightarrow Basis\ a\ \textbf{where}\ basis \in [a]$$
$$\textbf{type}\ K = \mathbb{C}\ Double$$
$$\textbf{type}\ Vec\ a = a \to K$$

O tipo K (aqui utilizamos a notação de campo básico) é o tipo de possíveis amplitudes de probabilidade.

As funções monádicas para vetores são definidas como:

$$return \in Basis\ a \Rightarrow a \rightarrow Vec\ a$$
$$return\ a\ b = \mathbf{if}\ a \equiv b\ \mathbf{then}\ 1.0\ \mathbf{else}\ 0.0$$
$$(\gg\!=) \in (Basis\ a, Basis\ b) \Rightarrow Vec\ a \rightarrow (a \rightarrow Vec\ b) \rightarrow Vec\ b$$
$$va \gg\!= f = \lambda b \rightarrow sum\ [(va\ a) * (f\ a\ b) \mid a \leftarrow basis]$$

$return$ é o construtor de vetores básicos, e $bind$, dada uma *operação unitária* (matriz) representada como uma função $a \rightarrow Vec\ b$, e dado um vetor $Vec\ a$, retorna $Vec\ b$ (i.e., ela specifica como um $Vec\ a$ pode ser transformado em um $Vec\ b$).

Proposition D.2.1 *A mônada indexada* Vec *satisfaz as equações monádicas.*

Exemplos de vetores sobre o conjunto dos booleanos podem ser definidos como segue:

$$\mathbf{instance}\ Basis\ Bool\ \mathbf{where}$$
$$basis = [False, True]$$
$$qFalse, qTrue, qFT, qFmT \in Vec\ Bool$$
$$qFalse = return\ False$$
$$qTrue = return\ True$$
$$qFT = (1\ /\ \sqrt{2})\ \$*\ (qFalse\ `mplus`\ qTrue)$$

Os primeiros dois são vetores unitarios básicos; e os dois últimos representam estados em superposição coerente de $False$ e $True$. Na notação de Dirac, esses vetores podem ser escritos, respectivamente, como $|False\rangle$, $|True\rangle$, $\frac{1}{\sqrt{2}}(|False\rangle + |True\rangle)$, e $\frac{1}{\sqrt{2}}(|False\rangle - |True\rangle)$. As operações $\$*$ e $`mplus'$ são definidas como produto escalar e soma de vetores, respectivamente.

Operações unitárias também podem ser definidas diretamente, por exemplo:

$$\mathbf{type}\ Uni\ a\ b = a \rightarrow Vec\ b$$
$$hadamard \in Uni\ Bool\ Bool$$
$$hadamard\ False = qFT$$
$$hadamard\ True = qFmT$$

D.3 Modelando Efeitos Quânticos II: Superoperadores como Setas Indexadas

Enquanto o modelo de computação quântica baseado em vetores de estado é ainda bastante considerado como um formalismo conveniente para descrever algoritmos quânticos, a utilização da medida para modelar ruído ou decoerência, e tratar computação quântica como um processo *interativo*, tem sido uma alternativa bastante interessante (AHARONOV; KITAEV; NISAN, 1998; RAUSSENDORF; BROWNE; BRIEGEL, 2003; DANOS et al., 2005).

Nesta seção, revisamos o modelo para computações quânticas gerais, incluindo a operação de medida, baseado em matrizes de densidade e superoperadores. Depois de expressar tal modelo em Haskell, mostramos que os superoperadores, utilizados para expressar todas as computações e medidas, são uma instância do conceito de *setas indexadas*, uma generalização de mônadas. O material apresentado nessa seção foi publicado em (VIZZOTTO; ALTENKIRCH; SABRY, 2006).

D.3.1 Setas Indexadas

Para tratar situações onde as mônadas são inaplicáveis, Hughes (HUGHES, 2000) introduziu uma nova abstração generalizando mônadas, chamada *setas*. Realmente, em adição a definição da noção de procedimento que pode realizar efeitos computacionais, setas podem ter um componente estático, ou aceitar mais que uma entrada.

Da mesma maneira como definimos um tipo monádico $m\ a$ representando uma *computação* retornando um valor a, podemos pensar em uma seta do tipo $a\ b\ c$ representando uma computação com entrada do tipo b retornando um c. Setas tornam a dependência na entrada explícita:

$$\begin{aligned}
&arr \quad \in forall\ b\ c.(b \to c) \to a\ b\ c\\
&(\ggg) \in forall\ b\ c\ d.a\ b\ c \to a\ c\ d \to a\ b\ d\\
&first\ \in forall\ b\ c\ d.a\ b\ c \to a\ (b,d)\ (c,d)
\end{aligned}$$

Em outras palavras, para ser uma seta, um tipo a deve suportar as três operações arr, $\ggg$, e *first* dos tipos como declarados acima. A função *arr* possibilita *elevarmos* funções "puras" em computações. A função $\ggg$ compõe duas computações. A função $first$ possibilita a aplicação de uma seta no contexto de outros dados.

Observe os requerimentos de $forall$ nas definições. Eles significam que podemos construir computações sobre *todas* as funções agindo sobre valores. Entretanto, como no caso das mônadas, precisamos selecionar alguns funções puras específicas. Este é exatamente o caso para computações quânticas: precisamos elevar funções simples agindo sobre conjunto de bases em funções agindo em vetores sobre essas bases. Consequentemente, definimos *setas indexadas*:

$$\begin{aligned}
&arr \in (I\ b, I\ c) \Rightarrow (b \to c) \to a\ b\ c\\
&(\ggg) \in (I\ b, I\ c, I\ d) \Rightarrow a\ b\ c \to a\ c\ d \to a\ b\ d\\
&first \in (I\ b, I\ c, I\ d) \Rightarrow a\ b\ c \to a\ (b,d)\ (c,d)
\end{aligned}$$

As operações para setas ou *setas indexadas* devem satisfazer as leis das setas (HUGHES, 2000), tal que essas operações são bem definidas sobre permutações arbitrárias e trocas associativas.

D.3.2 Superoperadores como Setas Indexadas

Intuitivamente, matrizes de densidade podem ser entendidas como uma perspectiva estatística do vetor de estado. No formalismo de matrizes de densidade, um estado quântico que era modelado como um vetor v é transformado em uma matriz de tal forma que as amplitudes do vetor de estado se transformam em um tipo de distribuição de probabilidade sobre vetores de estado.

$$\mathbf{type}\ Dens\ b = Vec\ (b,b)$$

Mapeamentos entre matrizes de densidade são chamados de *superoperadores*:

$$\mathbf{type}\ Super\ b\ c = (b,b) \to Dens\ c$$

A idéia é representar superoperadores como uma *matriz grande*, mapeando valores à matrizes de densidade (i.e., $Super\ b\ c \equiv (b,b) \to (c,c) \to K$).

Da mesma forma como o efeito da amplitude de probabilidade associado com vetores é modelados por uma *mônada indexadas* devido á restrição do $Basis$, o tipo $Super$ é modelado por uma *seta indexada*. As definições de arr, $\ggg$, e $first$ para $Super$ seguem abaixo:

$$\begin{aligned}
&arr \in (Basis\ b, Basis\ c) \Rightarrow (b \to c) \to Super\ b\ c\\
&arr\ f = fup\ (\lambda(b_1,b_2) \to return\ (f\ b_1, f\ b_2))\\
&(\ggg) \in (Basis\ b, Basis\ c, Basis\ d) \Rightarrow Super\ b\ c \to Super\ c\ d \to Super\ b\ d
\end{aligned}$$

$$(f \ggg g)\ (b_1, b_2) = (f\ (b_1, b_2) \gg\!= g)$$
$$\mathit{first} \in (\mathit{Basis}\ b, \mathit{Basis}\ c, \mathit{Basis}\ d) \Rightarrow \mathit{Super}\ b\ c \rightarrow \mathit{Super}\ (b, d)\ (c, d)$$
$$\mathit{first}\ f\ ((b_1, d_1), (b_2, d_2)) = \mathit{permute}\ ((f\ (b_1, b_2)) \langle * \rangle (\mathit{return}\ (d_1, d_2)))$$
$$\textbf{where}\ \mathit{permute}\ v\ ((b_1, b_2), (d_1, d_2)) = v\ ((b_1, d_1), (b_2, d_2))$$

A função *arr* constrói um superoperados dada uma função pura, aplicando a função à ambos vetor e seu dual. A composição de setas simplesmente aplica dois superoperadores em sequência. A função *first* aplica o superoperador f ao primeiro componente e deixa o segundo componente como estava. A definição calcula cada parte separadamente e então permuta os resultados para formar o tipo requerido.

Proposition D.3.1 *A seta indexada $Super$ satisfaz as equações requeridas para setas.*

Usando este modelo *geral* para computações quânticas estruturado como setas, podemos expressar de maneira elegante computações quânticas envolvendo medidas. Entretanto, este trabalho é somente baseado em dados quânticos. Ainda não conseguimos expressar algoritmos com interações combinadas de operações quânticas e clássicas. Como já notado por (GAY; NAGARAJAN, 2005; UNRUH, 2005) um modelo *completo* para expressar algoritmos quânticos deve acomodar ambos a medidas e interações combinadas de dados clássicos e quânticos.

D.4 Modelando Efeitos Quânticos III: Programas Mistos como Setas Indexadas

O modelo apresentado na seção acima é puramente quântico. Entretanto, diversos algoritmos quânticos são explicados em termos da sua *interação* entre informação clássica e quântica (por exemplo, uma medida no meio da computação). Um exemplo de algoritmo que apresenta tal interação é a teleportação quântica. Esse algoritmo apresenta dois processos quânticos se comunicando via *dado clássico*. Existe interesse na consideração de um modelo *misto* para computações quânticas envolvebdo *medidas* e o *fluxo de informação* entre o processos clássicos e quânticos (veja (RAUSSENDORF; BROWNE; BRIEGEL, 2003; GAY; NAGARAJAN, 2005; UNRUH, 2005)).

Portanto, gostaríamos de um *framework* capaz de representar ambos: (1)o *estado quântico* resultante de uma operação unitária aplicada em um dados estado quântico, e (2) o par de informação retornado por uma medida, isto é: (2a) correspondendo ao *valor de medida* produzido pela operação de medida (um auto-valor do observável), e (2b) o *estado quântico* que resulta da projeção imposta no estado quântico original pela medida.

O principal obstáculo introduzido pela necessidade desta uniformidade é que os resultados da medida (ambos valor e estado) são do tipo probabilístico, necessitando *conjuntos de possíveis resultados* para sua representação. A alternativa usual é o formalismo de matrizes de densidade.

Consequentemente, nesta seção apresentamos um modelo para computação *mistas* ou *combinadas* baseado em uma abordagem de medida sobre matrizes de densidade. Chamamos de computações mistas ou combinadas qualquer computação transformando um estado combinado, com dados clássicos e quânticos. Essencialmente, a idéia é ter uma matriz de densidade representando a parte do estado quântico (global) e uma distribuição de probabilidade de valores clássicos representando a parte clássica do estado. Um programa quântico agindo neste estado combinado é interpretado por um *superoperador de traço*, o qual projeto parte do estado quântico, retornando uma saída clássica, e deixando

o sistema em um novo estado (possivelmente em um espaço com dimensões reduzidas). O material apresentado nesta seção foi publicado em (VIZZOTTO; COSTA; SABRY, 2006).

D.4.1 Programas com Matrizes de Densidade

Pelo motivo que o superoperador de traço em geral *esquece* parte do estado, definimos uma relação entre as bases a qual chamamos de Dec (de *decomposição*):

$$\textbf{class}\ (Basis\ a, Basis\ b, Basis\ o) \Rightarrow Dec\ a\ b\ o\ \textbf{where}$$
$$dec \in [a] \rightarrow [(b, o)]$$

especificando que um conjunto básico a escrito pode ser escrito como (b, o). Então, um programa quântico de a para b,parametrizado por i, o tipo da distribuição de probabilidade clássica no sistema antes da operação, e o, a parte a ser medida, é representado por um superoperador de a para b, retornando uma distribuição de probabilidade clássica sobre o.

$$\textbf{type}\ DProb\ c = [(c, Prob)]$$
$$\textbf{type}\ QProgram\ i\ o\ a\ b = (DProb\ i, (a, a)) \rightarrow (DProb\ o, Dens\ b)$$

Note que os programs devem satisfazer a restrição $Dec\ a\ b\ o$, e que $DProb\ i$ é utilizado na operação clássica ou operações quânticas controladas por dados clássicos.

Qualquer operador unitário pode agora se definido como um programa agindo no estado misto que *esquece* $()$.

$$uni2qprog \in (Basis\ a, Basis\ b, Basis\ i, Dec\ a\ b\ ()) \Rightarrow$$
$$Lin\ a\ b \rightarrow QProgram\ i\ ()\ a\ b$$

A idéia é aplicar o método padrão para construir um superoperador a partir de uma operação unitária. Note que a entrada clássica é ignorada e a saída clássica é vazia: não existe interação com dados clássicos quando consideramos transformações unitárias. Por exemplo:

$$hadamardP \in QProgram\ i\ ()\ Bool\ Bool$$
$$hadamardP = uni2qprog\ hadamard$$

constrói um programa agindo no estado combinado a partir da transformação de $hadamard$.

Dado um estado quântico sobre o conjunto de bases (a, b), o programa quântico trR esquece o componente da direita, retornando um novo estado sobre b. O subespaço é medido antes antes de ser descartado retornando um distribuição de probabilidade sobre o conjunto básico que forma o subespaço. Neste caso, o dado de entrada clássico é ignorando.

$$trR \in (Basis\ a, Basis\ b, Dec\ (a, b)\ a\ b) \Rightarrow QProgram\ i\ b\ (a, b)\ a$$
$$trA \in (Basis\ a, Basis\ i, Dec\ a\ ()\ a) \Rightarrow QProgram\ i\ a\ a\ ()$$

Similarmente, o programa trA esquece (mede) todo o estado quântico, retornando uma distribuição de probabilidade clássica como resultado.

Assim podemos definir as três funções arr, $\ggg$, e *first*:

$$arr \in (Basis\ b, Basis\ c, Sub\ b\ ()) \Rightarrow (b \rightarrow c) \rightarrow QProgram\ i\ ()\ b\ c$$
$$arr = uni2qprog.fun2lin$$
$$(\ggg) \in (Basis\ a, Basis\ b, Basis\ c, Basis\ i, Basis\ o,$$
$$Basis\ o_2, Sub\ a\ o, Sub\ b\ o_2) \Rightarrow$$
$$QProgram\ i\ o\ a\ b \rightarrow QProgram\ o\ o_2\ b\ c \rightarrow QProgram\ i\ o_2\ a\ c$$
$$(f \ggg g)\ (dpi, (a_1, a_2)) = app\ g\ (f\ (dpi, (a_1, a_2)))$$
$$first \in (Basis\ a, Basis\ b, Basis\ c, Basis\ i, Basis\ o, Sub\ a\ o, Sub\ (a, c)\ o) \Rightarrow$$
$$QProgram\ i\ o\ a\ b \rightarrow QProgram\ i\ o\ (a, c)\ (b, c)$$

$$
\begin{array}{l}
\mathit{first}\ p\ (pi, ((a_1, c_1), (a_2, c_2))) = \\
\quad \textbf{let}\ (po, db) = p\ (pi, (a_1, a_2)) \\
\qquad vcc = \mathit{vreturn}\ (c_1, c_2) \\
\qquad dbc = [(((b_1, c_1), (b_2, c_2)), db\ (b_1, b_2) * vcc\ (c_1, c_2)) \mid \\
\qquad\qquad\qquad\qquad\qquad\qquad ((b_1, c_1), (b_2, c_2)) \leftarrow basis] \\
\quad \textbf{in}\ (po, dbc)
\end{array}
$$

A função arr constrói programas a partir de funções simples. A entrada clássica é ignorada e a saída clássica fica vazia. A função $\ggg$ compõe dois programas mistos e $first$ aumenta o espaço do estado quântico de um programa c. T A ação do novo programa (aumentado) é definida como o programa original sobre a com identidade sobre c.

Proposition D.4.1 *A seta indexada $QProgram\ i\ o$ satisfaz as equações requeridas para setas.*

D.5 Conclusão

Apresentamos um modelo geral e completo para computações agindo sobre um estado combinado (clássico e quântico). Este trabalho é um passo inicial para o desenvolvimento de uma linguagem geral trabalhando com dados clássicos e quânticos, possibilitando a interação entre os dois sistemas físicos.