A functional quantum programming language

by Jonathan James Grattage, BSc (Hons)

Thesis submitted to The University of Nottingham
for the degree of Doctor of Philosophy, September 2006

Abstract

This thesis introduces the language QML, a functional language for quantum computations on finite types. QML exhibits quantum data and control structures, and integrates reversible and irreversible quantum computations.

The design of QML is guided by the categorical semantics: QML programs are interpreted by morphisms in the category **FQC** of finite quantum computations, which provides a constructive operational semantics of irreversible quantum computations, realisable as quantum circuits. The quantum circuit model is also given a formal categorical definition via the category **FQC**.

QML integrates reversible and irreversible quantum computations in one language, using first order strict linear logic to make weakenings, which may lead to the collapse of the quantum wavefunction, explicit. Strict programs are free from measurement, and hence preserve superpositions and entanglement.

A denotational semantics of QML programs is presented, which maps QML terms into superoperators, via the operational semantics, made precise by the category **Q**. Extensional equality for QML programs is also presented, via a mapping from **FQC** morphisms into the category **Q**.

Acknowledgements

I'd like to thank my supervisors, Thorsten Altenkirch, of the School of Computer Science & IT, and Viacheslav Belavkin, of the School of Mathematical Sciences, for their support and expertise. Thorsten deserves a special mention for his knowledge and enthusiasm in helping me produce this thesis.

I'd also like to thank all the members of the Foundations of Programming Research Group for their help and support, especially: Conor McBride, Graham Hutton, Neil Ghani and Henrik Nilsson. My research colleagues Peter Morris, Catherine Hope, James Chapman, Alex Green, Mark Jago, Wouter Swierstra and Mauro Jaskelioff also deserve a special mention. Thanks also to John Cremona of the School of Mathematical Science for his useful feedback on my first and second year reports. I am also immensely appreciative of the feedback and enthusiasm of my examiners, Neil Ghani, of the University of Nottingham, and Simon Gay, of the University of Glasgow.

I would like to thank Amr Sabry of Indiana University, Bloomington, USA, and Juliana Vizzotto, of the Universidade Federal do Rio Grande do Sul, Brazil, for their collaboration with Thorsten and I in developing an equational theory for reasoning about QML programs. Their help, suggestions, and enthusiasm for the QML project have been indispensable. I am also indebted to Amr Sabry and his family for their help and hospitality after I became stranded in Indiana for three days.

Thanks are also extended to Peter Selinger, of Dalhousie University, Canada, for his extensive and detailed feedback on my papers and talks, and for his organisation of the Workshop on Quantum Programming Languages. His enthusiasm for the field of quantum programming has been inspiring.

I'd also like to acknowledge the help and support of my friends from computer science, physics, and the real world. I'd like to thank David Camplin especially for his support throughout my academic career.

Thanks are also due to the EPSRC and the LMS, who jointly funded this research via the MathFIT initiative, to the QNET Semantics of Quantum Computation Network, and also to the many other bodies that have funded me in various ways.

Lastly, I'd like to thank my family, whose support and encouragement has been

invaluable: my wife Janine, my parents Ruth and Trevor, my siblings Rebecca and Stephen, and my grandparents Henry and Ella Swinney, and also to Tess.

For my parents,
and my wife.

For now we see in a glass, darkly; but then face to face: now I know in part; but then I shall understand fully, even as I have been fully understood.

1 Corithians 13:12

Contents

List of Figures

Chapter 1

Introduction

This chapter details the principal reasons motivating much of the current research into quantum computing, and provides a short history of quantum computation. There is also a discussion of current research into the realisation of quantum computing devices and quantum programming languages. The motivation of the work described by this thesis, the development of a high-level functional quantum programming language, its operational and denotational semantics, and equational theory, are also presented.

1.1 Introduction to quantum computation

A quantum computer is one which makes advantageous use of the non-classical nature of quantum mechanics to compute algorithms in a fundamentally different way to traditional classical methods. Quantum methods give rise to unusual and non-intuitive properties of computations, such as the ability of a computation existing in a quantum superposition, *i.e.* in many states at once. The use of quantum computation does not alter the notion of what is computable; rather the methods by which the computational processes occur are fundamentally, different and rely on the quantum nature of systems to gain efficiency over their classical counterparts.

The efficiency gains made possible by quantum computation were first demonstrated on a computationally interesting problem in 1994, when Peter Shor of AT&T developed a quantum algorithm for factorisation [73]. This algorithm is exponentially faster than any currently known classical algorithm, and proved to be the catalyst

for an explosion in the interest of quantum computers. Classically, factorisation is computationally hard, with the best known classical algorithms having a complexity that is super-polynomial. Factorisation is in the Non-deterministic Polynomial (NP) complexity class, and is believed not to be in the Deterministic Polynomial (P) class, and therefore thought to be of intractable difficulty by complexity theorists. For this reason, factorisation, and related problems such as discrete logarithms and order finding problems, have been the basis of many modern cryptosystems, including the widely used RSA protocol [63]. Shor's factoring result means that by exploiting quantum mechanics, a quantum computer could decipher any secret message that was encrypted using certain popular cryptographic methods in only polynomial time, thus rendering the security of those algorithms void. This would have a huge impact on the fields of computer science and cryptography, if sufficiently powerful quantum computers are ever made available.

Two years later, in 1996, Lov Grover of Bell Labs discovered a quantum based algorithm for the fast searching of an unsorted database [32]. Grover's algorithm is only polynomially faster than classical search algorithms, but it is provably better than the best classical algorithm [58]. It does not, like Shor's algorithm, rely upon the *unproven* classical intractability of the factorisation problem. Grover's quantum database search algorithm offers a quadratic increase in speed over the best possible classical search algorithm for this problem, which is an exhaustive search. Searching forms the basis of many algorithms, such as solutions to problems in the NP complexity class, and as such Grover's algorithm could simply be 'plugged into' these existing algorithms to immediately increase their efficiency, and hence allow them to remain as feasible methods. Returning to cryptography, the searching of a Data Encryption Standard (DES) key-space to find the key used to encrypt a message is one possible application of Grover's search. Classically, the search-space is huge; to discover a 56-bit DES key would require searching 2^{55} possible keys. Applying Grover's quantum algorithm would afford a quadratic speed up, requiring only around 185 million operations. This is over one hundred million times more efficient.

Both Shor's and Grover's algorithms demonstrate the scale of the increase in efficiency offered by quantum computation, and Shor's factorisation algorithm has

been described as the quantum computer's 'killer app'. However, currently no physical quantum computers exist that can operate with an input larger than a few bits. Still, Grover's quantum database search algorithm and Shor's factorisation algorithm have both been implemented on these small scale systems, and have been found to work, but the technology as it presently exists does not scale to larger systems. Solving these fabrication problems is an active area of research in experimental physics today, and current experimental research is discussed briefly in section 1.2.

Several quantum programming languages also are currently under development. These are mostly, at present, low level languages which rely on a semi-formal style. This thesis focuses on the development of a functional language, QML (Quantum Meta Language) for quantum computations on finite types, with integrated reversible and irreversible quantum computation. A brief review of other languages and techniques is presented in section 1.4.

1.2 Experimental quantum computers

There are several research groups worldwide attempting to develop physical quantum computers, using different experimental arrangements. An active research area in physics which is used in quantum computer fabrication is that of trapped atoms. Atoms may be essentially trapped as an isolated species and then be controlled precisely. The atoms are held in ion traps consisting usually of an electromagnetic cage, and the atom is held inside by using laser cooling to reduce the atom's energy such that it cannot escape [76]. These trapped atoms could then be used as quantum bits, as described in section 3.1.1. The problem of quantum error correction in physical systems is dealt with comprehensively by Steane [75] and Cory *et al* [18]. Optically trapped atoms have been used as qubit registers, notably in the operation of a CNOT gate [50, 51]. Two qubits were stored in the internal and external degrees of freedom of an optically trapped atom. Grover's quantum search algorithm was realised in an optically trapped atom experiment [14]. A trapped atomic ion quantum bit array was used, and following a single query of the search space, the required element was found with an average probability of 60(2)%, exceeding the performance of any

possible classical search algorithm, which can only succeed with a maximum average probability of 50%.

Nuclear Magnetic Resonance, or NMR, is also a possible technique with which to control the properties of individual qubits [45]. NMR makes use of the two spin states of a spin-$\frac{1}{2}$ nucleus of an atom, which may be flipped to $\pm\frac{1}{2}$ in a magnetic field. A molecule is essentially a connected array of atoms, each with a nucleus, which may be used as an array of qubits which may interact with each other. Radio frequency fields may be used to identify and manipulate a single nucleus at a time. Chloroform [17] and cytosine molecules [38] have been used experimentally as quantum computers. A three qubit system has been developed by R. Laflamme and D. G. Cory [6, 44] based on malonic acid.

Nanoscale architectures have also been considered as a possibility for fabricating a register of quantum bits, as have quantum optical devices and many other nanoscale constructed devices. Benjamin *et al* investigated many systems including excitons and spins in quantum dots, using bulk magnets and constant Heisenberg coupling to switch the qubits on and off [9]. Biolatti *et al* have created an array of quantum dots to act as quantum data with the switching mechanism existing as interband optical transitions. They achieved subpicosecond decoherence-free switching in nanostructures which can be readily created [12]. Schenkel *et al* considered the use of the spins of electrons and nuclei in P^{31} atoms embedded in silicon as a qubit register [66]. Turchette *et al* used the birefringence of a single atom coupled to an optical resonator. The phase shifts which resulted were used in a quantum phase gate, for which they report the truth table, in reference [79]. Brune *et al* [20] demonstrated quantum teleportation between two high Q cavities containing a superposition of microwave states. Quantum teleportation has also been demonstrated by Bouwmeester *et al* [13]. The passage of information has been shown to be more efficient in quantum matter than in classical matter by Mattle *et al* [48] who coded information using two entangled particles and performing functions on one only. They transmitted ASCII characters using 5 trits (one of three messages) instead of the conventional 8 classical bits.

There is evidently much active research in various areas of physics in the realisation of quantum computing devices. Currently, the number of quantum bits achieved

experimentally is very small, and needs to increase by an order of magnitude before useful algorithms can be run.

1.3 History of quantum computation

The idea that quantum computing is possible was first put forward, independently, in 1982 by Richard Feynman [25] and Paul Benioff [7, 8]. Feynman's main motivation was the simulation of other systems which are chiefly governed by quantum mechanics. He theorised that a quantum system could more naturally and quickly simulate quantum systems, such as chemical interactions, which can be modelled considering the wavefunctions of the constituent atoms. To perform these kinds of calculations on classical computers requires an exponentially large amount of time, relative to the size of the system, but a quantum computer could efficiently simulate any quantum system. Benioff's premise centred on the continuing trend of miniaturisation in electronic circuitry. Circuits have steadily been manufactured at increasingly smaller scales until now, where they have reached the stage where quantum mechanical effects are significant in the behaviour of the devices. Benioff argued that a computer which makes use of these quantum effects may help the management of this hardware crisis, and perhaps move beyond the miniaturisation barrier.

In 1985 David Deutsch, one of the pioneers in quantum computing following Feynman, published a paper that is now considered to be a landmark in physics, although it was largely ignored at the time. In this paper Deutsch [22] hypothesised that quantum computers could perform some tasks much more efficiently than classical computers, but the examples included in the paper were of limited use and application, and were not guaranteed to work. The algorithms presented had to be rerun several times to guarantee the correct answer, wiping out any potential efficiency gains. Nevertheless, it was the ideas first presented in this article which allowed Shor to come up with the quantum factorisation algorithm [73] almost ten years later. Deutsch's idea can be explained by examining the behaviour of superpositions. It is widely accepted that matter has an associated wave–like nature, and, conversely, that waves exhibit particulate behaviour. An isolated electron in a quantum superposition can travel

along many different routes through a circuit *simultaneously* (according to quantum theory), as though it were a propagating wave. Imagine a water channel that splits into several branches; a boat could only travel down one branch at a time (classical), whereas a wave in the water can travel along all branches simultaneously (quantum). The essential idea proposed by Deutsch was that quantum computers could make use of quantum effects by having their input in a quantum superposition before any algorithm is applied. This would allow the computer to exist in multiple states simultaneously, calculating along all the different paths of the algorithm with just one run-through. This is the idea of *quantum parallelism*, and is one of the two major ideas underpinning all quantum algorithms. The concept of quantum parallelism is different to probabilistic computation, as the quantum mechanical complex-valued superpositions can interfere *destructively*, as waves can. A discussion of the differences between probabilistic and quantum computation is included in section 3.1.1.

The Shor factorisation algorithm, discussed in section 1.1, offered compelling evidence that quantum computers could exceed the capabilities of their classical counterparts by utilising quantum parallelism. This seemed to be confirmed by Grover's quantum search algorithm, which was published with a proof of its superiority over any classical algorithm [58]. Much research into quantum computation since Shor's discovery has been funded by governmental intelligence agencies, and for them the desire is certainly in developing applications for cryptology. This reasoning is justified as commercial quantum cryptographic hardware, from companies such as *MagiQ* and *id Quantique*, are now available on the open market. Additionally, there may be undiscovered applications of quantum computing, such as simulations of chemical interactions that could possibly lead to medical breakthroughs.

Shor's and Grover's algorithms provide evidence to challenge the (strong) *Church-Turing Hypothesis*, which states:

> "any algorithmic process can be simulated efficiently using a probabilistic Turing machine"

Quantum computers apparently break this assertion: They can simulate themselves (trivially) and classical computers (by the use of universal reversible circuits) efficiently, but it is not possible for classical computers to *efficiently* simulate quantum

devices. The number of bits required to store the state of a quantum computer rises exponentially with the number of quantum-bits ("qubits") used by the quantum algorithm. This highlights the difficulty with quantum computer simulations, and indeed any simulation of quantum mechanical processes; they are exponentially more complex, both in terms of space efficiency and time required for execution than classical analogues. This fact, in part, highlights the possible benefits of quantum computation, and justifies further research into the theory and practise of quantum computation.

The relatively counter–intuitive behaviour of quantum superpositions, quantum entanglement, and quantum measurement provides one motivation for this thesis. That is to provide a high level, functional, language for expressing these concepts. Even Feynman noted, quoted in [34]:

> "I think I can safely say that no one understands quantum mechanics"

1.4 Quantum programming

Quantum programming is now a firmly established field, with many introductory text books available [33, 36, 53, 57], and Preskill's online course notes [58]. However, quantum programs are usually presented in a semi-formal style and on a very low level, usually as families of quantum circuits. This is demonstrated by the presentation of the Deutsch algorithm given in section 3.9. This thesis aims to show that functional quantum programming languages can improve the presentation, further our understanding of the power of quantum computing, and lead to new applications of quantum computing – as they have done in conventional programming. Gay provides a full overview of the current state of quantum programming languages in his review paper [27].

One of the first proposals towards a quantum programming language were Knill's conventions for quantum pseudo-code [43]. Knill defines an imperative pseudocode suitable for implementation on a quantum random access machine (QRAM), which is also proposed in that research. The QRAM model is not formally defined, but it is proposed that it consists of a register machine with the ability to perform quantum

operations, including state preparation, unitary transformation and measurement, on quantum registers. It is acknowledged by Knill that quantum pseudocode as presented is not precise enough to be an implementable quantum programming language. However, it was an important step beyond the use of *ad hoc* narrative descriptions of how quantum operators and measurements should be applied, as noted by Gay [27].

More recently, Ömer implemented an imperative language QCL with quantum primitives and a syntax based on C [54]. This is considered to be the first real quantum programming language with a full definition. Sanders and Zuliani [65] proposed the language qGCL, which extends the probabilistic guarded command language by quantum primitives. A promising avenue of research is the integration of quantum programming with functional programming, [40, 52, 64]. Altenkirch, Vizotto and Sabry [84] have shown that quantum programming can be modelled using Haskell's arrow library [37], presenting a high level, but constructive, view of quantum effects.

Van Tonder has proposed a quantum λ-calculus incorporating higher order programs [80, 81]; however, measurements are not considered as part of this language. In [81] a semantics for a finitary, but higher order, calculus is suggested, based on Hilbert bundles. It is currently not clear, however, how this calculus could be realised operationally, *e.g.* using quantum circuits.

Selinger's influential paper [69] introduces a single-assignment (essentially functional) quantum programming language, which is based on the separation of *classical control* and *quantum data*. The language proposed, QPL, is a simple quantum programming language with some high-level features such as loops, recursive procedures, and structured data types. The language is statically typed, free of run-time errors, and has a clear denotational semantics presented in terms of complete partial orders of superoperators, with loops and recursion interpreted as least fixed points, following domain-theoretic semantic formalisms. The high-level structures of QPL are classical, and can be combined with operations on quantum data. Quantum data can be manipulated by using unitary operators or by measurement, which can affect the classical control flow. In more recent work, Selinger and Valiron [72] presented a functional language based on the *classical control* and *quantum data* paradigm. The language is based on call-by-value λ-calculus, and includes both classical and

quantum data, with an operational semantics allowing only classical control. The language includes a type-system based on affine intuitionistic linear logic, and they develop a type inference algorithm.

None of the approaches discussed so far introduce quantum control structures. In other words, quantum data can only be processed using combinators corresponding to quantum circuits or by measurement.

1.5 Research outline

The prevalent models of quantum computing at present are based on the idea that a quantum algorithm can be reinterpreted as a unitary transformation on a finite dimensional Hilbert space. This is a time reversible operation, which is followed by measurement modelled as a projection; a so-called designer Hamiltonian. It is a well know result that this is a sufficient model. However it remains to be shown whether always deferring measurement to the final step is an optimal method. Indeed, more recent low-level models of quantum computation make use of measurement as the tool to progress the computation at each step, using a measurement calculus [19, 61]. Separating the reversible and irreversible components of a computation in this way is unaccommodating to high level computational structures such as higher order types, continuations, recursive algorithms, and primitive and recursive data structures, as they are known from functional programming and the semantics of programming languages.

This situation can be compared with taking the "billiard ball" model, based on classical mechanics, as the primitive model of computation. Here a computation is modelled as a reversible transformation on the state space followed by a projection, which the reading of the result. The billiard ball model is discussed in section 2.1.2. This is a Turing complete model of computation, and is also adequate in terms of space complexity. However, it is not realistic to attempt to create algorithms using this model, not least due to the lack of high level computational structures.

As noted in section 1.1, physical implementations of quantum computers are currently very small and unable to process complex algorithms. This is a hindrance to

research into quantum algorithms as they cannot be developed in the same way that classical programs are, with the simple testing of new ideas on computers. Nielson and Chuang [53] observe that in order to be interesting, quantum algorithms must not only be comparable to their classical counterparts, but they must also improve upon the efficiency of the classical algorithms. Otherwise, it would be much simpler, efficient and cost effective to use the classical algorithm on a classical computer. Designing effective quantum algorithms is made harder by the fact that quantum mechanics itself is counter-intuitive in the way it behaves and the results obtained.

In light of the above discussion, it could be added to Chuang's assessment that the absence of high level computational structures compounds the problem of coming up with new and useful algorithms. The lack of high level computational structures may be an even greater hindrance to the development of quantum algorithms than the non-intuitive behaviour of quantum mechanics. Indeed, many computational paradigms such as logic programming, relational programming and categorical programming, have been mastered by many, and are quite far removed from any physical reality. These paradigms have been successful mainly due to their accommodating a plethora of high level structures. The lack of effective quantum algorithms has been referred to as "the quantum software crisis".

Selinger [69] suggests a quantum programming language which is not based on the strict sequential separation of reversible and irreversible computational steps. His approach is based on the mantra "classical control, quantum data", and demonstrates a one-assignment procedural language that includes recursion. Recursion is interpreted in the category of finite dimensional vector spaces with completely positive operators as morphisms. This category incorporates both unitary transformations and projections, and their compositions as computations. Hence, Selinger shows that a higher level structure such as recursion can be interpreted in a model of irreversible quantum computation. He further suggests that other high level data structures could be interpreted in this category.

The aim of this thesis is to use Selinger's research as a starting point to develop a new functional language for quantum algorithms that supports high level computational constructs. This language is called QML, for Quantum-ML (after the

functional language ML). The design of the language is inspired by taking a classical reversible model of computation, and exploring where a similarly designed quantum model differs. A semantics of QML is presented by interpreting terms as morphisms in the category of finite quantum computations **FQC**. The **FQC** semantics gives rise to a denotational semantics in terms of superoperators, the accepted domain of irreversible quantum computation, and at the same time to a compiler into quantum circuits, an accepted operational semantics for quantum programs.

In addition, QML features both basic quantum data structures and quantum control structures. In particular QML includes a quantum *if* construct which analyses quantum data without measuring, and hence without changing the data and therefore preserving any superposition or entanglement. QML thus differs from other work in quantum programming, as it allows both quantum data *and* quantum control.

1.6 Structure of this thesis

This chapter gives an introduction to this thesis and motivates the research presented. It includes a presentation of quantum computation and from a historical, theoretical and physical perspective. Current quantum programming languages are reviewed, and the structure of this presentation is described.

Chapter 2 presents a framework for discussing and reasoning about reversible classical computation. Physical models are discussed, and a model of computations as finite, reversible, circuits operating on classical bits is presented. An implementation of this model in Haskell is provided, and two important reversible universal gates are discussed.

Chapter 3 introduces quantum reversible circuits. The theory and linear algebra required to understand quantum computations is introduced, with a Haskell implementation, and a circuit model of reversible quantum computations is developed. This model is compared with the development in chapter 2 of a reversible model. An implementation of this model is provided, and two quantum algorithms are presented as circuits.

Chapters 4 and 5 formalise and extend the models introduced in chapters 2 and

3. Chapter 4 introduces the category **FCC**, of Finite Classical Computations. The model of reversible classical computations is made precise by defining reversible quantum circuits as the morphisms of a reversible subcategory of **FCC**, called $\mathbf{FCC}^{\simeq}$. The notion of heap and garbage is then introduced to allow irreversible computations to be defined. A denotational interpretation of **FCC** morphisms in the category of finite sets, **FinSet**, is also presented. Chapter 5 similarly gives a categorical model of reversible and irreversible quantum computations. The reversible quantum circuits introduced in chapter 3 give the morphisms in the category $\mathbf{FQC}^{\simeq}$, of reversible Finite Quantum computations. By introducing heap and garbage, in the same way as is done classically in chapter 5, a categorical model of irreversible quantum computations is developed, the category **FQC**. Strict computations, those that produce no garbage, are also introduced as morphisms of the category $\mathbf{FQC}^{\circ}$. As in the classical case, a mathematical denotation for each category is developed where **FQC** morphisms are interpreted as superoperators. This leads to a categorical formalism of the quantum circuit model, which is a contribution of this thesis.

Chapter 6 introduces the functional quantum programming language QML. The design and motivations of QML are discussed, and the syntax and typing rules presented. Several examples of QML programs are discussed, and variations of two important quantum algorithms are presented as QML programs. Discussions of quantum branching and orthogonality, and limitations on garbage when quantum control is used, are also presented. The language QML and it's syntax and semantics is the main contribution of this thesis.

In chapter 7, the operational semantics of QML is introduced, and is presented as a function that for each QML term derives an appropriate quantum circuit, via the category **FQC**, which is developed in chapter 5. A category of QML terms is also introduced, and an implementation of the operational semantics is also presented. There is also a discussion of how applying the mathematical interpretation of **FQC** objects, defined in chapter 5, gives rise to a denotational semantics of QML terms factored through the operational semantics.

Following the definition of the operational semantics of QML, chapter 8 discusses further directions the research presented in this thesis could be continued and ad-

vanced. This includes an outline of a denotational semantics for QML that is not factored through the operational semantics, which can be used to show compositionality. The denotation function of QML is defined as giving for every morphism in the category QML an equivalent morphism in the category **Q** of superoperators.

The last chapter of this thesis, chapter 9, summaries the contributions of this thesis in a conclusion.

Finally, appendix A provides an extensive discussion and analysis of Shor's algorithm for efficient factoring on a quantum computer. Shor's algorithm is important as it is one of the prime motivations for research in quantum computation, and it makes use of the quantum Fourier transform, which is used by many classes of quantum algorithms to gain an efficiency advantage over classical algorithms. An example of the QFT is implemented as a translation from the quantum circuit to a QML program.

1.7 Summary

Quantum computers are potentially very powerful tools which function in a fundamentally different way to classical computers. Research into practical implementations and the theory of quantum computing is of increasing interest, and quantum programming language theory is itself a new and rapidly developing field. The study of quantum computation is justified in its own right as a fundamentally different model of computation, and is further justified by the possibility of computing some classes of computations that are currently impractical using classical hardware.

In this thesis a functional quantum programming language, called QML, is presented, which has both quantum data and quantum control. This thesis includes the details of the development of QML by analogy to classical reversible computation, its categorical operational semantics as quantum circuits, and a denotational semantics presented as an interpretation in the category of superoperators.

Chapter 2

Reversible classical computation

The computers that we use currently are irreversible devices. They generate vast quantities of heat and noise, and little of the energy consumed is used in carrying out computations. The physical framework of computing devices is also inherently irreversible. The basic building blocks of our 'classical' computing systems, Boolean circuits, are irreversible. The universal NAND gate, for example, is obviously irreversible, having two inputs and only a single output. In addition, high-level abstract models of computation, such as λ-calculus and Cartesian closed categories, are based on irreversible processes; indeed, Cartesian products induce projections which are irreversible. This chapter serves to act as an answer to the question; are computers necessarily irreversible devices?

In closed systems, fundamental physical notions of Newtonian mechanics, Maxwellian electrodynamics, and quantum mechanics are time–reversible physical theories (with particular assignments for boundary conditions), and computing devices are physical objects. Open systems, which do allow for irreversible processes, are a derived notion; a subsystem of a larger closed system.

Is there some constraint in the physics of computation that forces irreversibility? Landauer [46] showed in 1961 that it is the act of erasing, or forgetting, information in a physical system that incurs energy loss, and it is from this 'entropy increase' that irreversibility stems. This is called Landauer's Principle, and is an answer to the entropy problem of the "Maxwell's Demon" thought experiment [42], presented in the next section.

2.0.1 Maxwell's Demon

Maxwell imagined two containers, A and B, connected by a trapdoor, which is controlled by a 'demon'. The containers are both initially filled with gas molecules at equal temperature. The demon observes the gas molecules on either side of the trapdoor, and when a molecule from container A approaches the trapdoor, the demon allows it through only if it has more than the average energy of all the molecules in container A. Conversely, only slower than average molecules are allowed to move from B to A. This process would eventually result in the average energy of the molecules in B increasing, while the average energy of the molecules in A decreases. Since the temperature of a gas is a function of molecular speed, the temperature in A will have decreased while that of B has increased. This seems to contradict the second law of thermodynamics, which states:

> "a closed system will tend towards maximum entropy"

In other words, when one part of an isolated system interacts with another part, energy tends to distribute equally among the accessible energy states of the system. As a result, the system tends to approach thermal equilibrium, at which point the entropy is at a maximum and the thermodynamic free energy is zero. However, the demon has to be considered part of the system, as it is interacting directly with the gas, and the demon has to store the information about the speed of the molecules of the gas. Landauer realised that certain measurements need not increase thermodynamic entropy as long as they are reversible. Due to the connection between thermodynamic entropy and information entropy, this also meant that the information the demon records must not be erased in order to not increase entropy. As the system is finite the demon will eventually run out of information storage space and must begin to erase the information that has been previously gathered. This erasing of the information is an irreversible process which increases the entropy of the system. Hence the entropy of the system, when the demon is included as part of the system, does not decrease, and the second law of thermodynamics is not violated.

2.0.2 Logically reversible computation

In the early 1970s, Bennett [10] discovered a way of breaking computation down into reversible steps. A excellent summary of the importance of reversible computation is given in [16], also quoted in Abramsky's work [1]:

> **Reversible computation:** Landauer [46] has demonstrated that it is only the 'logically irreversible' operations in a physical computer that necessarily dissipate energy by generating a corresponding amount of entropy for every bit of information that gets irreversibly erased; the logically reversible operations can in principle be performed dissipation-free. Currently, computations are commonly irreversible, even though the physical devices that execute them are fundamentally reversible ... At the basic level, however, matter is governed by classical mechanics and quantum mechanics, which are reversible. This contrast is only possible at the cost of efficiency loss by generating thermal entropy into the environment. With computational device technology rapidly approaching the elementary particle level it has been argued many times that this effect gains in significance to the extent that efficient operation (or operation at all) of future computers requires them to be reversible. The mismatch of computing organisation and reality will express itself in friction: computers will dissipate a lot of heat unless their mode of operation becomes reversible, possibly quantum mechanical.

2.1 Physical models of computation

Various physical models of computation have been proposed that make use of the reversible nature of physics to explain computation. First proposed by Edward Fredkin and Tommaso Toffoli, the billiard ball model of computing can be used as an idealised system for modelling high-performance computational processes, and behaves according to conservative logic [26], summarised here.

2.1.1 Conservative logic

"Conservative logic" obeys fundamental physical principles (including reversibility), and as such provides a useful theoretical framework in which to consider high performance computation, including efficiency and performance considerations. In particular, conservative logic predicts that sequential circuits may be constructed which

dissipate zero net power, in contrast with the computation devices presently available. The name "conservative logic" derives from the fact that the logic is designed to conserve the energy (and therefore mass, etc.) of the system throughout, thus obeying fundamental principles of physics.

The Church-Turing hypothesis, which states:

> "any algorithmic process can be simulated efficiently using a probabilistic Turing machine"

assumes several physical principles, namely that information cannot travel faster than a physical maximum, that the amount of information held in a system of finite size is finite also, and that it is possible to construct physical devices which act in a manner according to a set of physical laws and which can undergo AND, NOT and FAN-OUT functions. However, the AND function is inherently irreversible, and as such dissipates energy. It essentially erases some information about the system's evolution. The fact that computers behave according to physical laws, which allow for the dissipation of heat, leads to a theory of computation which mathematically can allow for a system in which no energy is lost. Though this billiard-ball model is to some extent non-physical at present, because it does not describe the systems which currently exist, it still holds merit as a theory of energy-lossless efficient computation.

The billiard ball model of computing has as its central tenet that it is ideally possible to build sequential circuits with zero internal power dissipation. This is clearly not the situation at present with regards to physical experimental evidence, due to the second law of thermodynamics, which has been written in many different forms and which is often misinterpreted. It states that "a closed system will tend towards maximum entropy."

Conservative logic takes a different standpoint, and aims to bring the logic and physics of computation together in a new theoretical framework. It is based on the unit wire and the Fredkin gate. The unit wire allows for the storage and transmission of information. The Fredkin gate, fully described later in section 2.4.2, provides a conditional routing gate. Combinations of these two operations allow for information processing in a heat-loss free manner. Essentially, it is theoretically possible to per-

form energy loss-free computation. Following the principle of loss-free computation the physical and reversible "billiard ball model" of computing can be developed.

2.1.2 Overview of the billiard ball model of computation

The billiard ball model is an implementation of conservative logic, based upon elastic collisions of billiard balls, both with each other and with reflectors. A basic framework for the model is a 2-D grid along which the balls can propagate and in which mirrors are situated. This system can be arranged to act as any logic circuit. The presence or absence of a ball at any grid point in the framework is equivalent to the binary 1 or 0 signal respectively, and the balls are indistinguishable from one another. The Fredkin gate (see section 2.4.2) can be created in the billiard ball model by using a series of five basic switch gates, shown in figure 2.1.

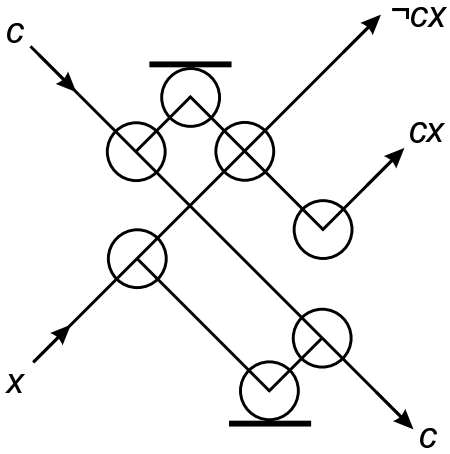


FIGURE 2.1: This 'circuit' acts as a simple switch operator. In the switch circuit, the control ball c appears to switch the path of the other ball x. In fact, when both balls enter the gate at the same time, they swap roles such that the target ball emerges along the expected path of the control ball. Note that the balls are indistinguishable, the lines denote mirrors, and the circles are snapshots of the balls at collusion instants. If there is no ball at the control input c, a ball at x will go through undeflected and emerge at $\bar{c}x$; if a ball is present at c, a ball at x would collide with it (causing the balls to exchange roles) and a ball would emerge at cx.

The billiard ball model predicts that the energy of a computation is proportional to the number of billiard balls involved, at the start and end points, and is not dependant on the processes undergone between these two states, no matter how lengthy or complicated. Thermal noise and quantisation issues must be considered as possible energy-loss mechanisms, and these are currently being debated concerning the relevance of their contribution, if any.

Other reversible computing frameworks include the kinematical model proposed by Toffoli [78], the microscopic theory of Bennett [11], the quantum mechanical theory of Benioff [7, 8], and the structural approach of Abramsky [1].

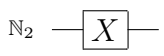
2.2 Reversible computations as circuits

In this section an inductive definition for building reversible circuits will be given. This will then be translated into a Haskell datatype, which will allow fairer comparisons between quantum computation (as circuits), which are by definition reversible, and classical computation, using the reversible model given here.

Note that the arity of a function is the number of arguments it takes. Similarly, the arity of a circuit is the number of bits, or wires, used throughout the circuit, and is therefore a natural number. By definition, a reversible computation ϕ must have an inverse, ϕ^{-1}, such that together ϕ, ϕ^{-1} give an isomorphism.

Using these definitions, the set of reversible computations (or circuits) of arity $a \in \mathbb{N}$, can be defined inductively:

Negation The only non-trivial operation possible on one (classical) bit is the negation operator, also known as the bit-flip, NOT operator, $\neg$, or X. As a circuit diagram this is denoted as:

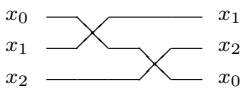


The notation X is traditionally used for negation in circuit diagrams, and $\mathbb{N}_2$ indicates that the wire carries a two-valued Boolean variable, a single classical bit. In this case the wire corresponds to one physical wire, rather than a bundle, hence the arity for this circuit is 1.

Negation is trivially reversible as applying it twice recovers the original value:

$$\neg\neg x = x$$

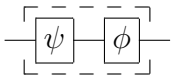
Wires Any possible reordering (permutation) is a valid reversible computation, and in a circuit is shown as a rewiring. This is represented as a bijection $\phi{:}[\,a\,] \simeq [\,a\,]$, where $[\,a\,]$ is the initial segment of a: $[\,a\,] = \{\, i \in \mathbb{N} \mid i < a \,\}$. This describes any rewiring, including the identity: $id_a = \mathit{wires}\ id$, where no permutation takes place. In a circuit diagram a rewiring is usually shown explicitly, for example:



with the bijection $\phi(0) = 2$, $\phi(1) = 0$, and $\phi(2) = 1$.

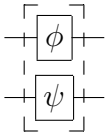
Rewiring is again trivially reversible. The arity of the rewiring circuit is the number of wires in the permutation; a in the bijection.

Sequential composition Given reversible circuits ϕ and ψ, of equal arity, a new circuit can be constructed where the output of ψ is passed as the input to ϕ. This is written $\phi \circ \psi$, and is shown diagrammatically as:



The arity of the new circuit is equal to the arity of the sub-circuits. As both ϕ and ψ are reversible, $\phi \circ \psi$ can be reversed using ψ^{-1} and ϕ^{-1} by constructing $\psi^{-1} \circ \phi^{-1}$.

Parallel composition combines any two reversible circuits in parallel, and can be thought of as the product of circuits. Given any reversible circuits ϕ and ψ, $\phi \times \psi$ can be constructed:



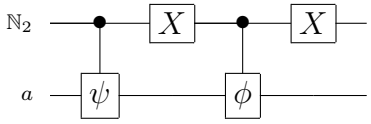
Although the operations ϕ and ψ operate on disjoint inputs, they combine in $\phi \times \psi$ to create a larger circuit which could now be applied to other circuits

of equal size. The two sub-circuits are not required to have to have the same arity; the arity of the new circuit is the sum of the sub-circuit arities.

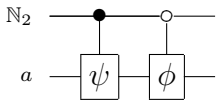
The inverse can be constructed using ψ^{-1} and ϕ^{-1}, to give $\phi^{-1} \times \psi^{-1}$.

Conditional Given reversible circuits ϕ and ψ of equal arity, a, the conditional circuit $\phi|\psi$ can be constructed. It performs a basic $if \ldots then \ldots else \ldots$ operation: *If* the control wire is True, *then* perform ψ on the remaining a wires, *else* apply ϕ. The control wire is the first, or uppermost, wire, and gives this circuit an arity of $1 + a$; the 1 representing the control wire for the circuits of arity a.

The unary conditional is commonly used, which performs the identity if the control is *false*. However, it is straightforward to reduce the binary conditional to the unary:



This representation can be further simplified by introducing a white dot on the control wire to indicate that the controlled circuit is applied only if the control wire is *false*:



If either ϕ or ψ is the identity, then the diagram can again be simplified by leaving out that branch.

The inverse is once again given using ψ^{-1} and ϕ^{-1}, to construct $\phi^{-1}|\psi^{-1}$.

2.3 Reversible circuits in Haskell

The implementation language used throughout this project is Haskell [55]. Haskell is a purely functional programming language with polymorphic types and a non-strict semantics. It is particularly suited to this research in its inclusion of pattern matching, currying, list comprehensions, guards, and definable operators. The language also supports recursive functions and algebraic data types, as well as lazy evaluation. It

has built-in support for monads and type classes. Due to the mathematical nature of the language it is also particularly well suited to the expression of mathematical and categorical concepts, which are frequently used throughout this work. The language QML, developed in this thesis, has been intentionally modelled to act as a Haskell–like language.

The reversible circuits defined in section 2.2, can be directly translated into a Haskell datatype. A function that translates each circuit into its matrix representation (a *compiler*) can then be built using this datatype. Other useful functions, such as an evaluator for matrices, can also be defined. The first step is to define a datatype that represents circuits, called *Circ*:

$$
\begin{aligned}
&\textbf{data}\ \mathit{Circ} = \mathit{Not} \\
&\qquad\qquad \mid\ \mathit{Wire}\ [\mathit{Int}] \\
&\qquad\qquad \mid\ \mathit{Par}\ \ \mathit{Circ}\ \mathit{Circ} \\
&\qquad\qquad \mid\ \mathit{Seq}\ \ \mathit{Circ}\ \mathit{Circ} \\
&\qquad\qquad \mid\ \mathit{Cond}\ \mathit{Circ}\ \mathit{Circ}
\end{aligned}
$$

In fact this datatype captures most of the information about reversible circuits given in section 2.2. A circuit can be any of the following: a *Not* operation; a *Wire*, which takes as an argument a list of integers describing the permutation (rewiring); a *Par* (parallel) operation, which takes two sub-circuits as arguments; a *Seq* (sequential) operation, which also takes two sub-circuits as arguments; and finally a *Cond* (conditional) operation, which takes the two possible circuits as arguments. Note that *Seq* simulates the circuit diagrams, so the composition $\psi \circ \phi$ becomes the sequence *Seq* ϕ ψ in Haskell, as in the diagrammatic definition of circuits given previously.

The *Circ* datatype does not capture, due to Haskell type limitations, all the constraints on what constitutes a valid circuit. For example, in the cases of *Seq* and *Cond* there is the requirement that the arity (number of wires) in each sub-circuit is the same, in order for them to be wired together correctly. In a dependently typed language, such as *Epigram* [49], this could be enforced, but in Haskell these type constraints have to be checked by an auxiliary function. This is because Haskell's type system does not allow data to appear in types; types can only be indexed by other types. The arity of a circuit is data, hence cannot appear in the type. The

$$
\begin{aligned}
&\mathit{arity} \in \mathit{Circ} && \to \mathit{Maybe}\ \mathit{Int}\\
&\mathit{arity}\quad (\mathit{Not}) && = \mathit{Just}\ 1\\
&\mathit{arity}\quad (\mathit{Wire}\ \vec{x}) && = \mathbf{do}\ \mathit{guard}\ (\mathit{chkPerm}\ \vec{x})\\
&&&\qquad \mathit{return}\ (\mathit{length}\ \vec{x})\\
&\mathit{arity}\quad (\mathit{Par}\ x\ y) && = \mathbf{do}\ m \leftarrow \mathit{arity}\ x\\
&&&\qquad n \leftarrow \mathit{arity}\ y\\
&&&\qquad \mathit{return}\ (m+n)\\
&\mathit{arity}\quad (\mathit{Seq}\ x\ y) && = \mathbf{do}\ m \leftarrow \mathit{arity}\ x\\
&&&\qquad n \leftarrow \mathit{arity}\ y\\
&&&\qquad \mathit{guard}\ (m \equiv n)\\
&&&\qquad \mathit{return}\ (m)\\
&\mathit{arity}\quad (\mathit{Cond}\ x\ y) && = \mathbf{do}\ m \leftarrow \mathit{arity}\ x\\
&&&\qquad n \leftarrow \mathit{arity}\ y\\
&&&\qquad \mathit{guard}\ (m \equiv n)\\
&&&\qquad \mathit{return}\ (1+m)\\
&\mathit{chkPerm} \in [\mathit{Int}] \to \mathbb{N}_2\\
&\mathit{chkPerm}\quad \vec{x} && = (\mathit{and}\ [\mathit{elem}\ x\ \vec{x} \mid x \leftarrow [0\,..\,\mathit{length}\ \vec{x}-1]])
\end{aligned}
$$

FIGURE 2.2: The *arity* function

function *arity*, defined in figure 2.2, makes use of the *Maybe* monad and *guards* to model and check for errors. A monad, in this context, is a way of linking together a set of functions, such that there is a predefined order of execution. The *Maybe* monad encapsulates the strategy of combining a chain of computations, that may each return *Nothing*, by ending the chain early if any step produces *Nothing* as output. In this case, the *arity* function must ensure that the input has certain properties before the result can be calculated. The *Maybe* monad simply returns either "*Just x*" or "*Nothing*," which would be the error state. The *Maybe* monad ensures any errors will be correctly propagated throughout the program.

The Haskell code for the *arity* function is given in figure 2.2. With this function, a circuit can be tested to see if it is valid by passing it as the argument. If *Just x*

is returned, where $x \in \mathbb{N}$, then the circuit is correct and its arity is x. If the error value *Nothing* is returned, then there is a circuit arity mismatch, or there is an invalid permutation. The error checking is performed by the *guard* commands in the problematic cases, which will return *Nothing* if the error–checking predicates passed to them evaluate to *False*, and will otherwise allow the computation to continue. In the case of *Wire* $\vec{x}$, the guard uses the auxiliary function *chkPerm* to ensure the permutation (rewiring) is valid. In the case of *Seq* and *Cond* it simply ensures that the size of each sub-circuit are equal. More informative error checking could be employed, but at the expense of brevity. This will be expanded on in the case of quantum circuits.

2.4 Reversible circuits: examples and theory

In this section some examples of circuits written in the *Circ* datatype are discussed, to illustrate the use of the *Circ* datatype and some principles of reversible computation.

A simple controlled-NOT(CNOT) circuit, which negates the second input if the first is *True*, and does nothing otherwise, can be defined as the circuit

$$cnot = X|\mathrm{id}_1$$

where id_a is the identity function on a bits. This circuit can be translated into the Haskell type *Circ* as:

$$cnotC \in Circ$$

$$cnotC = Cond\ Not\ (Wire\ [0])$$

A schematic circuit of CNOT, where X is the *Not* circuit and I is the identity on one bit (*wire* id_1), is shown in figure 2.3, along with the trivial simplification. Note that the terms 0 and *False*, and 1 and *True*, are used interchangeably.

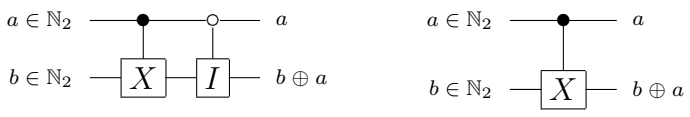


FIGURE 2.3: CNOT circuit and simplification, where $\oplus$ denotes addition *mod* 2

2.4.1 The Toffoli Gate

A useful circuit in reversible computation is the three-bit controlled-CNOT, called the Toffoli gate. This gate has three inputs, the first two of which are controls and the third is the target. This gate acts as the identity on the target, unless both control bits are set to *True*, in which case the value of the target bit is flipped. A circuit diagram for the Toffoli gate is shown in figure 2.4. Note how the circuit appears to

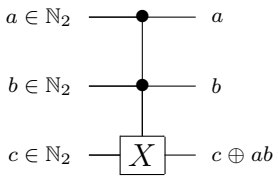


FIGURE 2.4: Schematic circuit representation of the Toffoli gate, or controlled-CNOT.

be an extension of the CNOT circuit. This fact can be used to define the Toffoli gate as

$$toffoli = cnot|\mathrm{id}_2$$

which in Haskell becomes:

$$toffoliC \in Circ$$
$$toffoliC = Cond\ cnotC\ (Wire\ [0,1])$$

The Toffoli gate is self-inverse, as applying it twice has the effect $(a, b, c) \rightarrow (a, b, c \oplus ab) \rightarrow (a, b, c)$. The Toffoli gate is also an important operation in reversible computation, as it can be used to emulate the irreversible NAND gate. Being able to use this operation allows any (deterministic) irreversible operation to be implemented reversibly, as the NAND gate (with copying, using CNOT) is *universal*; it can be used to generate any function $f \in \mathcal{C}_n \rightarrow \mathcal{C}_n$. Figure 2.5 shows the circuit for implementing the NAND gate using a Toffoli gate. This assertion holds as long as extra *heap* input is available, and a non-useful *garbage* output is allowed, both of which are required for this reversible implementation of the NAND gate.

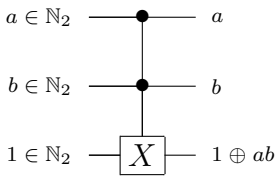


FIGURE 2.5: Toffoli gate implementation of NAND, where the third output gives the NAND of the first two; note $1 \oplus ab = \neg(a\&b)$.

2.4.2 The Fredkin Gate

Another example of a useful circuit is the three-bit controlled-swap gate, called the *Fredkin* gate. This is a universal reversible gate, assuming that input can be initialised in any way, and is used in the billiard-ball model of computation discussed in section 2.1.2. Its behaviour is such that if the control-bit (often the first wire, but sometimes the third) is set to true, then the values of the other two inputs are exchanged. In the circuit representation this can be written using a conditional that either performs a permutation, or the identity:

$$fredkin = swap|\mathrm{id}_2$$

where $swap = wires\ \phi$, and ϕ is the swap bijection. In Haskell the *fredkin* circuit can be rendered as:

$$fredkinC \in Circ$$
$$fredkinC = Cond\ (Wire\ [1,0])\ (Wire\ [0,1])$$

Schematically this is shown in figure 2.6, where a $\times$ on a wire denotes a *swap* operation. Again this operation is self-inverse, and is therefore fully reversible. Another

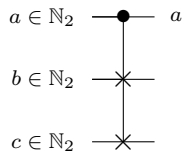


FIGURE 2.6: The Fredkin gate, which swaps b and c, but only if a is *True*

interesting property of this gate, and a reason why it is useful for the billiard-ball model of computation, is that the number of *True* and *False* values is conserved

throughout. In the billiard-ball model, where balls represent *True* values, this is equivalent to the number of balls going into the circuit being the same as the number coming out. This corresponds to the physical phenomenon of conservation of energy (and therefore mass). The two properties of reversibility and conservation make this operation interesting to physicists as both can be motivated by fundamental physical principles. The Fredkin gate can model the operation 'x AND y' by setting $a = x$, $b = y$ and $c = \mathit{False}$. The outputs are, in the same order, x, $\neg xy$ and the desired xy. NOT can also be easily implemented by setting $a = x$, $b = \mathit{False}$ and $c = \mathit{True}$, giving x, x and finally $\neg x$. Both of these operations are shown in figure 2.7. These

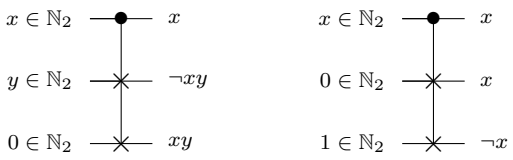


FIGURE 2.7: The Fredkin gate performing AND *(left)* and NOT *(right)*

two operations can be wired together to perform the NAND operation, and hence the Fredkin gate is also universal. Again, this is assuming the availability of extra heap input and garbage output, as required for reversibility.

2.5 Simulating a circuit

The *Circ* datatype allows reversible classical circuits to be represented in Haskell. To run circuits a simulator is needed. In this section a compiler from circuits to functions will be defined. Also described is an evaluator which will, given some input, return the output that would be produced by applying that input to the circuit. The input to a circuit is simply a Boolean valued vector with dimension equal to the arity of the circuit. A compiled circuit is a function from (Boolean valued) vectors (possible input vectors) to vectors (the output vector); in other words, a matrix. Vectors and matrices can be simply modelled by the following Haskell type-synonyms:

type $\mathit{Vec} = [\mathbb{N}_2]$

type $\mathit{Mat} = \mathit{Vec} \to \mathit{Vec}$

2.5.1 Evaluating a circuit

It would be useful to be able to apply some input to a circuit and have the associated output returned. This is achieved by an evaluation function,

$$eval \in Circ \to Vec \to Maybe\ Vec$$

This definition assuming there is a function

$$comp \in Circ \to Maybe\ Mat$$

The functions are defined in this order to simplify the definition of *comp*, as *comp* assumes that a circuit passed to it is well-formed. The *eval* function makes no such assumption, first calling *arity* on the circuit, which guarantees it is well formed, or produces the error state *Nothing*. This cannot be done so easily in the *comp* function due to its recursive nature. The *eval* function is fairly simple. It ensures the circuit is valid, via the *arity* function, then ensures the input is valid for the circuit, by checking its length against the circuit arity, and finally compiles the circuit to a matrix, via *comp*, to which the input vector is then applied. If the circuit, or input vector, are invalid, then *Nothing* is returned. This is expressed in Haskell as:

$$\begin{aligned}
&eval \in Circ \to Vec \to Maybe\ Vec \\
&eval\quad c\qquad v\quad = \mathbf{do}\ a \leftarrow arity\ c \\
&\qquad\qquad\qquad\qquad\quad guard\ (a \equiv length\ v) \\
&\qquad\qquad\qquad\qquad\quad m \leftarrow comp\ c \\
&\qquad\qquad\qquad\qquad\quad return\ (m\ v)
\end{aligned}$$

2.5.2 Compiling a circuit

In order for the *eval* function to operate, a function $comp \in Circ \to Mat$ must be defined, which transposes a circuit into its matrix representation. The function to compile a circuit is defined using pattern matching. *Not* and *Wire* are the trivial base cases, while *Seq*, *Par*, and *Cond* recursively call *comp* on each sub-circuit, before doing the necessary computations. The behaviour in each case is fully explained by the code, given in figure 2.8.

$$
\begin{aligned}
&comp \in Circ \to Maybe\ Mat \\
&comp \quad (Not) \qquad = return\ (\lambda v \to \mathbf{case}\ v\ \mathbf{of}\ [True] \to [False] \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad [False] \to [True]) \\
&comp \quad (Wire\ \vec{p}) \quad = return\ (\lambda vs \to [vs\ !!\ p \mid p \leftarrow \vec{p}]) \\
&comp \quad (Seq\ x\ y) \quad = \mathbf{do}\ m \leftarrow comp\ y \\
&\qquad\qquad\qquad\qquad\qquad\quad n \leftarrow comp\ x \\
&\qquad\qquad\qquad\qquad\qquad\quad return\ (m \circ n) \\
&comp \quad (Par\ x\ y) \quad = \mathbf{do}\ ax \leftarrow arity\ x \\
&\qquad\qquad\qquad\qquad\qquad\quad m \leftarrow comp\ x \\
&\qquad\qquad\qquad\qquad\qquad\quad n \leftarrow comp\ y \\
&\qquad\qquad\qquad\qquad\qquad\quad return\ (\lambda vs \to \mathbf{let}\ (a, b) = splitAt\ ax\ vs \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathbf{in}\ \ m\ a +\!\!+ n\ b) \\
&comp \quad (Cond\ x\ y) = \mathbf{do}\ m \leftarrow comp\ x \\
&\qquad\qquad\qquad\qquad\qquad\quad n \leftarrow comp\ y \\
&\qquad\qquad\qquad\qquad\qquad\quad return\ (\lambda(v : vs) \to \mathbf{if}\ v\ \mathbf{then}\ True : (m\ vs) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathbf{else}\ \ False : (n\ vs))
\end{aligned}
$$

FIGURE 2.8: The *comp* compile function for classical reversible circuits

2.6 Summary

In this chapter the fundamental notion of how computations are modelled has been explored. Abstract models of computation, such as λ-calculus and Cartesian-closed categories are based on irreversible processes. However, more fundamental physical notions describe processes in closed systems and here all processes are reversible. This includes theories such as Newtonian mechanics, Maxwellian electrodynamics, and quantum mechanics. Open systems, which do allow for irreversible processes, are a derived notion; a subsystem of a larger closed system. This chapter followed the physical notion that reversibility is the fundamental concept to model computation, from which irreversibility may be derived. A framework for reversible computations has been presented in this chapter, with an implementation given in Haskell, and the

theory and practise of reversible computation was explored.

Chapter 3

Reversible quantum computation

This chapter will introduce the fundamentals of quantum computing, including the notation, mathematics, and theory. The quantum circuit model will be presented, using a development that follows the format used to describe reversible classical circuits introduced in section 2.2. Throughout the development the differences and similarities between classical reversible computation and quantum computing are highlighted and discussed.

An implementation of quantum circuits in Haskell is developed, including a compiler. A discussion of Deutsch's Algorithm and Quantum Teleportation is also included, with a presentation of the algorithms as quantum circuits.

3.1 Fundamentals of quantum computation

To understand quantum algorithms, and how they can achieve the efficiency gains they are theoretically capable of, it is first necessary to study the fundamental physics and mathematics of quantum computers. The focus of this section is not how quantum computers can be fabricated (though this is an active area of research, see section 1.2) but the how the behaviour of quantum computers can be modelled. This section outlines the theory of how quantum computation is modelled mathematically, and how quantum bits interact.

3.1.1 Introduction to quantum bits (qubits)

The quantum bit, also called a qubit, is the fundamental unit of quantum information, and is analogous to the classical bit. A qubit, like a classical bit, can exist in a state which is either of the classical states **0** (False) or **1** (True). These states are represented by $|0\rangle$ and $|1\rangle$, in the Dirac "Bra-ket" notation commonly used in quantum computing. The notation used defines qubits using a Ket column vector, and its row vector conjugate-transpose, the Bra. The Dirac vector notation is employed here because, unlike classical bits, qubits can also be in linear combinations of the states $|0\rangle$ and $|1\rangle$, which is called a *superposition* of states. A superpositions can be represented in Dirac notation as:

$$|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle = \begin{pmatrix}\alpha\\ \beta\end{pmatrix} \tag{3.1}$$

where $|\psi\rangle$ represents the qubit in a superposition. The state is represented by a vector in two-dimensional complex vector space, where α and β are both complex numbers representing the amplitudes of the two components of the quantum state. α and β have the additional property that $|\alpha|^2$ is the probability of the qubit collapsing to $|0\rangle$ when measured, and $|\beta|^2$ is the probability of the qubit collapsing to $|1\rangle$. The states $|0\rangle$ and $|1\rangle$ are known as the computational basis states, and form an *orthonormal* basis for the vector space. When a qubit state is measured in the computational basis, it will always collapse from its superposition into one of the basis states, with a probability equal to the absolute-square of its amplitude of that state prior to measurement. The probabilities sum to 1; $|\alpha|^2 + |\beta|^2 = 1$. This is a projective measurement, and is an irreversible process. Measurement is a key feature of quantum physics, and some algorithms specify measurement in a basis other than the computational basis; an example of which is a measurement in the diagonal basis. Measurements in a different basis can be expressed by a unitary change of basis, followed by a measurement with respect to the computational basis, followed by the inverse of the basis change operation. As measurement is an irreversible process, a full discussion is postponed until section 5.5.2, where measurement is formalised using the notion of partial trace on superoperators.

The Ket vector $|\varphi\rangle = \binom{\alpha}{\beta}$, has an associated Bra vector $\langle\varphi| = (\alpha^*, \beta^*)$, where

$(x + y \times i)^* = x - y \times i$ is the complex conjugate. The bra is, by definition, the adjoint (conjugate transpose) of the ket:

$$|\varphi\rangle^\dagger = \begin{pmatrix}\alpha\\ \beta\end{pmatrix}^\dagger = \begin{pmatrix}\alpha^*\\ \beta^*\end{pmatrix}^T = (\alpha^*, \beta^*) = \langle\varphi|$$

The inner–product, discussed in section 3.1.3, is then given by $\langle\phi \mid \varphi\rangle$; the "bracket" of the Dirac notation.

Relationship to probabilistic computation

Quantum amplitudes initially seem to bear similarities to probabilistic computation, where a computation can be in a mixed state such as $p_0[0] + p_1[1]$, where p_0 and p_1 are the probabilities of the computation being in either of the classical states 0 or 1, respectively. Here similarly $p_0 + p_1$ must sum to 1. However, there are two fundamental differences between the probabilities in probabilistic mixed states and the amplitudes of quantum states: The first difference is that probabilities can only be positive numbers between 0 and 1, whereas quantum amplitudes can be negative and can include an imaginary component (as they are complex–valued), with the restriction that $|\alpha|^2 + |\beta|^2 = 1$. As the amplitudes of quantum states can have negative values (a negative phase), these would be subtracted from any probability calculation. This means that quantum states may *destructively* interfere with each other, thus decreasing the probabilities of other states from occurring. Classically, there is no analogous idea of a negative probability.

The second difference is that, with a probabilistic computation, the probabilities describe our lack of knowledge about the current state of the system. The system itself is only ever in one of the possible states at any time, with the evolution of the system being fully described by a probability chain. However, the amplitudes of a quantum state do not describe imperfect knowledge of the system; rather they describe the *actual*, though unknown, current state of the system. A quantum bit *can* to some degree (prescribed by the amplitudes), be in *both* of the possible states simultaneously; this is a superposition.

For example, a random classical bit can be understood by imagining a coin toss. A tossed coin only ever lands to show heads or tails. If the coin is tossed and then

covered, the coin is either showing heads or tails; it must be in one of those two states. By looking the incorrect possibility is removed. A "quantum coin" is different in that it can exist in a superposition of states between heads and tails while it is covered, and only collapses to one state when observed, in a similar vein to the famous Schrödinger's Cat thought-experiment [67]. If the quantum coin is measured, but the result not observed, it becomes a classical probabilistic coin. The quantum state has collapsed and our lack of knowledge about which state it collapsed into is represented by a probability.

Physical realisations

Often, in physics, a physical quantity which may be used experimentally as a qubit is the spin of an electron. Spin is a quantum mechanical property of some subatomic particles (*fermions*) which can exist in one of two opposing (orthogonal) states, which physicists conventionally denote $|+\rangle$ and $|-\rangle$[1]. Note that fermions do not actually spin about an axis.

The spin of an electron is a quantum property which can be denoted as a vector from the centre to the surface of a unit sphere, and given a geometric interpretation: the Bloch Sphere, shown in figure 3.1. Every point on the surface of the Bloch sphere gives a possible value for the spin, with each point having different amplitudes; *i.e.* different α and β.

Electron spin has the property that, although it can point in any direction, when measured it will only ever be found to be *up* or *down* (hence $|+\rangle$ and $|-\rangle$), with a probability proportional to the complex valued components of the original spin. The quantum effect of electron spin was first demonstrated in the famous Stern–Gerlach experiment [29].

Equation 3.1 for a qubit gives $|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle$. Since $|\alpha|^2 + |\beta|^2 = 1$, by Euler's Formula [24], equation 3.1 can be rewritten as:

$$|\psi\rangle = e^{i\gamma}(\cos\frac{\theta}{2}\,|0\rangle + e^{i\varphi}\sin\frac{\theta}{2}\,|1\rangle)$$

[1]The computer science convention of $|0\rangle$ and $|1\rangle$ will be used here, where $|+\rangle = \frac{1}{\sqrt{2}}\,|0\rangle + \frac{1}{\sqrt{2}}\,|1\rangle$ and $|-\rangle = \frac{1}{\sqrt{2}}\,|0\rangle - \frac{1}{\sqrt{2}}\,|1\rangle$.

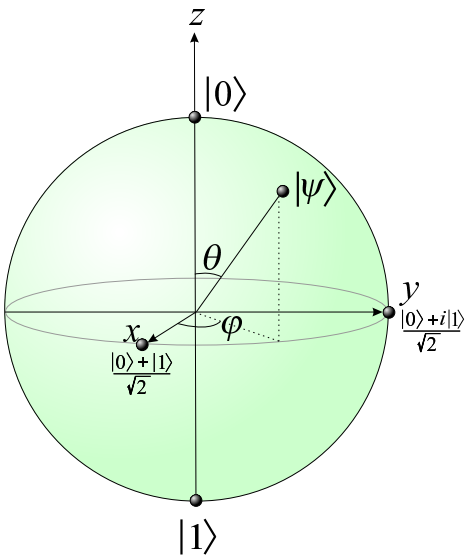


FIGURE 3.1: The Bloch Sphere: a geometric representation of a qubit

where θ, φ and γ are real numbers. The factor $e^{i\gamma}$ can be ignored in this case as it has no observable effect; it is a normalisation factor of mathematical interest. This leaves:

$$|\psi\rangle = \cos\frac{\theta}{2}|0\rangle + e^{i\varphi}\sin\frac{\theta}{2}|1\rangle$$

The variables θ and φ can now be interpreted as defining a point on the Bloch sphere, with θ being the angle from the z axis, and φ the angle from the x axis. The Bloch sphere can be used to visualise a quantum bit, and the spin of an electron is a realisation of this geometric representation of a quantum bit.

3.1.2 Multiple qubit registers

A multiple qubit system is an array of single qubits, but they are not treated as a simple collection of single bits, like bit–strings in a classical computer are. The individual qubits in a quantum register can be *entangled*, due to their quantum mechanical nature, with the possibility of their amplitudes interfering. Interference is the property that makes parallel quantum computing so potentially powerful.

A two qubit system is denoted mathematically as:

$$|\psi\rangle = \alpha_{00}|00\rangle + \alpha_{01}|01\rangle + \alpha_{10}|10\rangle + \alpha_{11}|11\rangle = \begin{pmatrix} \alpha_{00} \\ \alpha_{01} \\ \alpha_{10} \\ \alpha_{11} \end{pmatrix}$$

where $|\alpha_x|^2$ is the probability of state x being observed when the state is measured. The sum of the probabilities must still equal 1; $\sum_x |\alpha_x|^2 = 1$.

Qubit registers can be made by grouping individual qubits together in some way. The details of this vary according to the physical realisation, but conceptually if there are two qubits, $|x\rangle$ and $|y\rangle$, then $|xy\rangle$ can be called a two-qubit register. The action of combining the two qubits into a single register is modelled on the underlying vector space as $|xy\rangle = |x\rangle \otimes |y\rangle$, where $\otimes$ is the tensor product.

Although quantum registers can be constructed by combining individual qubits, once grouped together and operated on, they can no longer be considered independent of one another. For example, the two qubit state:

$$|epr\rangle = \frac{|00\rangle + |11\rangle}{\sqrt{2}} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ 0 \\ 0 \\ 1 \end{pmatrix} \tag{3.2}$$

cannot be described as the tensor of two individual qubits. The $|epr\rangle$ state is *entangled*, which means that the two individual qubits are now correlated. This is actually a well-known state, first formulated by Einstein, Podolsky, and Rosen as the "EPR paradox" [23]. Measuring the first qubit of this state would result in either 0 (with the resulting state $|00\rangle$) or 1 ($|11\rangle$), each with a probability of $\frac{1}{2}$. In either case, subsequently measuring the second qubit gives a deterministic (non-probabilistic) result, which is always equal to the first measurement. This correlation holds even if the two qubits are separated in any way. Experiments have shown that this correlation exists even when the two states are separated by over 10km [77]. This results demonstrates a key difference between quantum and classical systems, in which an entangled state

of two qubits cannot be expressed as the tensor product of single-qubit states. Classical systems, conversely, can always be decomposed into the Cartesian product of single-bit states. The EPR state presented in this way is sometimes referred to as the Bell state, with the EPR state then defined as an anti-correlated pair of qubits instead, which is closer to the original statement of the EPR thought experiment.

In general, a system composed of n quantum bits can exist in 2^n possible states. Unfortunately there is no simple geometric interpretation, such as the Bloch sphere, for more than one qubit, which makes them initially difficult to understand conceptually.

3.1.3 Operations on quantum states

The evolution of a quantum system can be described by a unitary transformation (operator). If the state of a qubit is represented as a complex-valued column vector (as above), then a unitary operator can be represented as a complex-valued matrix U, such that $U^{-1} = U^\dagger$, where $U^\dagger$ is the conjugate-transpose, or adjoint, of U, where U is a unitary matrix:

$$U^\dagger = \begin{pmatrix} u_{00} & u_{01} \\ u_{10} & u_{11} \end{pmatrix}^\dagger = \begin{pmatrix} u_{00}^* & u_{01}^* \\ u_{10}^* & u_{11}^* \end{pmatrix}^T = \begin{pmatrix} u_{00}^* & u_{10}^* \\ u_{01}^* & u_{11}^* \end{pmatrix}$$

The description of the behaviour of the operator U on a state is given by matrix multiplication:

$$U\left|\phi\right\rangle = \begin{pmatrix} u_{00} & u_{01} \\ u_{10} & u_{11} \end{pmatrix} \begin{pmatrix} \alpha \\ \beta \end{pmatrix} = \begin{pmatrix} u_{00}\alpha + u_{01}\beta \\ u_{10}\alpha + u_{11}\beta \end{pmatrix}$$

A common unitary operator used on quantum states is the Hadamard operator, H, which is sometimes called the "square-root of NOT" (despite the fact that $H^2 \neq \text{NOT}$). The Hadamard operator's action is given by the matrix:

$$H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \tag{3.3}$$

A unitary transformation can also be fully described by its action on the basis states, which can extended linearly to the entire space the operator acts on. The Hadamard

operator written in this way becomes:

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

$$H|1\rangle = \frac{1}{\sqrt{2}}|0\rangle - \frac{1}{\sqrt{2}}|1\rangle = |-\rangle$$

The action of the Hadamard transformation is to, in each case, produce an equal quantum superposition, but with a difference in phase. The negative phase in the case of $H|1\rangle$ does not change the measurement probabilities, as these are given by the absolute square of the amplitude. Note that it is clear from this representation that the Hadamard operation maps the orthogonal spaces $|0\rangle$ and $|1\rangle$ to the orthogonal spaces $|+\rangle$ and $|-\rangle$. Preserving orthogonality is a property of all unitary operations, otherwise they would not be sufficient to model quantum computation. More generally, unitary operations preserve the inner-product:

$$\langle v|w\rangle = \langle Uv|Uw\rangle \in \mathbb{C}$$

and the inner-product can be thought of as a measure of orthogonality: if $\langle v|w\rangle = 0$ then v and w are orthogonal ($v \perp w$).

The phase difference between the two possible outcomes formed on application of the Hadamard transform to the two basis states means that applying the transform again restores the original quantum state. This is evident from multiplying the Hadamard matrix with itself, which gives the identity matrix:

$$H^2 = \left(\tfrac{1}{\sqrt{2}}\begin{pmatrix}1 & 1\\ 1 & -1\end{pmatrix}\right)^2 = \frac{1}{2}\begin{pmatrix}2 & 0\\ 0 & 2\end{pmatrix} = \begin{pmatrix}1 & 0\\ 0 & 1\end{pmatrix} = I$$

Four other common one-qubit unitary rotations, so called because they rotate a vector about the Bloch sphere, are the Pauli matrices. These are:

$$I = \begin{pmatrix}1 & 0\\ 0 & 1\end{pmatrix}; \qquad X = \begin{pmatrix}0 & 1\\ 1 & 0\end{pmatrix}; \qquad Y = \begin{pmatrix}0 & -i\\ i & 0\end{pmatrix}; \qquad Z = \begin{pmatrix}1 & 0\\ 0 & -1\end{pmatrix} \tag{3.4}$$

Unitary transformations can be combined using the tensor product $\otimes$ to give a single transform which acts on the state space spanned by both operators. With

matrices the tensor product is sometimes called the Kronecker product, a term used to make clear that the result has a particular block structure imposed upon it, in which each element of the first matrix is replaced by the second matrix, scaled by that element:

$$U \otimes V = \begin{bmatrix} u_{00}V & u_{01}V & \cdots \\ u_{10}V & u_{11}V & \\ \vdots & & \ddots \end{bmatrix} = \begin{bmatrix} u_{00}v_{00} & u_{00}v_{01} & \cdots & u_{01}v_{00} & u_{01}v_{01} & \cdots \\ u_{00}v_{10} & u_{00}v_{11} & & u_{01}v_{10} & u_{01}v_{11} & \\ \vdots & & & & & \\ u_{12}v_{00} & u_{10}v_{01} & & \ddots & & \\ u_{12}v_{10} & u_{10}v_{11} & & & & \\ \vdots & & & & & \end{bmatrix} \tag{3.5}$$

For example, given two qubits, the operation $X \otimes Y$ corresponds to the unitary transform that would apply the X transform to the first qubit and the Y transform to the second.

3.2 Linear algebra for quantum computation in Haskell

The aim of this section is to provide an overview of basic ideas used in linear algebra which are used to mathematically model closed quantum systems, and to implement the concepts presented in the functional programming language Haskell [55], see section 2.3.

3.2.1 Pure quantum states as vectors

Linear algebra concerns the study of vector spaces and the linear operations on those spaces. The vector space of interest in (finite) quantum computing is the space of all (finite) n-tuples of complex numbers, denoted $\mathbb{C}^n$: the space of qubit registers. These can be thought of as a function from some basis A, which is a finite spanning set, into a complex number, such that $v \in A \to \mathbb{C}$. In the case of quantum computation over the computational basis states, the base will be the spanning set of the Boolean values of length n, since this spans all possible computational values of n-bits. For

full introduction and development of vector spaces and linear algebra, see reference [62].

The Haskell implementation of such a vector is:

$$\textbf{data}\ Vec = Vec\{\, vSize \in Int, funV \in [\mathbb{N}_2] \to \mathbb{C}\}$$

$$vreturn \in [\mathbb{N}_2] \to Vec$$

$$vreturn \quad \vec{b} \quad = Vec\ (length\ \vec{b})\ (\lambda b \to \textbf{if}\ b \equiv \vec{b}_a\ \textbf{then}\ 1\ \textbf{else}\ 0)$$

where $vSize$ is the size of the vector, and $funV$ is the function defining the vector. The function $vreturn$ is a vector return function, which returns a vector with a 1 in the index associated with the input, and 0s in all others, and $\mathbb{N}_2$ denotes the type of Booleans. Note that as only vectors over the computational basis are considered, the dimension of a vector denotes how many elements of the computational basis it can represent. For example, a size of 1 means the vector can represent 1 qubit, and thus has a dimension of 2^1. A size of 2 means the vector can represent a pair of qubits, and thus has a dimension of 2^2. The EPR vector, discussed below, has a size of 2 in this notation.

To illustrate how vectors can be defined using the Vec datatype, examples of vectors are given below. The computational basis states of $|0\rangle$ and $|1\rangle$ ($False$ and $True$) are presented as the following vectors:

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

In Haskell, the basis states are defined as:

$$vF, vT \in Vec$$

$$vF = vreturn\ [False]$$

$$vT = vreturn\ [True]$$

A more interesting example is the EPR state vector shown in equation 3.2, and discussed in section 3.1.2. This can be written in Haskell as:

$$vEPR \in Vec$$

$$vEPR = Vec\ 2\ eprF$$

$$\begin{aligned} \textbf{where}\ eprF\ [False, False] &= 1/\sqrt{2} \\ eprF\ [True, True] &= 1/\sqrt{2} \\ eprF\ \_ &= 0 \end{aligned}$$

where *eprF* is a function that defines the action of the EPR state vector: $1/\sqrt{2}$ if either $|00\rangle$ or $|11\rangle$ are input, and 0 otherwise.

The basis of a vector is a simple function of its size, and code for fully enumerating the base is:

$$
\begin{aligned}
&\mathit{base} \in \mathit{Int} \to [[\mathbb{N}_2]] \\
&\mathit{base} \quad 0 \quad = [[\,]] \\
&\mathit{base} \quad n \quad = [\,\mathit{True} : \vec{b} \mid \vec{b} \leftarrow \vec{b'}] +\!\!+ [\,\mathit{False} : \vec{b} \mid \vec{b} \leftarrow \vec{b'}] \\
&\qquad\quad \textbf{where } \vec{b'} = \mathit{base}\ (n-1)
\end{aligned}
$$

The *base n* function can be thought of as producing a list of all possible indexes of a vector of dimension 2^n. For example, *base* 2 would produce the list below:

$$[[\,\mathit{True}, \mathit{True}], [\,\mathit{True}, \mathit{False}], [\mathit{False}, \mathit{True}], [\mathit{False}, \mathit{False}]]$$

There are several standard operations on vectors defined in linear algebra that are useful in quantum computation, and these are encapsulated in the *VEC* class:

$$
\begin{aligned}
&\textbf{class } \mathit{VEC}\ a\ m \mid a \to m \textbf{ where} \\
&\quad \mathit{adjoint} \in a \to a \\
&\quad (\otimes) \quad \in a \to a \to a \quad \text{-- Tensor Product: } v \otimes w \\
&\quad (\langle\cdot\rangle) \quad \in a \to a \to \mathbb{C} \quad \text{-- Inner Product: } \langle v|w\rangle \\
&\quad (\rangle\cdot\langle) \quad \in a \to a \to m \quad \text{-- Outer Product: } |v\rangle\langle w| \\
&\quad (\$*) \quad \in \mathbb{C} \to a \to a \quad \text{-- Scalar Product: } \lambda \times v
\end{aligned}
$$

where $v, w \in a$, $\lambda \in \mathbb{C}$, and a is a type for which this class has been instantiated. The type *Vec* of vectors must by definition be able to have these operations defined for it, and the class can be instantiated with the following definitions:

$$
\begin{aligned}
&\textbf{instance } \mathit{VEC}\ \mathit{Vec}\ \mathit{Mat} \textbf{ where} \\
&\quad \mathit{adjoint}\ (\mathit{Vec}\ a\ v) \quad = \mathit{Vec}\ a\ v^\dagger \\
&\qquad \textbf{where } f^\dagger\ x = (f\ x)^* \\
&\quad \mathit{Vec}\ a\ v \otimes \ \mathit{Vec}\ a'\ w = \mathit{Vec}\ (a + a') \\
&\qquad\qquad\qquad\qquad (\lambda\vec{b} \to \textbf{let } (b_1, b_2) = \mathit{splitAt}\ a\ \vec{b} \\
&\qquad\qquad\qquad\qquad\quad \textbf{in } v\ b_1 \times w\ b_2) \\
&\quad \mathit{Vec}\ a\ v\langle\cdot\rangle \ \ \mathit{Vec}\ a'\ w \mid a \equiv a' = \mathit{sum}\ [(v\ \vec{a})^* \times w\ \vec{a} \mid \vec{a} \leftarrow \mathit{base}\ a] \\
&\qquad\qquad\qquad\quad \mid \mathit{otherwise} = \mathit{error}\ \texttt{"Vec: |<.>| Outer Product"}
\end{aligned}
$$

$$\begin{aligned} Vec\ a\ v\rangle\cdot\langle\ \ Vec\ a'\ w &= Mat\ a\ a'\ (\lambda\vec{b}_a \to \lambda\vec{b}_b \to v\ \vec{b}_a \times (w\ \vec{b}_b)^*) \\ x \qquad \$*\ \ Vec\ a\ v\ \ &= Vec\ a\ (\lambda b \to x \times (v\ b)) \end{aligned}$$

The instantiation of this class for *Vec* follows the standard linear algebra definitions for vectors. This code also makes use of the type *Mat* of matrices, which are defined in the following section.

3.2.2 Quantum operations as matrices

As a vector is represented as a function $v \in A \to \mathbb{C}$, a matrix is represented as a function of type $m \in A \to B \to \mathbb{C}$. Vectors denote quantum states, and unitary matrices denote quantum operators (see section 3.1.3). This is encapsulated by the *Mat* type:

$$\textbf{data}\ Mat = Mat\{\, inS \in Int, outS \in Int, funM \in [\mathbb{N}_2] \to [\mathbb{N}_2] \to \mathbb{C}\}$$

where *inS* and *outS* return the size of the input and output vectors, respectively, and *funM* defines the function that is the action of the linear operator. As with *Vec*, the basis is of *Mat* restricted to the computational basis of Booleans, $\mathbb{N}_2$.

As an example, the matrix for negation (the Pauli-X matrix from equation 3.4), can be defined in Haskell as:

$$\begin{aligned} &mNot \in Mat \\ &mNot = Mat\ 1\ 1\ notF \\ &\qquad \textbf{where}\ notF\ [x]\ [y] \mid x \not\equiv y \quad = 1 \\ &\qquad\qquad\qquad\qquad\quad \mid otherwise = 0 \\ &\qquad\qquad\quad notF\ \_\ \_ \quad = error\ \texttt{"mNot Arity"} \end{aligned}$$

The code for *mNot* defines a linear operator between vectors of a single Boolean, and the action of the operator is to map *True* to *False*, with a probability of 1, and vice-versa.

The matrix for the Hadamard transform (see section 3.1.3) can be similarly encoded as:

$$\begin{aligned} &mHad \in Mat \\ &mHad = Mat\ 1\ 1\ hadF \\ &\quad \textbf{where}\ hadF\ [True]\ [True] = -1/\sqrt{2} \end{aligned}$$

$$
\begin{aligned}
hadF\ [True]\ [False] &= 1/\sqrt{2} \\
hadF\ [False]\ [False] &= 1/\sqrt{2} \\
hadF\ [False]\ [True] &= 1/\sqrt{2} \\
hadF\ \_\ \ \ \_ &= error\ \texttt{"mHad Arity"}
\end{aligned}
$$

mHad is again a matrix between vectors representing a single qubit, and the action of the function is fully enumerated and can be seen to be equal to that given in equation 3.3.

The useful operations encapsulated in the *VEC* class can also be instantiated for the *Mat* type, using the standard definition of each operator for matrices:

$$
\begin{aligned}
&\textbf{instance}\ VEC\ Mat\ Mat\ \textbf{where} \\
&\quad adjoint\ (Mat\ a\ b\ m) = Mat\ a\ b\ m^{\dagger} \\
&\qquad \textbf{where}\ f^{\dagger}\ x\ y = (f\ y\ x)^{*} \\
&\quad Mat\ a\ b\ m \otimes\ \ Mat\ a'\ b'\ n \\
&\qquad\qquad = Mat\ (a+a')\ (b+b') \\
&\qquad\qquad\qquad (\lambda \vec{b}_a \rightarrow \lambda \vec{b}_b \rightarrow \\
&\qquad\qquad\qquad\quad \textbf{let}\ (a_1, a_2) = splitAt\ a\ \vec{b}_a \\
&\qquad\qquad\qquad\qquad\ (b_1, b_2) = splitAt\ b\ \vec{b}_b \\
&\qquad\qquad\qquad\quad \textbf{in}\ (m\ a_1\ b_1) \times (n\ a_2\ b_2)) \\
&\quad Mat\ a\ b\ m\langle\cdot\rangle\ \ Mat\ \_\ \_\ n \\
&\qquad\qquad = sum\ [(m\ \vec{b}\ \vec{a})^{*} \times (n\ \vec{a}\ \vec{b}) \\
&\qquad\qquad\qquad\quad \mid \vec{a} \leftarrow base\ a, \vec{b} \leftarrow base\ b] \\
&\quad m \qquad\quad \rangle\cdot\langle\ \ n = m \gg\!= adjoint\ n \\
&\quad x \qquad\quad \$*\ \ Mat\ a\ b\ m = Mat\ a\ b\ (\lambda \vec{b}_a \rightarrow \lambda \vec{b}_b \rightarrow x \times (m\ \vec{b}_a\ \vec{b}_b))
\end{aligned}
$$

The most complicated function is the tensor product, $\otimes$, which follows the definition given in equation 3.5.

It now remains to define a method of composing matrices and vectors, and this is done using the standard notion of matrix multiplication. This is defined as the $\gg\!=$ operation (pronounced *bind*), as it has been shown that, although technically not a monad [82], vectors and linear operators correspond to a Kleisli structure [4], which is a more general notion, sometimes referred to as an indexed monad. The difference

between monads and indexed monads is that the function is not required to be an endofunctor; the function is only defined for a subset of the objects. This is exactly the notion required to model quantum state vectors as monads: a function which associates each element of the basis with a complex-valued probability amplitude. A quantum vector can only act over types which are constituents of the basis [82]. The three monad laws still apply.

A class of types with a $\gg=$ operator can be defined simply as:

class *Bind a b* **where**

$(\gg=) \in a \to b \to a$

In the case of *Vec* the return operation is the function *vreturn*, defined previously, and the $\gg=$ operator can be instantiated as shown in figure 3.2, which defines the

instance *Bind Vec Mat* **where**

Vec d vf $\gg=$ *Mat m n mf*

| $d \equiv m$ = *Vec n*

$(\lambda\vec{b} \to$ **if** *length* $\vec{b} \equiv n$

then *sum* $[vf\ \vec{a} \times (mf\ \vec{a}\ \vec{b}) \mid \vec{a} \leftarrow base\ d]$

else *error* `"Size mismatch"`)

| *otherwise* = *error* (`"Type error: |Vec @>>= Mat|"`)

FIGURE 3.2: The bind operation between a vector and a matrix

$\gg= \in Vec \to Mat \to Vec$ operation as simple matrix multiplication of a matrix with a vector. In the case of $\gg= \in Mat \to Mat \to Mat$, the instantiation is again matrix multiplication, as shown in figure 3.3.

instance *Bind Mat Mat* **where**

Mat a b f $\gg=$ *Mat a′ b′ f′*

| $b \equiv a'$ = *Mat a b′* $(\lambda\vec{b}_a \to funV\ ((Vec\ b\ (f\ \vec{b}_a)) \gg= Mat\ a'\ b'\ f'))$

| *otherwise* = *error* (`"Type error: |Mat @>>= Mat|"`)

FIGURE 3.3: The bind operation between matrices

The bind operation presented here, which is a matrix multiplication, gives a description of the evolution of a quantum system, and can be thought of as a way of sequencing quantum computations. The return can be thought of as a way of terminating computations. This fits the idea of a monad as a way of encapsulating computation, and was originally noted by Bird and Mu [52] and developed by Altekirch, Vizzotto and Sabry [84], where a full proof that the monad laws are satisfied is presented. The three monad laws are:

$$(return\ x) \gg\!= f = f\ x \qquad \text{-- left-identity with respect to } \gg\!=$$
$$m \gg\!= return = m \qquad \text{-- right-identity with respect to } \gg\!=$$
$$(m \gg\!= f) \gg\!= g = m \gg\!= (f \gg\!= g) \qquad \text{-- associativity law for } \gg\!=$$

and the proof that these hold follows exactly that presented in references [84, 82], which is extended by Vizzotto in her PhD thesis [83]. Section 5.3.1 gives a categorical review of the material presented in this section.

3.3 The quantum circuit model

In this section an inductive definition for building quantum circuits will be given. The quantum circuit model is a standard way of expressing quantum algorithms [53]. The presentation in this thesis will be slightly different than standard, in that it follows the same format as the definition of reversible classical circuits given in section 2.2.

As with the classical circuit presentation, the quantum circuit definition will be translated into a Haskell datatype, which can be compared with the classical implementation in section 2.3. An analysis of both the differences and similarities will then follow. As shown previously, the arity of a circuit is number of qubits, or (quantum) wires, used throughout the circuit, and is therefore a natural number. By definition, a unitary computation ϕ must have an inverse, ϕ^{-1}, such that together ϕ, ϕ^{-1} give an isomorphism. The notation $\mathcal{Q}_2$ is now introduced to denote a single qubit.

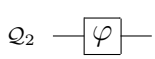
The set of quantum computations (or circuits) of arity $a \in \mathbb{N}$, can be defined inductively:

Rotation In contrast to classical reversible computation, where there is only one non-trivial operation possible on one bit (negation), when dealing with a quantum

bit any one qubit unitary transformation, *rot* φ, is a valid operation. Denoted as a matrix, φ must be unitary and of the form:

$$\begin{pmatrix} \lambda_0 & \lambda_1 \\ \kappa_0 & \kappa_1 \end{pmatrix}$$

with $\lambda_0^* \kappa_0 + \lambda_1^* \kappa_1 = 0$. As a circuit diagram *rot* φ is denoted as:



Negation is a particular rotation given by *rot* X where X is given by:

$$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

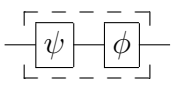
As all rotations are unitary, the inverse is given by the adjoint of φ, $\varphi^\dagger$. The inverse construction is *rot* φ^{-1}, with:

$$\varphi^{-1} = \varphi^\dagger = \begin{pmatrix} u_{00} & u_{01} \\ u_{10} & u_{11} \end{pmatrix}^\dagger = \begin{pmatrix} u_{00}^* & u_{10}^* \\ u_{01}^* & u_{11}^* \end{pmatrix}$$

Wires Any possible reordering (permutation) of qubits is a valid operation, and in a circuit is shown as a rewiring. This is represented as a bijection $\phi : [a] \simeq [a]$, where $[a]$ is the initial segment of a, defined as $\{ i \in \mathbb{N} \mid i < a \}$. This describes any rewiring, including the identity $id_a = \mathit{wires}\ id$, where no permutation takes place. This is the same as in the classical case, and is shown the same way schematically, except with the "wires" now carrying quantum data.

As before, rewiring is trivially reversible, and the arity of the rewiring circuit is the number of wires in the permutation, which is a in the bijection.

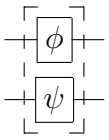
Sequential composition Given quantum circuits ϕ and ψ, of equal arity, a new circuit can be constructed where the output of ψ is passed as the input to ϕ. This is written $\phi \circ \psi$, and is shown diagrammatically as:



The arity of the new circuit is equal to the arity of the sub-circuits. As both ϕ and ψ are unitary, $\phi \circ \psi$ can be reversed using ψ^{-1} and ϕ^{-1}, and by constructing $\psi^{-1} \circ \phi^{-1}$.

This is exactly the same construction as for the classical case.

Parallel composition This combines any two circuits in parallel, and can be thought of as the tensor product of circuits in the quantum case. Given any quantum circuits ϕ and ψ, $\phi \otimes \psi$ can be constructed, shown diagrammatically as:

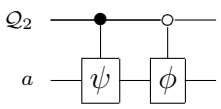


The arity of the new circuit is the sum of the sub-circuit arities, and the inverse can be constructed using ψ^{-1} and ϕ^{-1}, to give $\psi^{-1} \otimes \phi^{-1}$.

This is the same construction as in the classical case, except that classically the Cartesian product is used to build the product circuit. In the quantum case the tensor product is used. The tensor product acts on the Hilbert space representation, which is the Cartesian product on the underlying basis.

Conditional Given quantum circuits ϕ and ψ of equal arity, a, the conditional circuit $\phi|\psi$ can be constructed. It performs the same basic $if \ldots then \ldots else \ldots$ operation as in the classical case, except now the control wire could be in a quantum superposition. If the control wire is in a superposition then both ϕ and ψ are applied to the appropriate components of their inputs.

The arity of the circuit is $1 + a$ and it is shown schematically as:



If either ϕ or ψ is the identity, then the diagram can be simplified by leaving out the relevant branch, as in the classical case, and the inverse is once again given using ψ^{-1} and ϕ^{-1}, to construct $\phi^{-1}|\psi^{-1}$.

3.4 Comparing classical and quantum circuits

By comparing the definition of quantum circuits presented above, with the definition of classical circuits presented in section 2.2, it should be clear that there are striking similarities. The main difference is that reversible classical computations are bijections between finite sets, while reversible quantum computations are unitary operations between finite dimensional Hilbert spaces; which seems to imply that these notions are fundamentally unalike. However, the Hilbert spaces of quantum computations are generated by finite sets, and unitary operators on Hilbert spaces are the quantum mechanical equivalent of bijections between finite sets.

Two further differences are evident from the inductive definitions of classical and quantum circuits: the use of the tensor product rather than the Cartesian product in the definition of sequential composition of circuits, and the general rotation operator which replaces the classical negation operator. Both of these differences follow from the move to Hilbert spaces in the quantum case. The tensor product is the Hilbert space equivalent of the product operation, and is modelled as the Cartesian product on the underlying finite set that generates the Hilbert space. Classically, negation is the only non-trivial reversible operation on a single bit. In the quantum setting, because of the move to Hilbert space, there are now an infinite number of one qubit operations, as there are an infinite number of unitary operations on one qubit. The power of quantum over classical computing is derived as consequence of these few differences. Using an operation such as the Hadamard transform, a single qubit can be placed into an equal superposition, and then used as the control qubit in a conditional operation. From this, quantum entanglement can be produced, and quantum parallelism, which together are the keys to the power of quantum algorithms.

These similarities have been made clear in this development as the definition of quantum circuits proceeded by mapping each concept from the definition of reversible classical circuits to the quantum mechanical equivalent, summarised in figure 3.4.

Classical Case	Quantum Case
Finite sets	Finite dimensional Hilbert spaces
Cartesian product ($\times$)	Tensor product ($\otimes$)
Bijections	Unitary operators

FIGURE 3.4: Table showing the analogous concepts of reversible classical and quantum computation. To give the quantum analogue, each classical concept is lifted to finite dimensional Hilbert spaces.

3.5 Quantum circuits in Haskell

The quantum circuits defined in section 3.3 can be directly translated into a Haskell datatype in the same way as the translation was performed on classical reversible circuits in section 2.3. A compiler function can be built using this datatype, following the classical development in chapter 2. Again, other useful functions, such as an evaluator for the quantum circuits, can also be defined. The first step is to define a datatype that represents circuits. This will also be called *Circ*, with quantum circuits assumed unless stated from now on.

$$
\begin{aligned}
\textbf{data}\ \mathit{Circ} = {} & \mathit{Rot}\ (\mathbb{C},\mathbb{C})\ (\mathbb{C},\mathbb{C}) \\
\mid {} & \mathit{Wire}\ [\mathit{Int}] \\
\mid {} & \mathit{Par}\ \ \mathit{Circ}\ \mathit{Circ} \\
\mid {} & \mathit{Seq}\ \ \mathit{Circ}\ \mathit{Circ} \\
\mid {} & \mathit{Cond}\ \mathit{Circ}\ \mathit{Circ}
\end{aligned}
$$

FIGURE 3.5: A datatype for quantum circuits in Haskell

This definition of *Circ* allows for a quantum circuit to be any one of the following: a *Rot*ation operation; a *Wire*, which takes as an argument a list of integers describing the permutation (rewiring); a *Par* (parallel) operation, which takes two sub-circuits as arguments; a *Seq* (sequential) operation, which also takes two sub-circuits as arguments; and finally a *Cond* (conditional) operation, which takes the two possible circuits as arguments. The *Circ* datatype bears a striking resemblance to that devel-

oped for classical circuits. Indeed, the only difference is that the *Not* circuit has been replaced by a *Rot*. The rotation *Rot* takes two pairs of two complex numbers. These are used to define the unitary matrix for the rotation:

$$Rot\ (\lambda, \lambda')\ (\kappa, \kappa') \equiv rot \begin{pmatrix} \lambda & \lambda' \\ \kappa & \kappa' \end{pmatrix}$$

As in the classical case this datatype does not capture all the constraints on what constitutes a valid circuit, for the same reasons presented in section 2.3. This can be overcome in precisely the same way: by defining the *arity* function for quantum circuits, shown in figure 3.6. This function makes use of the *Error* monad rather than the *Maybe* monad, which returns either $OK\ \ x$ if the computation succeeds ($Just\ x$ in the *Maybe* monad), otherwise $Error\ \ string$ where the string is a useful error message, rather than the *Maybe* monads *Nothing*. As with the *Maybe* monad, error results are propagated through the program, with the informative message. The *arity* function makes use of *eguard* to simplify the use of this monad, which takes as arguments *boolexp* and *string*. If the Boolean expression *boolexp* evaluates to *True* then the computation can proceed, otherwise $Error\ \ string$ is returned, with *string* detailing the source of the error. It also makes use of two auxiliary functions, *chkPerm* and *orthTest*. The function *chkPerm* ensures the permutation function is valid, as previously, whereas *orthTest* ensures that the rotation passed to *Rot* is valid, as defined in section 3.3.

3.6 Example quantum circuits

Using the Haskell *Circ* datatype it is straightforward to define examples of quantum circuits. In the reversible circuit datatype of section 2.3, the single bit negation operator *Not* is a primitive circuit. However, in the quantum case 1-qubit rotations need to be defined using the *rot* constructor, to which the unitary matrix that defines the rotation has to be supplied as an argument. A negation circuit, *notC*, can therefore be defined as:

$$notC \in Circ$$
$$notC = Rot\ (0,1)\ (1,0)$$

$$
\begin{aligned}
&\mathit{arity} && \in \mathit{Circ} \to \mathit{Error}\ \mathit{Int}\\
&\mathit{arity}\ (\mathit{Rot}\ x\ y) && = \mathbf{do}\ \mathit{eguard}\ (\mathit{orthTest}\ x\ y)\\
&&&\qquad (\texttt{"Orthogonality"} +\!\!+ \mathit{show}\ (x, y))\\
&&&\quad \mathit{return}\ 1\\
&\mathit{arity}\ (\mathit{Wire}\ \vec{x}) && = \mathbf{do}\ \mathit{eguard}\ (\mathit{chkPerm}\ \vec{x})\\
&&&\qquad (\texttt{"Wire: "} +\!\!+ \mathit{show}\ \vec{x})\\
&&&\quad \mathit{return}\ (\mathit{length}\ \vec{x})\\
&\mathit{arity}\ (\mathit{Cond}\ x\ y) && = \mathbf{do}\ m \leftarrow \mathit{arity}\ x\\
&&&\quad n \leftarrow \mathit{arity}\ y\\
&&&\quad \mathit{eguard}\ (m \equiv n)\\
&&&\qquad (\texttt{"Cond: arity ="} +\!\!+ \mathit{show}\ (m, n))\\
&&&\quad \mathit{return}\ (1 + m)\\
&\mathit{arity}\ (\mathit{Par}\ x\ y) && = \mathbf{do}\ m \leftarrow \mathit{arity}\ x\\
&&&\quad n \leftarrow \mathit{arity}\ y\\
&&&\quad \mathit{return}\ (m + n)\\
&\mathit{arity}\ (\mathit{Seq}\ x\ y) && = \mathbf{do}\ m \leftarrow \mathit{arity}\ x\\
&&&\quad n \leftarrow \mathit{arity}\ y\\
&&&\quad \mathit{eguard}\ (m \equiv n)\\
&&&\qquad (\texttt{"Seq: arity ="} +\!\!+ \mathit{show}\ (m, n))\\
&&&\quad \mathit{return}\ m\\
&\mathit{orthTest} && \in (\mathbb{C}, \mathbb{C}) \to (\mathbb{C}, \mathbb{C}) \to \mathbb{N}_2\\
&\mathit{orthTest}\ (\lambda, \lambda')\ (\kappa, \kappa') && = \lambda^* \times \kappa \equiv -\lambda'^* \times \kappa'\\
&\mathit{chkPerm} && \in [\mathit{Int}] \to \mathbb{N}_2\\
&\mathit{chkPerm}\ \vec{x} && = (\mathit{and}\ [\mathit{elem}\ x\ \vec{x} \mid x \leftarrow [0\,..\,\mathit{length}\ \vec{x} - 1]])
\end{aligned}
$$

FIGURE 3.6: The quantum circuit *arity* function

which follows the construction give in section 3.3.

Using this definition of negation, all the reversible classical circuits presented previously can be translated into quantum circuits. These circuits have the same action on the computational basis as in the classical case, but can now also accept superpositions of the basis as input. For example, the CNOT circuit, which negates the second input if the first is *True*, and does nothing otherwise, is defined as before:

$$cnot = rot\ X\,|\mathrm{id}_1$$

which gives an an almost identical Haskell implementation as in the classical case:

$$cnotC \in Circ$$
$$cnotC = Cond\ notC\ (Wire\ [0])$$

with *notC* replacing the classically primitive *Not* operation. Both the *notC* and *cnotC* circuits can be schematically represented in exactly the same way as in the classical case, except the type of the wires has now changed from $\mathbb{N}_2$ to $\mathcal{Q}_2$, to denote quantum data; shown in figure 3.7.

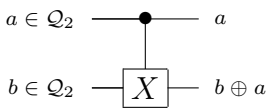


FIGURE 3.7: The simplified quantum CNOT circuit, where $\oplus$ denotes addition *mod* 2

Similarly, the Toffoli gate from section 2.4.1, and the Fredkin gate from section 2.4.2, can be translated in exactly the same way, giving only the Haskell implementation for brevity:

$$toffoliC \in Circ$$
$$toffoliC = Cond\ cnotC\ (Wire\ [0,1])$$
$$fredkinC \in Circ$$
$$fredkinC = Cond\ (Wire\ [1,0])\ (Wire\ [0,1])$$

As both of these gates are classically universal, as discussed in section 2.4, it follows that any classical function can be defined in this quantum setting: quantum circuits subsume classical circuits. However, it is important to note that these gates alone are not universal quantum gates. By universal quantum gates what is meant is a set of

quantum gates from which any quantum circuit can be approximated. In the quantum case, the CNOT operation, plus a small set of one qubit rotations, are universal. In reference [53] the single qubit rotations *Hadamard* (see section 3.1.3), *Phase* (S), and $\pi/8$ (T) are used, but many other options are possible. The rotations S and T are defined as:

$$S = \begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix}; \qquad T = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{pmatrix}$$

S is know as the phase-gate, while T is sometimes called the $\pi/8$-gate.

The universal rotations *had*, *rS*, and *rT*, for the Hadamard transform, Phase gate, and $\pi/8$ gate, are defined as simply *rot x*, where x is the appropriate matrix. In Haskell these become:

$$\begin{aligned}
&hadC \in Circ \\
&hadC = Rot\ (h,h)\ (h,-h)\ \textbf{where}\ h = 1/\sqrt{2} \\
&rS \quad \in Circ \\
&rS \quad = Rot\ (1,0)\ (0,i) \\
&rT \quad \in Circ \\
&rT \quad = Rot\ (1,0)\ (0,e^{i\pi/4})
\end{aligned}$$

For completion, the remaining Pauli operators from section 3.1.3 are implemented in Haskell as:

$$\begin{aligned}
&pI \ \in Circ \\
&pI \ = Rot\ (1,0) \quad (0,1) \\
&pX \in Circ \\
&pX = notC \\
&pY \in Circ \\
&pY = Rot\ (0,-i)\ (i,0) \\
&pZ \in Circ \\
&pZ = Rot\ (1,0) \quad (1,-1)
\end{aligned}$$

For further examples, the Deutsch algorithm is implemented as a quantum circuit in section 3.9, and the quantum teleport algorithm is implemented in section 3.10.

3.7 Simulating a quantum circuit

The quantum *Circ* datatype allows quantum circuits to be represented in Haskell, as the classical *Circ* datatype allowed classical circuits to be described. In order to run classical circuits a simulator is defined in section 2.5. A simulator for quantum circuits will now be defined, following the presentation given for classical circuits: a compiler function from circuits to circuits will be defined, and an evaluator which will apply a state to a circuit to generate a new state. Quantum states and operations are compiled into vectors and linear operators, using the *Vec* and *Mat* types defined in section 3.2.

3.7.1 Evaluating a quantum circuit

Performing the same function as in the classical case, the evaluation function uses the yet to be defined compilation function, *comp*, to compile a circuit into a matrix, and then applies an input vector to that matrix. This is done by simply multiplying the matrix m with the vector v, using the monad bind notation from section 3.2: $v \gg\!= m$. The output is then a vector which represents the quantum state of the input after applying the quantum circuit. The evaluation function is again called *eval*, and is defined as:

$$
\begin{aligned}
&\mathit{eval} \in \mathit{Circ} \to \mathit{Vec} \to \mathit{Error}\ \mathit{Vec} \\
&\mathit{eval}\quad c\qquad v\quad = \mathbf{do}\ a \leftarrow \mathit{arity}\ c \\
&\qquad\qquad\qquad\qquad\quad \mathit{eguard}\ (a \equiv \mathit{size}\ v) \\
&\qquad\qquad\qquad\qquad\qquad\quad (\texttt{"Input arity: "} +\!\!+ \mathit{show}\ a) \\
&\qquad\qquad\qquad\qquad\quad m \leftarrow \mathit{comp}\ c \\
&\qquad\qquad\qquad\qquad\quad \mathit{return}\ (v \gg\!= m)
\end{aligned}
$$

Note that the output type of this function is *Error Vec*; the *eval* function does not assume the circuit passed as an argument is well formed, and is compatible with the input. If the size of the quantum state input is not equal to the arity of the circuit passed, then an *Error* is returned with a informative error message - if the *arity* function itself does not propagate an error. If the *arity* function, defined in figure 3.6, detects any orthogonality or permutation errors, or a malformed circuit, then the

relevant error message is propagated and returned.

3.8 Compiling a quantum circuit

The *comp* function is where most of the work of simulating a quantum circuit takes place, as it translates a circuit in the *Circ* datatype into a linear operator represented in the *Mat* datatype. It works in the same way as the classical *comp* function, as a recursive function with *Rot* and *Wire* as base cases. *Seq*, *Par*, and *Cond* recursively call *comp* on their sub-circuits before doing the necessary computation. The code for the *comp* function fully describes the behaviour, and is given in figure 3.8.

3.9 The Deutsch algorithm

The Deutsch algorithm was one of the first quantum algorithms developed to show a clear advantage over the best possible classical algorithm. Although the algorithm solves no practical problem, Gay [27] observes that it

> "embodies what seem to be the essential aspects of an efficient [low-level] quantum algorithm: preparation of a superposed state, then application of unitary transformations in such a way as to take advantage of quantum parallelism and then concentrate the resulting global information into a single place, and finally an appropriate measurement."

The Deutsch algorithm is an oracle problem, which assumes the existence of a black-box which computes an unknown function $f \in \mathbb{N}_2 \to \mathbb{N}_2$. The purpose of the algorithm is to find out whether f is a constant function. Classically, the solution is simple: compute f *True* and f *False* and compare the results. This requires two queries of the oracle. The Deutsch Algorithm can produce an answer with only a single query to the oracle, by exploiting the advantages of a quantum system.

To compute the Deutsch algorithm a quantum version of the oracle is required. A lifting of f to a unitary transformation gives $\widehat{f}$ such that:

$$\widehat{f}\,|xy\rangle = |x\rangle\,|y \oplus (fx)\rangle$$

$$
\begin{aligned}
&comp && \in Circ \to Error\ Mat\\
&comp\ (Wire\ \vec{p}) && = return\ (Mat\ dim\ dim\ permuteF)\\
&\quad \textbf{where} && \quad dim = length\ \vec{p}\\
&&& \quad permuteF\ \vec{x}\ \vec{y} \mid \vec{y} \equiv [\vec{x} !!\ p \mid p \leftarrow \vec{p}] = 1\\
&&& \qquad\qquad \mid otherwise = 0\\
&comp\ (Seq\ c_1\ c_2) && = \textbf{do}\ m_1 \leftarrow comp\ c_1\\
&&& \quad m_2 \leftarrow comp\ c_2\\
&&& \quad return\ (m_1 \gg\!= m_2)\\
&comp\ (Par\ c_1\ c_2) && = \textbf{do}\ m_1 \leftarrow comp\ c_1\\
&&& \quad m_2 \leftarrow comp\ c_2\\
&&& \quad return\ (m_1 \otimes m_2)\\
&comp\ (Cond\ c_1\ c_2) && = \textbf{do}\ a \leftarrow arity\ c_1\\
&&& \quad b \leftarrow arity\ c_2\\
&&& \quad eguard\ (a \equiv b)\\
&&& \qquad \texttt{("Cond arity: "} +\!\!+ show\ (a, b))\\
&&& \quad m_1 \leftarrow comp\ c_1\\
&&& \quad m_2 \leftarrow comp\ c_2\\
&&& \quad \textbf{let}\ cond\ (True : \vec{x})\ (True : \vec{y}) = (funM\ m_1)\ \vec{x}\ \vec{y}\\
&&& \qquad cond\ (False : \vec{x})\ (False : \vec{y}) = (funM\ m_2)\ \vec{x}\ \vec{y}\\
&&& \qquad cond\ \_\qquad\qquad \_\qquad\qquad = 0\\
&&& \quad return\ (Mat\ (1 + a)\ (1 + b)\ cond)\\
&comp\ (Rot\ (\lambda, \lambda')\ (\kappa, \kappa')) && = \textbf{do let}\ rotF\ [True]\ [True] = \kappa'\\
&&& \qquad\qquad\qquad rotF\ [True]\ [False] = \kappa\\
&&& \qquad\qquad\qquad rotF\ [False]\ [True] = \lambda'\\
&&& \qquad\qquad\qquad rotF\ [False]\ [False] = \lambda\\
&&& \qquad\qquad return\ (Mat\ 1\ 1\ rotF)
\end{aligned}
$$

FIGURE 3.8: The *comp* compile function from quantum circuits of type *Circ* into linear operators of type *Mat*

where $\oplus$ is exclusive-or. This can be reformulated as:

$$\widehat{f}\,|x\rangle\,|0\rangle = |x\rangle\,|fx\rangle \tag{3.6}$$

$$\widehat{f}\,|x\rangle\,|1\rangle = |x\rangle\,|1 \oplus (fx)\rangle \tag{3.7}$$

by defining the action of $\widehat{f}$ over the space of the second qubit. Setting the second qubit input into this oracle to $\frac{1}{\sqrt{2}}\,|0\rangle - \frac{1}{\sqrt{2}}\,|1\rangle$, denoted $|-\rangle$, gives:

$$\widehat{f}\,|x\rangle\,|-\rangle = (-1)^{(f\ x)}\,|x\rangle\,(|0\rangle - |1\rangle)$$

since if $f\ x = \mathit{False}$ then $\widehat{f}\,|x\rangle\,|-\rangle = \frac{1}{\sqrt{2}}\,|x\rangle\,(|0\rangle - |1\rangle)$ and if $f\ x = \mathit{True}$ then $\widehat{f}\,|x\rangle\,|-\rangle = \frac{1}{\sqrt{2}}\,|x\rangle\,(|1\rangle - |0\rangle)$.

Following the derivation of the algorithm as given in references [27, 53], the "trick" is to apply the quantum black box function $\widehat{f}$ to qubits prepared in the following state:

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

This can be prepared by applying $H \otimes H$ to the state $|01\rangle$. Applying this to the oracle gives:

$$\begin{aligned}\widehat{f}\,|+\rangle\,|-\rangle &= \frac{\widehat{f}\,|0\rangle\,|-\rangle + \widehat{f}\,|1\rangle\,|-\rangle}{\sqrt{2}} \\ &= \begin{cases} \pm\,|+\rangle\,|-\rangle & \text{if } f\ \mathit{False} = f\ \mathit{True} \\ \pm\,|-\rangle\,|-\rangle & \text{otherwise} \end{cases}\end{aligned}$$

The difference between the two outcomes has now been encoded in the state of the first qubit. If the first qubit is $|+\rangle$ then the function is constant, and if it is $|-\rangle$ then the function is balanced. Applying the Hadamard transform H to the first qubit gives:

$$\pm\,|0\rangle\,|-\rangle \quad \text{if } f\ \mathit{False} \equiv f\ \mathit{True}$$

$$\pm\,|1\rangle\,|-\rangle \quad \text{if } f\ \mathit{False} \not\equiv f\ \mathit{True}$$

as $H\,|+\rangle = |0\rangle$ and $H\,|-\rangle = |1\rangle$.

The algorithm can therefore determine a global property of the function f using only one query of the oracle. A quantum circuit implementation of Deutsch's algorithm is shown in figure 3.9.

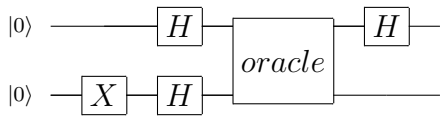


FIGURE 3.9: Quantum circuit implementing the Deutsch algorithm

This circuit follows the convention that the input is always set to $|0\rangle$, and uses the Pauli-X negation operator and two Hadamard transforms to prepare the state $|+\rangle\,|-\rangle$.

The circuit has been translated into the following Haskell circuit implementation, $deutschC$, which takes as an argument another circuit that acts as the oracle:

$$
\begin{aligned}
&deutschC \in Circ \quad \rightarrow Circ \\
&deutschC \quad oracle = (Seq\ (Seq\ (Seq\ (Par\ (Wire\ [0])\ pX) \\
&\qquad\qquad\qquad\qquad\qquad\qquad (Par\ hadC\ hadC)) \\
&\qquad\qquad\qquad\qquad\quad oracle) \\
&\qquad\qquad\qquad\quad (Par\ hadC\ (Wire\ [0])))
\end{aligned}
$$

where $hadC$ is the Hadamard rotation, and pI, pX are the Pauli rotations of identity and negation. The Deutsch algorithm circuit, written as defined in section 3.3, becomes:

$$deutsch = (rot\ H \otimes \mathrm{id}_1)\ \circ\ oracle\ \circ\ (rot\ H \otimes rot\ H)\ \circ\ (\mathrm{id}_1 \otimes rot\ X)$$

where H and X are the Hadamard matrix and negation, and oracle is some predefined circuit. Note the sequential ordering of the Haskell circuit follows that of the diagram in figure 3.9. As the definitions of circuits and their diagrams are equivalent, only the most appropriate notation will be used in future.

There are four different possible definitions of the oracle $\widehat{f}$, and in Haskell these can be written as:

$$
\begin{aligned}
&f_1, f_2, f3, f4 \in Circ \\
&f_1 = cnotC \qquad\qquad\qquad\qquad \text{-- } f\ x = x, \text{ balanced} \\
&f_2 = Seq\ (Seq\ (Par\ pX\ (Wire\ [0])\ cnotC) \\
&\qquad\qquad (Par\ pX\ (Wire\ [0])) \qquad \text{-- } f\ x = \neg\ x, \text{ balanced} \\
&f3 = Par\ (Wire\ [0])\ (Wire\ [0]) \quad \text{-- } f\ x = 0, \text{ constant}
\end{aligned}
$$

$f4 = Par\ (Wire\ [0])\ pX$ -- $f\ x = 1$, constant

A implementation of the Deutsch algorithm in the language QML is presented in chapter 6, with the code given in figure 6.7.

3.10 Quantum teleportation

The quantum teleportation algorithm describes a way to transmit a qubit of information using only two classical bits of data and an EPR pair. The two people (Alice and Bob) who wish to communicate first have to share an EPR pair, with Alice taking one and Bob the other. When Alice later wants to transmit the state of a qubit to Bob she can do this by entangling the qubit she wishes to send with her half of the entangled pair, and then she performs a Hadamard operation on the qubit to be transmitted before measuring her qubit and her half of the EPR pair. The measurements result in obtaining two classical bits of data, and since Alice's half of the EPR pair was entangled with Bob's, this measurement causes an instant change to the state of Bob's half of the EPR pair. Transmitting the classical data to Bob allows him to then perform a defined corrective operation to his qubit, which results in his half of the EPR pair now being equal to Alice's original qubit. The corrective operation Bob has to perform is a NOT if Alice's EPR half collapsed to $|1\rangle$, else nothing, followed by a Pauli-Z operation if Alice's qubit collapses to $|1\rangle$, else nothing. Note that the measurement collapses Alice's qubit, and it is necessary that this measurement be performed in order for Bob to be able to perform the corrective operations to his qubit. Thus this algorithm does not contradict the no-cloning theorem of quantum mechanics.

In any quantum computation involving measurement, the measurement can always be deferred to the final step. Deferring the measurement allows the algorithm to be expressed as the circuit shown in figure 3.10, which assumes a quantum communication channel for the two bits of data.

In figure 3.10 the first qubit is the one Alice wishes to transmit, $|a\rangle$, the second is Alice's half of the EPR pair, epr_a, and the third is Bob's half of the EPR pair, epr_b. At the end of the computation Bob's qubit will contain the original value

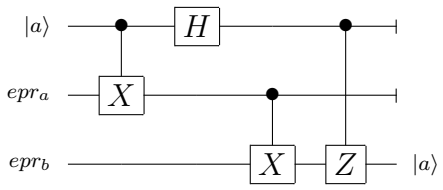


FIGURE 3.10: Quantum circuit implementing the teleportation algorithm

of Alice's qubit. Note that the teleportation algorithm makes use of measurement, which is denoted in the circuit by the top two wires being terminated. Measurement is an irreversible process, and is discussed in section 5.2 which develops irreversible quantum computation.

3.10.1 The principle of deferred measurement

It should be noted that the circuit in figure 3.10 is different to the narrative description of the algorithm, as the description makes use of measurement and a classical communication channel, whereas the circuit is entirely quantum and the measurements occur at the end. It follows from the principle of deferred measurement, which is defined by Nielsen and Chuang [53] as:

> Measurements can always be moved from an intermediate stage of a quantum circuit to the end of the circuit; if the measurement results are used at any stage of the circuit then the classically controlled operations can be replaced by conditional quantum operations.

that these two presentations of the algorithm give the same result. Measurements are an irreversible process, and are hence formalised in section 5.5.2, which defines measurement using the partial trace operation on superoperators. However, having a quantum communication channel between Alice and Bob negates the need for the quantum teleportation protocol entirely; Alice could simply transmit her qubit through the channel.

The teleportation circuit of figure 3.10 can be translated into the *Circ* datatype as shown:

$$
\begin{aligned}
&cpZ \in Circ \\
&cpZ = Cond\ pZ\ (Wire\ [0]) \\
&teleportC \in Circ \\
&teleportC = Seq\ (Seq\ (Seq\ (Seq\ (Seq\ (Par\ cnotC\ (Wire\ [0])) \\
&\qquad\qquad (Par\ hadC\ \ (Wire\ [0,1])))) \\
&\qquad\qquad (Par\ (Wire\ [0])\ cnotC)) \\
&\qquad\qquad (Wire\ [1,0,2])) \\
&\qquad\qquad (Par\ (Wire\ [0])\ cpZ)) \\
&\qquad\qquad (Wire\ [1,0,2])
\end{aligned}
$$

where cpZ defines the circuit for a controlled-Pauli-Z operation. Note that in order for the control to be applied to the correct qubits they first need to be permuted.

The implementation of the quantum teleport algorithm as a quantum circuit shows some of the failings of this model for expressing quantum algorithms, as it requires some narration in order for the algorithm to be understood and performed correctly. A full discussion of quantum teleportation can be found in section 1.3.7 of reference [53], and an implementation of this algorithm in the language QML is included in chapter 6, with the code presented in figure 6.6.2.

3.11 Other models of quantum computation

The quantum circuit model is not the only model of quantum computation. Another model is the "one way quantum computer," introduced by Raussendorf, Browne and Briegel [59, 60] to address concerns about the scalability of quantum computational systems. The one way model works by only performing a sequence of one-qubit measurements on the cluster state, which is an Pauli-Z entangled multi-qubit state. The model is "one way" as the entanglement in a cluster state is destroyed by the one-qubit measurements, and therefore the cluster state can be used only once. The cluster state is a resource for quantum computation. In the one way quantum computer, the order and choices of measurements determine the algorithm computed.

It has been shown that any quantum circuit can be represented in the one way model [59]; however, not all quantum information processing methods available in the

one way model are available in the circuit model [60]. It is more expressive of how the computation can take place, not requiring as much narration as the circuit model (for example, see quantum teleportation above).

As every step in the one way model makes use of measurement, it seems at present that this model may be closer to a realisable model of quantum computing. In the quantum circuit model the quantum state has to be protected from decoherence all the way through the computation, which is a major difficulty in the fabrication of actual quantum computers. In this sense, decoherence means the unwanted interaction of the quantum system with the environment, in such a thermodynamically irreversible way that ensures different elements in the wavefunction of both the superposition of the system and the environment can no longer interfere with each other. The one way model, however, only has to be free from decoherence for each step of the computation, after which a measurement takes place. There has recently been much work in moving away from the circuit model as the realisable model, to measurement-based one-way quantum computers, such as that in reference [85].

3.12 Further reading

Detailed introductions to the theory of quantum computation can be found in several plenary texts: *cf.* [15, 36, 53, 58]. Nielsen and Chuang's text [53] is currently regarded as the definitive general publication on this subject. It is, according to the authors, written assuming the reader has an understanding of mathematics, physics, or computer science. Complimentary reading of a linear algebra text is recommended for those unfamiliar with the mathematical techniques required. Hirvensalo's book [36] begins in an accessible manner for students of computer science or mathematics, but assumes the reader to be very familiar with some advanced mathematical techniques. However, the book does contain two very large appendices, taking almost a third of the book, describing these mathematical techniques, and a compact history and explanation of quantum physics. Brown's book [15] is a popular–science history of quantum computing, and gives a good overview of the field and how quantum computers work, both theoretically and practically, and the gains they may give over

classical computers. Finally, [58] is a set of lecture notes from university courses on quantum computation, and is comprehensible and accessible, with Preskill's work being highly regarded in the field.

More information on the alternative one way quantum computer model of quantum computations can be found in references [59, 60, 85].

3.13 Summary

This chapter serves as an introduction to quantum computation. It introduces the concept of a qubit and qubit registers, and how these can be physically realised. The focus of this chapter is the development of a circuit model of quantum computation, which follows the development of the classical model in chapter 2. An implementation of the circuit model in Haskell is also provided, with a compiler and evaluator. Deutsch's algorithm – the prototypical quantum algorithm – is also discussed and realised in the quantum circuit model developed in this chapter. Analogies between quantum and reversible classical computation are also discussed.

Chapter 4

FCC: Reversible & irreversible classical computation

This chapter presents a categorical interpretation of classical reversible and irreversible computations, as developed in chapter 2. The notion of a category is explained, and the development of **FCC**, the category of Finite Classical Computations, is given. The purpose of this development is expand the development of **FCC** given here, to the analogous development of the category **FQC**, in chapter 5, which is used to express the operational semantics of QML.

This chapter also gives a mathematical interpretation of the category **FCC** as finite sets, which is a formal interpretation of the classical circuit compiler presented in section 2.5. The mapping from **FCC** to **FinSet** is also shown to be functorial.

4.1 Finite Classical Computations

It is often stated that quantum computations have to be reversible due to the underlying unitary nature of quantum mechanics, which is a model of reversible processes. However, this is also true for the classical case. Landauer's principle, discussed in chapter 2, sets no lower bound on energy dissipation in classical reversible computations, as no information need ever be erased [10], and the current understanding of the classical physical laws is that they are inherently reversible, within certain boundary conditions (see chapter 2).

Given maximal knowledge of the terminal state of any closed physical process, the laws of physics can be applied in reverse to calculate the initial state. This means that even in the classical case the physical processes of ideal computation are reversible. So, in order to explain *irreversible* computation classically, it should be embedded in a *reversible* framework of computation. In this chapter a categorical notion of what a computation is classically will be given, based on a reversible model of finite computations, resulting in the category of Finite Classical Computations (**FCC**). Indeed, the category **FCC** may be seen as a categorical interpretation of Bennet's work [10] on reversible computation.

A category of Finite Quantum Computations (**FQC**) will be developed in chapter 5 by analogy to the development of **FCC** here, allowing both the differences and similarities of the two computational models to be highlighted. The analysis of these two categories guides the design of QML, aiming to realise structures common to both computational paradigms by syntactic constructs established in classical functional programming.

4.2 Rudimentary category theory

There are many advanced concepts available in category theory, but these are mostly unnecessary for the analysis here. Further information on category theory can be found in references [47, 56].

A category, $\mathcal{C}$, is a collection, or class, of objects $(a, b, c, ...)$, with a collection of unique morphisms, also called arrows, between them. For any two morphisms $f \in a \to b, g \in b \to c$, there exists a unique composition morphism, $g \circ f \in a \to c$, which is associative. There is also the additional constraint that a distinguished identity morphism must exist for every object.

A monoidal category is a category, $\mathcal{C}$, as above, equipped with a binary functor $\otimes \in \mathcal{C} \times \mathcal{C} \to \mathcal{C}$, called a tensor, with a unit object I. A monoidal category must have three natural isomorphisms, which express the fact that the tensor operation must be associative, have a left and right identity. Associativity is given by:

$$\alpha_{a,b,c} \in (a \otimes b) \otimes c \to a \otimes (b \otimes c)$$

the left-identity by:

$$\lambda_a \in I \otimes a \to a$$

and the right identity by:

$$\rho_a \in a \otimes I \to a$$

Additionally, these three natural transformations are subject to certain coherence conditions, which are given by Mac Lane [47].

A strict monoidal category is a monoidal category in which the three natural transformations α, λ, and ρ are all the identity transformation.

Finally, a groupoid is a category in which every morphism is an isomorphism, *i.e.* there exists an inverse for every morphism, such that the composition of a morphism and its inverse gives the identity morphism.

4.3 Reversible computations in $\mathbf{FCC}^{\simeq}$

In this section the definitions of reversible circuits, given in section 2.2, will be reformulated in the category $\mathbf{FCC}^{\simeq}$; the category of *reversible* finite classical computations. For full details on each construction and the notation used, refer to section 2.2. The purpose of this reformulation is to make precise the informal construction of reversible computations given in section 2.2.

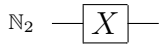
Reversible computations are modelled here as a category, where for every morphism $\phi \in \mathbf{FCC}^{\simeq}\ a\ b$ there is an inverse $\phi^{-1} \in \mathbf{FCC}^{\simeq}\ b\ a$, such that ϕ and ϕ^{-1} are isomorphisms, and a and b are finite sets. The morphisms represent computations, and the requirement for the existence of an inverse computation, such that there is an isomorphism, ensures the computation is reversible. As every morphism in $\mathbf{FCC}^{\simeq}$ is an isomorphism, it follows that $\mathbf{FCC}^{\simeq}$ is in fact a groupoid. Any isomorphic objects are assumed to be equal, *i.e.* $\mathbf{FCC}^{\simeq}$ is strict. It follows from this that $\mathbf{FCC}^{\simeq}\ a\ b = \{\,\}$ if $a \neq b$, and consequently homsets of $\mathbf{FCC}^{\simeq}$ are denoted $\mathbf{FCC}^{\simeq}\ a = \mathbf{FCC}^{\simeq}\ a\ a$; the source and target bit-vectors must be of the same size (have the same number of wires) for the computation to be reversible. $\mathbf{FCC}^{\simeq}$ has a strict monoidal structure $(I, \otimes)$, where $I = 0$ and $a \otimes b = a + b$. A special object of Booleans is defined as $\mathbb{N}_2$,

with $\mathbb{N}_2 = 1$; the monoid of addition lifts to a strict monoidal structure on **FCC**.

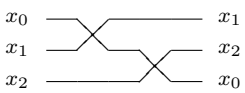
As computations take place on bit-vectors, which are collections of Booleans, only objects generated from $(I = 0, \mathbb{N}_2 = 1, \otimes = +)$ are interesting; if $\mathbb{N}_2$ represents a wire, then $\mathbb{N}_2 \otimes \mathbb{N}_2$ is two wires, etc. Hence natural numbers $a \in \mathbb{N}$ can be used to denote the object $\mathbb{N}_2^a$. This gives $I = 0$, $\mathbb{N}_2 = 1$, and $a \otimes b = a + b$, as stated previously. The objects of the category $\mathbf{FCC}^{\simeq}$ are therefore the initial segment of a, as defined previously, $[a] = \{i \in \mathbb{N} \mid i < a\}$. Note that $\mathbf{FCC}^{\simeq}$ is the free symmetric monoidal category on one object: $\mathbb{N}_2$.

The morphisms of the category $\mathbf{FCC}^{\simeq}$ a are the circuits of arity a, defined in section 2.2, which can be characterised inductively:

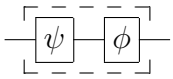
negation $X \in \mathbf{FCC}^{\simeq}$ 1 gives the only non-trivial one bit classical operation, where X is the NOT function:



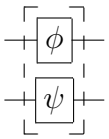
wires *wires* $\phi \in \mathbf{FCC}^{\simeq}$ a where $\phi : [a] \simeq [a]$ is a bijection. This represents any rewiring, including the identity $id_a = wires\, id$. For example, the bijection $\phi(0) = 2$, $\phi(1) = 0$, and $\phi(2) = 1$ gives the following $\mathbf{FCC}^{\simeq}$ 3 morphism:



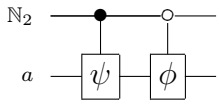
sequential composition Given $\phi \in \mathbf{FCC}^{\simeq}$ a and $\psi \in \mathbf{FCC}^{\simeq}$ a, the composition $\phi \circ \psi \in \mathbf{FCC}^{\simeq}$ a can be constructed:



parallel composition Given $\phi \in \mathbf{FCC}^{\simeq}$ a and $\psi \in \mathbf{FCC}^{\simeq}$ b, the parallel composition $\phi \otimes \psi : \mathbf{FCC}^{\simeq}$ $(a \otimes b)$ can be constructed:



conditional Given $\phi, \psi \in \mathbf{FCC}^{\simeq}\ a$, the conditional operation $\phi|\psi \in \mathbf{FCC}^{\simeq}\ (1 \otimes a)$ can be constructed:



The inverse of each construction is calculated in exactly the same way as for classical reversible circuits, as described in section 2.2.

The requirement that $(\mathbf{FCC}^{\simeq}, I, \circ, 1, \otimes)$ is a strict monoidal category allows the circuit diagrams, which are the morphisms, to be unambiguously interpreted. In Haskell, the morphisms of $\mathbf{FCC}^{\simeq}$ are given exactly by the *Circ* datatype presented in section 2.3.

4.4 Irreversible computations in FCC

In this section a notion of irreversible computations will be derived from the notion of reversible computations given in the previous section, to give the category of all possible finite classical computations; denoted **FCC**.

The category **FCC** describes classical operations as bijections (and are hence reversible) on the elements of finite sets. The objects of the category are therefore the finite sets, as in $\mathbf{FCC}^{\simeq}$. Every morphism in the category **FCC** represents an irreversible computation, but is in fact of the form $\phi = (h, g, \phi')$, with $h, g \in \mathbb{N}$ representing the number of ancillary *heap* inputs and *garbage* outputs required to compute the underlying reversible computation ϕ'. A morphism $(h, g, \phi) \in \mathbf{FCC}\ a\ b$ can be transposed into the category $\mathbf{FCC}^{\simeq}$ as $\phi \in \mathbf{FCC}^{\simeq}\ (a \otimes h)\ (b \otimes g)$, with the requirement that $a \otimes h = b \otimes g$.

An irreversible computation $(h, g, \phi) \in \mathbf{FCC}\ a\ b$ can be visualised diagrammatically as the reversible computation ϕ, where heap and garbage have been explicitly labelled ($\vdash$ and $\dashv$, respectively) as shown in figure 4.1.

Using the heap and garbage, any irreversible computation $\phi \in \mathbf{FCC}\ a\ b$ can be interpreted as a reversible computation $\phi' \in \mathbf{FCC}^{\simeq}\ (a \otimes b)\ (b \otimes a)$, by $\phi'(x, 0^b) = (\phi x, x)$ where 0^b denotes a heap register of length b initialised to 0. ϕ' takes each $(x, 0^b)$ to a distinct output, and is therefore a reversible function on the finite set

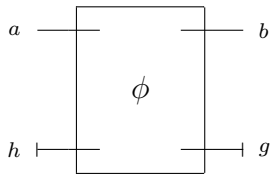


FIGURE 4.1: Visualisation of a computation in **FCC**, where ϕ is a reversible $\mathbf{FCC}^{\simeq}$ morphism, and heap and garbage are explicit.

$a \otimes b$, which requires $a + b$ bits to be computed. Therefore, the maximum amount of heap required to make a function reversible is bounded by the size of the output of the irreversible function: $h \leqslant b$; and the maximum garbage required is bounded by the size of the input to the irreversible function: $g \leqslant a$. The values given for heap and garbage by this calculation are the maximum required, and are thus often sub-optimal. For example, making the function $f\ x = (x, x)$ reversible, where $a = 1$ and $b = 2$, would give a function with $h = b = 2$ and $g = a = 1$. However, f can be implemented using only a single bit of heap and no garbage; it is simply the CNOT operation.

It is also true that for any computation $\phi \in \mathbf{FCC}^{\simeq}\ a$ there is an equivalent computation $\widehat{\phi} \in \mathbf{FCC}\ a\ a$. This is given by the rule:

$$\frac{\phi \in \mathbf{FCC}^{\simeq}\ a}{\widehat{\phi} \in \mathbf{FCC}\ a\ a} \tag{4.1}$$

where $\widehat{\phi} = (I, I, \phi)$, with $I = 0$; there is no heap or garbage.

In summary, given finite sets $a, b \in \mathbb{N}$, a finite classical computation $\phi \in \mathbf{FCC}\ a\ b$ is given by a triple $\phi = (h, g, \phi')$ where

$h \in \mathbb{N}$ is the size of the initial heap, initialised to 0

$g \in \mathbb{N}$ is the size of the garbage, to be disposed of at the end of the computation

$\phi' \in \mathbf{FCC}^{\simeq}\ c$ is a reversible computation, with $c = a \otimes h = b \otimes g$

It should be noted that this is different from the presentation of **FCC** in reference [3]. Only finite sets of powers of 2 are considered (bit-vectors) in this development

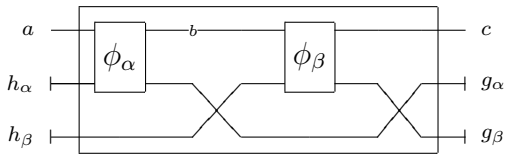


FIGURE 4.2: Sequential composition in **FCC**: $\phi_{\beta\circ\alpha}$

of **FCC**, and the heap is always initialised as a vector of 0s, so no extra information about the heap is required.

In order to complete the category **FCC**, definitions of identity and composition are required. Given $a \in \mathbb{N}$, the identity morphism can be defined as $id_a = (I, I, id_a)$. Note that this is simply the lifting from $\mathbf{FCC}^{\simeq}$ of the reversible identity morphism. Two computational systems can be composed by combining the heap and garbage to each computation. The composition of computations $\alpha = (h_\alpha, g_\alpha, \phi_\alpha) \in \mathbf{FCC}\ a\ b$ and $\beta = (h_\beta, g_\beta, \phi_\beta) \in \mathbf{FCC}\ b\ c$ is $\beta \circ \alpha = (h, g, \phi) \in \mathbf{FCC}\ a\ c$ where

$$\begin{aligned} h &= h_\alpha \otimes h_\beta \\ g &= g_\alpha \otimes g_\beta \\ \phi_{\beta\circ\alpha} &= (id_c \otimes S) \circ (\phi_\beta \otimes id_{g_\alpha}) \circ (id_b \otimes S) \circ (\phi_\alpha \otimes id_{h_\beta}) \end{aligned}$$

$S \in a \otimes b \simeq b \otimes a$ is a trivial symmetric swap operation. Diagrammatically, this construction is shown in figure 4.2. To simplify the presentation, the isomorphic swap operations can be omitted from the definition of $\phi_{\beta\circ\alpha}$ to give:

$$\begin{aligned} h &= h_\alpha \otimes h_\beta \\ g &= g_\alpha \otimes g_\beta \\ \phi_{\beta\circ\alpha} &= (\phi_\beta \otimes id_{g_\alpha}) \circ (\phi_\alpha \otimes id_{h_\beta}) \end{aligned}$$

This completes the definition of **FCC** as a category. In addition to the standard notion of sequential composition, **FCC** inherits the monoidal structure of $\mathbf{FCC}^{\simeq}$, and a parallel composition can be defined, in much the same way as the sequential composition. For $\alpha \in \mathbf{FCC}\ a\ b$ and $\beta \in \mathbf{FCC}\ c\ d$, their parallel composition is

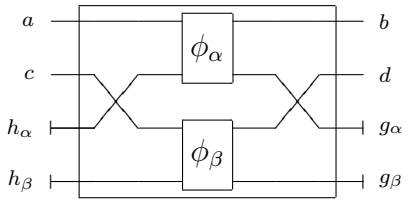


FIGURE 4.3: Parallel composition in **FCC**: $\phi_{\alpha\otimes\beta}$

defined as $\alpha \otimes \beta = (h, g, \phi) \in \mathbf{FCC}\ (a \otimes c)\ (b \otimes d)$ where

$$\begin{aligned} h &= h_\alpha \otimes h_\beta \\ g &= g_\alpha \otimes g_\beta \\ \phi_{\alpha\otimes\beta} &= \phi_\alpha \otimes \phi_\beta \end{aligned}$$

again omitting monoidal isomorphisms. These are shown in the diagrammatic representation of parallel composition, which is given in figure 4.3.

The neutral element of parallel composition, the Cartesian product on **FCC**, can be obtained by lifting $I_{\mathbf{FCC}^\simeq}$ to $I_{\mathbf{FCC}}$.

4.5 Strict reversible computations

Another category of computations is that of *strict* computations $\alpha = (h, \phi) \in \mathbf{FCC}^\circ\ a\ b$. $\mathbf{FCC}^\circ$ computations can be defined as computations where there is no garbage, $g = 0$, hence it is omitted. It is a monoidal subcategory of **FCC**, and is of no particular interest in the classical case, but is important in the development of **FQC**.

From the definitions of $\mathbf{FCC}^\simeq$ (no heap or garbage), $\mathbf{FCC}^\circ$ (no garbage but allows heap) and **FCC** (allows heap and garbage), it follows that:

$$\mathbf{FCC}^\simeq \subset \mathbf{FCC}^\circ \subset \mathbf{FCC}$$

4.6 Modelling reversible classical computations in FinSet

The compiler function *comp* described in section 2.5.2 takes as input a classical reversible circuit, and outputs a bijection $f \in [\mathbb{N}_2] \to [\mathbb{N}_2]$. The translation from

morphisms in the category $\mathbf{FCC}^{\simeq}$ into bijections on bit-vectors will be formalised in this section by defining a translation function that takes objects and morphisms from $\mathbf{FCC}^{\simeq}$ into the category **FinSet** of finite sets, which will then be shown to be functorial.

The objects of the category of finite sets **FinSet** are identical to the objects of $\mathbf{FCC}^{\simeq}$ defined in section 4.3, as $\mathbf{FCC}^{\simeq}$ objects are finite sets. Finite sets are again given by natural numbers, $a \in \mathbb{N}$, which are identified with their initial segment, $[a]$. The homsets of the category are given by $\mathbf{FinSet}\ a\ b = a \to b$, and are functions between finite sets.

A functor, F, is a mapping between categories that assigns to all objects or morphisms of the source category, A, an object or morphism of the target category, B, such that certain structural properties are preserved. Given a functor $F \in A \to B$, the conditions that must be met are:

> domains and codomains are preserved: given $f \in A \to B$, then $Ff \in FA \to FB$.
>
> identities are preserved: $\forall a \in A. F(I_A) = I_{FA}$.
>
> composition is preserved: if $f \circ g$ is computable in A, then $F\ (f \circ g) = Ff \circ Fg$, where the second composition is formed in B.

All circuits $\phi \in \mathbf{FCC}^{\simeq}\ a$ can be interpreted as $[\![\phi]\!] \in \mathbf{FinSet}\ a\ a$, by induction over the inductive definition of $\mathbf{FCC}^{\simeq}$ given in section 4.3. Because the morphisms of $\mathbf{FCC}^{\simeq}$ are reversible, they correspond to bijections in the category **FinSet**, hence the domain and codomain of the **FinSet** morphisms are the same as when interpreting $\mathbf{FCC}^{\simeq}$ morphisms. The translation from $\phi \in \mathbf{FCC}^{\simeq}\ a$ to $[\![\phi]\!] \in \mathbf{FinSet}\ a\ a$ is defined in this way as:

negation $[\![X]\!] = \neg$

wires $[\![\mathit{wires}\ \phi]\!] = f$ where $f\ a\ b = \mathbf{if}\ \phi\ a \equiv b\ \mathbf{then}\ 1\ \mathbf{else}\ 0$.

sequential composition $[\![\phi \circ \psi]\!] = [\![\phi]\!] \circ [\![\psi]\!]$

parallel composition $[\![\phi \otimes \psi]\!] = [\![\phi]\!] + [\![\psi]\!]$

conditional $[\![\phi \mid \psi]\!] = [\![\phi]\!] | [\![\psi]\!]$ where

$$\begin{aligned}
\phi|\psi\ (0, a)\ (0, b) &= \phi\ a\ b \\
\phi|\psi\ (1, a)\ (1, b) &= \psi\ a\ b \\
\phi|\psi\ _\quad\ _\quad &= 0
\end{aligned}$$

Note that this functor corresponds to the Haskell *comp* function, which translated circuits into matrices, given in section 2.5.2.

4.7 Modelling irreversible computations in FinSet

The interpretation of $\mathbf{FCC}^{\simeq}$ objects and morphisms in the category **FinSet**, given in the previous section, will now be extended to an interpretation of the category **FCC** of irreversible finite computations. The objects of **FCC** are the same as those of $\mathbf{FCC}^{\simeq}$ and are thus interpreted in the same way. A morphism $\phi \in \mathbf{FCC}\ a\ b$ is actually given by a triple, $\phi = (h, g, \phi')$, where $h, g \in \mathbb{N}$ give the sizes of the heap and garbage required to compute the reversible circuit $\phi' \in \mathbf{FCC}^{\simeq}$. By defining an injection on finite sets that initialises the heap, 0^h,and using the projection operation π to discard garbage at the end of a computation, an interpretation of $\phi \in \mathbf{FCC}\ a\ b$ can be defined. The interpretation of **FCC** is given by interpreting morphisms as functions on finite sets, thus $(h, g, \phi) \in \mathbf{FCC}\ a\ b$ is interpreted as :

$$\pi_g \circ [\![\phi]\!] \circ (-, 0^h) \in [a] \to [b]$$

where $[\![\phi]\!] \in [a \otimes h] \to [b \otimes g]$ is the associated permutation on finite sets, $(-, 0^h) \in [a] \to [a \otimes h]$ initialises the heap to *False*, and $\pi_g \in [b \otimes g] \to b$ projects away the garbage. The "morphism as functions" interpretation can be shown as the following commuting square:

$$\begin{array}{ccc}
a \otimes h & \xrightarrow{[\![\phi]\!]} & b \otimes g \\
{\scriptstyle (-,0^h)}\uparrow & & \downarrow{\scriptstyle \pi_g} \\
a & \xrightarrow[f]{} & b
\end{array}$$

4.7.1 Extensional equality of **FCC**

Two morphisms f, g are extensionally equivalent if

$$\forall x. fx = gx$$

where x is in the domain of f and g. Extensional equality for **FCC** is given by interpreting morphisms as functions on finite sets, using the functor $[\![\cdot]\!] \in \mathbf{FCC} \to \mathbf{FinSet}$ described in section 4.7.

4.7.2 $\mathbf{FCC} \to \mathbf{FinSet}$ is functorial

The interpretation of **FCC** into **FinSet** via $[\![\cdot]\!]$ given in the previous section fulfils the definition of a functor. This follows from the fact that the identities, domains, and composition are all preserved by the mapping $[\![\cdot]\!] \in \mathbf{FCC}\ a\ b \to \mathbf{FinSet}\ a\ b$, which is shown by the following proposition:

Proposition 1. $[\![\cdot]\!] \in \mathbf{FCC} \to \mathbf{FinSet}$ is a strict monoidal functor, that is:

$$[\![\mathrm{id}]\!] = \mathrm{id}$$

$$[\![f \circ g]\!] = [\![f]\!] \circ [\![g]\!]$$

$$[\![f \otimes g]\!] = [\![f]\!] + [\![g]\!]$$

Proof. Both 1. and 3. follow directly from monoidal identities. The only interesting case is 2. which follows from the fact that the diagram shown in figure 4.4 commutes. □

4.8 FCC in Haskell

It is a straightforward task to implement **FCC** morphisms in Haskell. A morphism is defined as $(h, g, \phi) \in \mathbf{FCC}\ a\ b$ where $a, b, h, g \in \mathbb{N}$ and $\phi \in \mathbf{FCC}^{\simeq}$. a and b give the sizes of the input and the output to the morphism, and h and g give the sizes of any heap and garbage required to compute ϕ. It is shown in section 4.3 that $\mathbf{FCC}^{\simeq}$ morphisms are represented in Haskell by the classical *Circ* datatype presented

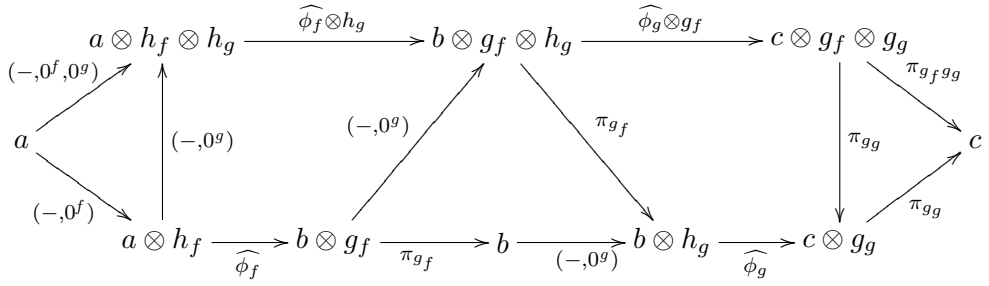


FIGURE 4.4: Composition is preserved by lifting **FCC** to **FinSet**. The path along the top of the diagram shows $[\![f \circ g]\!]$, while the lower path shows $[\![f]\!] \circ [\![g]\!]$. This diagram commutes.

in section 2.3. An **FCC** morphism can be encapsulated as a Haskell record datatype with named fields as:

$$\textbf{data FCC} = \textbf{FCC}\{\, a, b, h, g \in \mathit{Int}, \phi \in \mathit{Circ}\,\}$$

There are some requirements on this datatype which cannot be enforced by Haskell, for example that $\phi \in \mathit{Circ}$ is a valid reversible circuit, so instead a function can be written which validates a Haskell **FCC** object. This function, *validFCC*, makes use of the classical circuit *arity* function from section 2.3, and returns *Nothing* if the morphism is invalid:

$$\begin{aligned}
&\mathit{validFCC} \in \textbf{FCC} \to \mathit{Maybe}\ \textbf{FCC}\\
&\mathit{validFCC}\ f = \textbf{do let}\ \mathit{ah} = \mathit{Just}\ (a\ f + h\ f)\\
&\qquad\qquad\qquad\qquad\quad \mathit{bg} = \mathit{Just}\ (b\ f + g\ f)\\
&\qquad\qquad\qquad\quad \mathit{guard}\ (\mathit{arity}\ (\phi\ f) \equiv \mathit{ah} \wedge (\mathit{ah} \equiv \mathit{bg}))\\
&\qquad\qquad\qquad\quad \mathit{return}\ f
\end{aligned}$$

This function ensures the validity of the morphism by checking that $a \otimes h = b \otimes g$, and that the arity of the morphism agrees with $a \otimes h$.

There is no need to implement $\textbf{FCC}^{\simeq}$ and $\textbf{FCC}^{\circ}$ separately, as $\textbf{FCC}^{\simeq}$ morphisms are those which have $h, g = 0$, and $\textbf{FCC}^{\circ}$ morphism are those which have $g = 0$.

An implementation of **FinSet**, and the $[\![\cdot]\!]$ functor, has already been presented in Haskell. In section 2.5 the types *Vec* and *Mat* model **FinSet** objects and the functions between them as vectors and matrices, and the function $\mathit{comp} \in \mathit{Circ} \to \mathit{Mat}$ directly implements the functor $[\![\cdot]\!] \in \textbf{FCC}^{\simeq}\ a \to \textbf{FinSet}\ a\ a$.

4.9 Summary

The category **FCC** of Finite Classical Computations has been presented, with the development proceeding via the category $\mathbf{FCC}^{\simeq}$ of reversible classical computations, following the presentation of reversible computation given in 2. The category $\mathbf{FCC}^{\circ}$ of strict computations has also been introduced, as well as a mathematical interpretation of $\mathbf{FCC}^{\simeq}$ and **FCC** in the category **FinSet**. An implementation of **FCC** morphisms has been given in Haskell. It is also shown in this chapter that all irreversible classical computations can be implemented in a reversible setting, by the definition of **FCC** morphism (irreversible computations) as a $\mathbf{FCC}^{\simeq}$ morphism (reversible computations), with extra heap and garbage.

Chapter 5

FQC: Reversible & irreversible quantum computation

In this chapter three different categories of finite quantum computation will be defined. First, the groupoid $\mathbf{FQC}^{\simeq}$ of reversible quantum computations will be defined, following the definition of $\mathbf{FCC}^{\simeq}$ in section 4.3. From the definition of $\mathbf{FQC}^{\simeq}$ a definition for the category of irreversible quantum computations, **FQC**, will be derived, in the same way that **FCC** was derived from $\mathbf{FCC}^{\simeq}$ in section 4.4. A final category of strict, or pure, finite quantum computations, $\mathbf{FQC}^{\circ}$, will then be defined as a subcategory of **FQC**. The categories **FQC** and $\mathbf{FQC}^{\circ}$ form the basis of the operational semantics for the language QML. The purpose of the parallel developments is to highlight where quantum computing differs from classical computing, and, furthermore, to highlight the similarities between the two cases.

This chapter also introduces the mathematical categories of unitary operations, $\mathbf{Q}^{\simeq}$; isometric operations, $\mathbf{Q}^{\circ}$; and superoperators, **Q**; which act as mathematical interpretations of $\mathbf{FQC}^{\simeq}$, $\mathbf{FQC}^{\circ}$ and **FQC**, respectively. **Q** is a very important category that will form the basis of the denotational semantics of QML, discussed in section 7.5 and also chapter 8. A definition of extensional equality for **FQC** is also presented, which makes use of these mathematical formalisms, and implementations of all six categories are provided in Haskell.

5.1 Reversible quantum computations in $\mathbf{FQC}^{\simeq}$

As section 4.3 reformulates the definition of reversible circuits into the category $\mathbf{FCC}^{\simeq}$ of reversible finite classical computations, so this section reformulates the definition of quantum circuits given in section 3.3 into the category $\mathbf{FQC}^{\simeq}$. The reformulation is again done to make the informal definition of quantum computations, given in section 3.3, precise, which will provide the basis for an operational semantics of QML. The developments presented in this chapter also provide a formal definition of the standard quantum circuit model, which is a contribution of this thesis.

In the classical groupoid $\mathbf{FCC}^{\simeq}$, reversible computations are modelled as morphisms $\phi \in \mathbf{FCC}^{\simeq}\ a\ b$ such that an inverse $\phi^{-1} \in \mathbf{FCC}^{\simeq}\ b\ a$ exists, and so that together they form an isomorphism. The objects a, b of $\mathbf{FCC}^{\simeq}$ are finite sets, and ϕ defines a bijection between finite sets. The reversible quantum computations introduced in section 3.3 are formalised in the category $\mathbf{FQC}^{\simeq}$, in which reversible quantum computations are modelled as morphisms $\phi \in \mathbf{FQC}^{\simeq}\ a\ b$, with an inverse $\phi^{-1} \in \mathbf{FQC}^{\simeq}\ b\ a$. In the quantum case, $\mathbf{FQC}^{\simeq}$ objects are again finite sets, and ϕ is a unitary quantum operation between the Hilbert spaces generated by those sets, by taking the finite sets to be the basis of the Hilbert space. The inverse can be found by taking the adjoint of the unitary operation ϕ: $\phi^{\dagger} = \phi^{-1}$. Again, ϕ, ϕ^{-1} together give an isomorphism. For details of these concepts, see section 3.2.

As in the classical reversible case, any isomorphic objects are assumed to be equal, *i.e.* $\mathbf{FQC}^{\simeq}$ is strict, and as before it follows from this that $\mathbf{FQC}^{\simeq}\ a\ b = \{\,\}$ if $a \neq b$. Thus, as with $\mathbf{FCC}^{\simeq}$, the homsets of $\mathbf{FQC}^{\simeq}$ are denoted $\mathbf{FQC}^{\simeq}\ a = \mathbf{FQC}^{\simeq}\ a\ a$.

$\mathbf{FQC}^{\simeq}$ also has a strict monoidal structure $(I, \otimes)$, which in the quantum case is interpreted as the tensor product $\otimes$ on vectors, rather than the Cartesian product $\times$ on finite sets used in $\mathbf{FCC}^{\simeq}$. As in $\mathbf{FCC}^{\simeq}$, the natural numbers $a \in \mathbb{N}$ can be used to denote the object $\mathcal{Q}_2^a$, giving in this case $I = 0$, $\mathcal{Q}_2 = 1$, and $a \otimes b = a + b$, where $\mathcal{Q}_\mathbf{2}$ represents quantum bits. Quantum computations take place on qubit registers, analogous to classical registers, which in this case are objects generated from $(I, \mathcal{Q}_2, \otimes)$. The tensor product can be though of as adding wires to the quantum circuit, as the Cartesian product adds classical wires in $\mathbf{FCC}^{\simeq}$. The definition of the

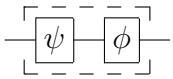
initial segment of a is unchanged: $[a] = \{i \in \mathbb{N} \mid i < a\}$.

As morphisms in $\mathbf{FCC}^{\simeq}$ are given inductively in section 4.3, morphisms in $\mathbf{FQC}^{\simeq}$ a, which are the quantum circuits of arity a, can also be characterised inductively (with the details of each construction given in section 3.3) as follows:

rotation $\mathit{rot}\ u \in \mathbf{FQC}^{\simeq}\ 1$ denotes a rotation that acts on a single qubit, where u defines a unitary operation. The inverse operation is given by the adjoint of the unitary operation, $\phi^{-1} = \mathit{rot}\ u^{\dagger} \in \mathbf{FQC}^{\simeq}\ 1$. In $\mathbf{FCC}^{\simeq}$ there is only the negation operation available on one bit.

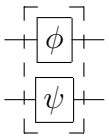
wires $\mathit{wires}\ \phi \in \mathbf{FQC}^{\simeq}\ a$ where $\phi : [a] \simeq [a]$ is a bijection, which can represent any reordering of qubits, including the identity $id_a = \mathit{wires}\, id$. Rewirings are trivially reversible, and this is the same as in the case of $\mathbf{FCC}^{\simeq}$.

sequential composition Given $\phi \in \mathbf{FQC}^{\simeq}\ a$ and $\psi \in \mathbf{FQC}^{\simeq}\ a$, the composition $\phi \circ \psi \in \mathbf{FQC}^{\simeq}\ a$ can be constructed.



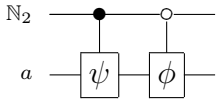
This is exactly the same construction as for the category $\mathbf{FCC}^{\simeq}$.

parallel composition Given $\phi \in \mathbf{FQC}^{\simeq}\ a$ and $\psi \in \mathbf{FQC}^{\simeq}\ b$, the parallel composition $\phi \otimes \psi : \mathbf{FQC}^{\simeq}\ (a \otimes b)$ can be constructed.



This is similar to the parallel construction from $\mathbf{FCC}^{\simeq}$, except here the tensor product is used, as explained in section 3.3.

conditional Given $\phi, \psi \in \mathbf{FQC}^{\simeq}\ a$, the conditional operation $\phi|\psi \in \mathbf{FQC}^{\simeq}\ (1 \otimes a)$ can be constructed.



which is again the same construction as in the definition of the category $\mathbf{FCC}^{\simeq}$.

As with $\mathbf{FCC}^{\simeq}$, in order to allow the unambiguous interpretation of circuits, it is required that $(\mathbf{FQC}^{\simeq}, \mathrm{id}, \circ, I, \otimes)$ is a strict monoidal category. The morphisms of $\mathbf{FQC}^{\simeq}$, as defined above, are given in Haskell exactly by the quantum *Circ* datatype, defined in section 3.5.

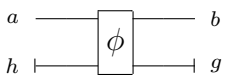
The development of $\mathbf{FQC}^{\simeq}$ above for reversible quantum computation is almost identical to the development of $\mathbf{FCC}^{\simeq}$ presented in section 4.3. The main differences are that morphisms are now unitary maps between Hilbert spaces, although the objects are still finite sets; parallel composition is modelled by the tensor product, although this is still the Cartesian product on the underlying finite sets; and while in the classical case there is only the negation operation on one bit, in the quantum case any unitary operation on one qubit is possible.

5.2 Irreversible quantum computations in FQC

A notion of irreversible quantum computations will be developed in this section, in the same way for the classical case in section 4.4. A category **FQC** of all finite quantum computations will be derived from the category $\mathbf{FQC}^{\simeq}$, in the same way that **FCC** was derived from $\mathbf{FCC}^{\simeq}$.

Objects of this category are the same as for the category $\mathbf{FQC}^{\simeq}$, *i.e.* the finite sets that represent the computational basis states, and morphisms are quantum computations, which now may include *heap* and *garbage*. As in the definition of **FCC**, every morphism in **FQC** is written as a triple $\phi = (h, g, \phi')$, where $h, g \in \mathbb{N}$ again represent ancillary heap inputs and garbage outputs that are required to compute the reversible operation ϕ', which is again given as a circuit. As in the classical case, a morphism $(h, g, \phi) \in \mathbf{FQC}\ a\ b$ can be embedded into the category $\mathbf{FQC}^{\simeq}$ by explicitly adding the heap and garbage, as $\phi \in \mathbf{FQC}^{\simeq}\ (a \otimes h)\ (b \otimes g)$, with the same requirement that $a \otimes h = b \otimes g$.

Figure 4.1 gives a visualisation of an **FCC** morphism, and the visualisation of a quantum morphism $(h, g, \phi) \in \mathbf{FQC}\ a\ b$ is identical:



All $\mathbf{FQC}^{\simeq}$ morphisms can be lifted into the category $\mathbf{FQC}$: for any reversible computation $\phi \in \mathbf{FQC}^{\simeq}\ a$ there is an equivalent computation $\widehat{\phi} \in \mathbf{FQC}\ a\ a$, as in the classical case. The same rule as given in equation 4.1 holds for the quantum case, suitably adjusted:

$$\frac{\phi \in \mathbf{FQC}^{\simeq}\ a}{\widehat{\phi} \in \mathbf{FQC}\ a\ a}$$

where $\widehat{\phi} = (I, I, \phi)$ and $I = 0$; there is no heap or garbage.

To summarise, given the finite sets $a, b \in \mathbb{N}$ which represent the size of the input and output quantum registers, the finite quantum computation $\phi \in \mathbf{FQC}\ a\ b$ is given by the triple $\phi = (h, g, \phi')$ where

$h \in \mathbb{N}$ is the size of the initial heap, initialised to 0

$g \in \mathbb{N}$ is the size of the garbage, to be disposed at the end of the computation

$\phi' \in \mathbf{FQC}^{\simeq}\ c$ is a reversible quantum computation, with $c = a \otimes h = b \otimes g$

As with the classical case of $\mathbf{FCC}$, only finite sets of powers of 2 are considered (bit-vectors), and the heap is always initialised as a vector of 0s, so no extra information about the heap is required.

To finalise the definition of $\mathbf{FQC}$ as a category, definitions of identity and composition are required. These definitions correspond exactly to the equivalent definitions for $\mathbf{FCC}$: given $a \in \mathbb{N}$, the identity morphism can be defined as $id_a = (I, I, id_a)$, as in the case of $\mathbf{FCC}$. This is a lifting of the identity morphism for $\mathbf{FQC}^{\simeq}$, in the same way that the identity morphism for $\mathbf{FCC}$ is lifted from $\mathbf{FCC}^{\simeq}$.

Composition of two computations given as $\mathbf{FQC}$ morphisms is also defined in the same way as for the category $\mathbf{FCC}$: given $\alpha = (h_\alpha, g_\alpha, \phi_\alpha) \in \mathbf{FQC}\ a\ b$ and $\beta = (h_\beta, g_\beta, \phi_\beta) \in \mathbf{FQC}\ b\ c$, the composition morphism is given by $\beta \circ \alpha = (h, g, \phi) \in \mathbf{FQC}\ a\ c$, where

$$\begin{aligned} h &= h_\alpha \otimes h_\beta \\ g &= g_\alpha \otimes g_\beta \\ \phi_{\beta\circ\alpha} &= (\phi_\beta \otimes id_{g_\alpha}) \circ (\phi_\alpha \otimes id_{h_\alpha}) \end{aligned}$$

which is shown diagrammatically in figure 5.1. Note that some trivial monoidal swap

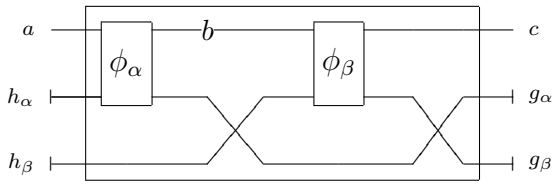


FIGURE 5.1: Sequential composition in **FQC**: $\phi_{\beta\circ\alpha}$. Compare with figure 4.2 showing sequential composition in **FCC**.

isomorphisms have been omitted from this definition of $\phi_{\beta\circ\alpha}$ given here, but are shown in figure 5.1. The full definition is the same as that given in section 4.4 for the case of **FCC**.

Parallel composition for **FQC** can also be defined following the pattern used to define parallel composition for the category **FCC**, but replacing the Cartesian product on sets with the tensor product on Hilbert spaces: given $\alpha \in \mathbf{FQC}\ a\ b$ and $\beta \in \mathbf{FQC}\ c\ d$, $\alpha \otimes \beta = (h, g, \phi) \in \mathbf{FQC}\ (a \otimes c)\ (b \otimes d)$ where

$$\begin{aligned} h &= h_\alpha \otimes h_\beta \\ g &= g_\alpha \otimes g_\beta \\ \phi_{\alpha\otimes\beta} &= \phi_\alpha \otimes \phi_\beta \end{aligned}$$

Again, monoidal isomorphisms are omitted from the definition, but are shown in figure 5.2, which gives a diagrammatic interpretation of parallel composition in **FQC**.

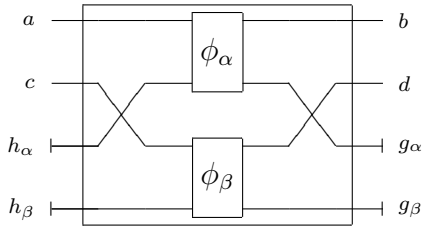


FIGURE 5.2: Parallel composition in **FQC**: $\phi_{\alpha\otimes\beta}$. Compare with figure 4.3 showing parallel composition in **FCC**

The neutral element of parallel composition, which is the tensor product on **FQC**, can be obtained by lifting $I_{\mathbf{FQC}^\simeq}$ to $I_{\mathbf{FQC}}$.

This completes the definition of the category **FQC**, the category of irreversible (and reversible) finite quantum computations, which formalises categorically the quantum circuit model of quantum computations.

5.2.1 Strict and impure quantum computations

A category of quantum computations of particular interest is the category of *strict*, or *pure*, computations $\alpha = (h, \phi) \in \mathbf{FQC}^{\circ}\ a\ b$, which formalises the notion of quantum computations that do not involve measurement. $\mathbf{FQC}^{\circ}$ computations are be defined as computations where there is no garbage, $g = 0$, hence it is omitted from the definition. It is a monoidal subcategory of **FQC**, and is of interest as it is the category of all pure quantum computations. Strict computations are not simply reversible computations, since heap inputs are allowed. The lack of garbage output, however, means that $\mathbf{FQC}^{\circ}$ morphisms can be modelled as linear operations from pure quantum states to pure quantum states. In general, **FQC** morphisms allow garbage, which implies measurement. Measurement induces the decoherence of a quantum state, leading to probabilistic results, and so irreversible quantum computations in **FQC** cannot be modelled as linear operations between pure states. It follows that $\mathbf{FQC}^{\circ}$ is the largest subset of **FQC** that can be modelled using linear unitary mappings between pure quantum states. Mixed quantum states can be represented as density matrices, and linear functions between density matrices are called superoperators, introduced in section 5.5.1. The use of superoperators gives a mathematical model for understanding general **FQC** computations.

From the definitions of $\mathbf{FQC}^{\simeq}$ (no heap or garbage), $\mathbf{FQC}^{\circ}$ (no garbage but allows heap) and **FQC** (allows heap and garbage), it follows that:

$$\mathbf{FQC}^{\simeq} \subset \mathbf{FQC}^{\circ} \subset \mathbf{FQC}$$

While morphisms on $\mathbf{FQC}^{\simeq}$ and $\mathbf{FQC}^{\circ}$ can be modelled as linear operators between pure quantum states, the use of garbage in **FQC** means that **FQC** morphisms are modelled as superoperators on density matrices, presented in section 5.5.1.

5.3 Modelling reversible quantum computations in $\mathbf{Q}^{\simeq}$

$\mathbf{FQC}^{\simeq}$ can be understood mathematically as unitary operators between complex-valued vectors, which model the morphisms and objects of $\mathbf{FQC}^{\simeq}$. As $\mathbf{FQC}^{\simeq}$ is simply a categorical formalism of the quantum circuits presented in section 3.3, then the mathematical interpretation is the same as that for quantum circuits. A brief categorical reformulation of the linear algebra from section 3.2 will be presented here, followed by an interpretation of $\mathbf{FQC}^{\simeq}$ objects in the category of unitary operators: $\mathbf{Q}^{\simeq}$ (pronounced *Unit*, for unitary).

5.3.1 Categorical review of linear algebra

In section 3.2, definitions of general linear algebra concepts were presented with definitions provided in the functional language Haskell. In this section those definitions will be reformulated as definitions in the category **FinVec**, of finite complex-valued vectors, making use of the category of finite sets, **FinSet**.

The objects of the category of finite sets **FinSet** are again the natural numbers, $a \in \mathbb{N}$, which are identified with their initial segment, $[a]$. The homsets of the category are given by $\mathbf{FinSet}\ a\ b = a \to b$. Given $a \in \mathbb{N}$, the function $\mathbf{C}\ a = a \to \mathbb{C}$ can be defined. This function on objects $\mathbf{C} \in \mathbf{FinSet} \to \mathbf{Set}$ is monadic, *i.e.* it gives rise to the Kleisli structure discussed in section 3.2.2. Monadic return and bind ($\gg\!=$) operations can be formulated in this category as:

$$
\begin{aligned}
&\mathit{return} \in a \to \mathbf{C}\ a \\
&\mathit{return}\ a = \lambda b \to \mathbf{if}\ a \equiv b\ \mathbf{then}\ 1\ \mathbf{else}\ 0 \\
&(\gg\!=) \in (\mathbf{C}\ a) \to (a \to \mathbf{C}\ b) \to \mathbf{C}\ b \\
&v \gg\!= f = \lambda b \to \Sigma\ a\ .\ (v\ a) \times (f\ a\ b)
\end{aligned}
$$

The associated Kleisli category is the category of finite dimensional complex vector spaces **FinVec**. The homsets of **FinVec** are given by $\mathbf{FinVec}\ a\ b = a \to \mathbf{C}\ b$, where $a, b \in \mathbb{N}$, and hence correspond to $a \times b$ complex matrices. Since computations over bit-vectors are being modelled, it is useful to define $\mathbf{C_2}\ a = \mathbf{C}\ (\mathbb{N}_2 \to a)$ and $\mathbf{FinVec_2}\ a\ b = \mathbf{FinVec}\ (\mathbb{N}_2 \to a)\ (\mathbb{N}_2 \to b)$. This category is the category of the vectors and matrices discussed in section 3.2, and implemented in Haskell, with the

monadic return and bind given by the functions *vreturn* and $\gg=$, shown in figure 3.2. Note that what is referred to here as a monad is actually a more general notion, which is sometimes referred to as an indexed monad or a Kleisli structure; see section 3.2 for details.

The Cartesian product on finite sets gives the tensor product on **FinVec**: on objects, $a \otimes b = ab$; while on morphisms, given $f \in \mathbf{FinVec}\ a\ b$ and $g \in \mathbf{FinVec}\ c\ d$, the tensor product is

$$f \otimes g \in \mathbf{FinVec}\ (a \otimes c)\ (b \otimes d)$$
$$f \otimes g = \lambda(a, c) \to \lambda(b, d) \to (f\ a\ b) \times (g\ c\ d)$$

The unit of the tensor is $I = 1$, and $(\mathbf{FinVec}, \otimes, I)$ is a strict monoidal category. The tensor product in $\mathbf{FinVec_2}$ is given by $+$ on the natural numbers, which are identified with their initial segments.

Some useful operations defined in section 3.2 can also be reformulated into the category **FinVec**. The inner-product $\langle v|w\rangle \in \mathbb{C}$ of two vectors $v, w \in \mathbf{C}\ a$ is defined as

$$\langle v|w\rangle = \Sigma a.(v\ a)^* \times (w\ a)$$

where $(x + y \times i)^* = x - y \times i$ is the complex conjugate. The *norm* of a vector $|v| \in \mathbb{R}^+$ is defined as $|v| = \langle v|v\rangle$, and is the inner-product of a vector with itself. Two vectors are orthogonal, $v \perp w$, if $\langle v|w\rangle = 0$. A base of a vector space is orthonormal if every base vector is orthogonal to every other base vector. The adjoint of a vector $f \in \mathbf{FinVec}\ a\ b$ is given by $f^\dagger = \lambda b\ a \to (f\ a\ b)^*$, with the defining property that $\langle v|fw\rangle = \langle f^\dagger v|w\rangle$.

By definition, a map $u \in \mathbf{FinVec}\ a\ b$ is unitary if its adjoint is its inverse: $u \circ u^\dagger = I$. This implies that u is an isomorphism, and hence also that $a = b$. Unitary maps are isometric, *i.e.* they preserve the inner-product, $\langle v|w\rangle = \langle u\ v|u\ w\rangle$. However, not all isometric maps are unitary; for example, the diagonal maps given by $\delta_\sigma \in \mathbf{FinVec}\ \sigma\ (\sigma \otimes \sigma)$, which are given by $\delta_\sigma\ a\ (b, c) = \mathbf{if}\ a \equiv b\ \&\ b \equiv c\ \mathbf{then}\ 1\ \mathbf{else}\ 0$, are isometric but not unitary.

A linear map $f \in \mathbf{FinVec}\ a\ a$ is self-adjoint, *iff* $f = f^\dagger$. A self-adjoint map has only real eigenvalues; $f\ v = \lambda \times v$ implies $\lambda \in \mathbb{R}$. The map is positive if all eigenvalues

are positive, that is $\lambda \geqslant 0$. The trace of a map $tr\ f$ is the sum of all eigenvalues, which can be directly calculated as $tr\ f = \Sigma\ a.f\ a\ a$.

All basic linear algebra definitions required for the understanding the mathematical models of **FQC** have now been defined.

5.3.2 $\mathbf{FQC}^{\simeq}$ as unitary operations

In section 4.6, the category $\mathbf{FCC}^{\simeq}$ is given an interpretation in terms of the category **FinSet**. In this section a parallel development giving an interpretation of $\mathbf{FQC}^{\simeq}$ in the category $\mathbf{Q}^{\simeq}$ of unitary operators is presented.

The category $\mathbf{FQC}^{\simeq}$ models quantum circuits with no heap and no garbage, and hence no measurement takes place. Therefore, reversible quantum computations can be modelled by linear functions on pure quantum states, which are modelled as vectors with *norm* 1.

The category $\mathbf{Q}^{\simeq}$ is introduced to model reversible computations. The objects of the category $\mathbf{Q}^{\simeq}$ are the natural numbers, which correspond to the size of the quantum register, or equivalently the number of wires used by the circuit. The homsets $\mathbf{Q}^{\simeq}\ a\ b$ are the unitary maps $u \in \mathbf{FinVec_2}\ a\ b$. As $\mathbf{Q}^{\simeq}$ is the category of reversible computations, $u \in \mathbf{Q}^{\simeq}\ a\ b$ is only non-empty if $a = b$, hence $\mathbf{Q}^{\simeq}\ a\ a$ is abbreviated to $\mathbf{Q}^{\simeq}\ a$; as $\mathbf{FQC}^{\simeq}\ a = \mathbf{FQC}^{\simeq}\ a\ a$.

All circuits $\phi \in \mathbf{FQC}^{\simeq}\ a$ can be interpreted as $[\![\phi]\!] \in \mathbf{Q}^{\simeq}\ a$, by induction over the inductive definition of $\mathbf{FQC}^{\simeq}$ given in section 5.1:

rotation $[\![rot\ u]\!] = u$

wires $[\![wires\ \phi]\!] = f$ where $f\ a\ b = \mathbf{if}\ \phi\ a \equiv b\ \mathbf{then}\ 1\ \mathbf{else}\ 0$.

sequential composition $[\![\phi \circ \psi]\!] = [\![\phi]\!] \circ [\![\psi]\!]$

parallel composition $[\![\phi \otimes \psi]\!] = [\![\phi]\!] \otimes [\![\psi]\!]$, using the fact that $2^{a \times b} = 2^a + 2^b$.

conditional $[\![\phi \mid \psi]\!] = [\![\phi]\!] | [\![\psi]\!]$ where

$$\phi|\psi\ (0,a)\ (0,b) = \phi\ a\ b$$
$$\phi|\psi\ (1,a)\ (1,b) = \psi\ a\ b$$
$$\phi|\psi\ \_\quad\ \_\quad = 0$$

Note that $\phi|\psi$ is unitary/isometric if both ϕ and ψ are also unitary/isometric.

This mapping from reversible circuits to unitary operators is in fact exactly the same as the compilation function presented in section 3.8, which takes as input a reversible quantum circuit and outputs a unitary matrix that models the circuit.

As $\mathbf{Q}^{\simeq}$ is a monoidal category, and *wires* extends to a monoidal functor, it follows that $[\![\cdot]\!]$ respects the equality on computations. However, it should be noted that it identifies more computations than circuit equality, as it identifies circuits which have the same behaviour; see section 5.5.4. This interpretation is full; every $\phi \in \mathbf{Q}^{\simeq}\ a$ can be generated by the appropriate reversible computation, which is a consequence of the Solovay–Kitaev theorem [21], that describes an efficient classical algorithm for creating an arbitrary single-qubit superposition.

5.4 Modelling strict quantum computations in $\mathbf{Q}^{\circ}$

In this section a mathematical model for the category $\mathbf{FQC}^{\circ}$, of strict quantum computations, will be presented. The category $\mathbf{FQC}^{\circ}$ is an extension of $\mathbf{FQC}^{\simeq}$, with the addition of ancillary heap inputs, $(h, \phi') \in \mathbf{FQC}^{\circ}\ a\ b$ such that $b = a + h$. As ϕ' is a reversible quantum circuit then the computation can still be understood in terms of unitary operators over a Hilbert space; however the notion of heap is lost in this interpretation. A more natural setting for the mathematical interpretation of $\mathbf{FQC}^{\circ}$ morphisms would allow the embedding of a Hilbert space a into a (possibly) larger Hilbert space b, which models the addition of the heap to the computation. This can be achieved by the use of isometries rather than unitary operators to model strict computations with heap.

5.4.1 $\mathbf{FQC}^{\circ}$ as isometric operations

An isometry is a completely positive, distance preserving isomorphism between metric spaces. In the case of strict quantum computations the mapping is between pure quantum states, represented in the usual way as complex-valued vectors. The distance function in quantum mechanics is given by the inner-product. Analogously to $\mathbf{Q}^{\simeq}$, the category $\mathbf{Q}^{\circ}$ (pronounced *Isom*, from isometry) is now introduced to mathematically model strict computations. The objects of the category $\mathbf{Q}^{\circ}$ are the natural numbers, as in $\mathbf{Q}^{\simeq}$ (section 5.3.2), and the morphisms are the isometric operators between objects: $i \in \mathbf{FinVec_2}\ a\ b$, where i is an isometry from a register of size a to one of size b, with $a \leqslant b$, hence an isometry can be written $i \in \mathbf{Q}^{\circ}\ a\ b$.

A method of interpreting heap is required to model $\mathbf{FQC}^{\circ}$, and this is achieved using a map which initialises the heap of a computation: given $0^h \in \mathbf{C_2}\ h$, a heap initialisation map $\otimes 0^h \in \mathbf{FinVec_2}\ a\ (a \otimes h)$, can be defined which initialises the second part of the product, the heap. The heap initialisation is isometric: $\otimes 0^h \in \mathbf{Q}^{\circ}\ a\ (a \otimes h)$. Using the heap initialisation, $\mathbf{FQC}^{\circ}$ objects can now be interpreted; any $(h, \phi) \in \mathbf{FQC}^{\circ}\ a\ b$ with $\phi \in \mathbf{FQC}^{\simeq}\ (a \otimes h)\ b$, is interpreted as $[\![h, \phi]\!] \in \mathbf{Q}^{\circ}\ a\ b$ by $[\![h, \phi]\!] = \phi\ (\otimes 0^h)$, hence $0^h \in \mathbf{C_2}\ h$ is the constant zero vector; a collection of h wires initialised to 0 in the circuit interpretation.

For orthogonal maps; that is maps $f, g \in \mathbf{Q}^{\circ}\ a\ b$ such that for all $\vec{a} \in \mathbf{C_2}\ a$ it is the case that $f\ \vec{a} \perp g\ \vec{a}$ (see section 5.3.1), another form of the conditional can be defined, $f|^{\circ}g \in \mathbf{Q}^{\circ}\ (\mathcal{Q}_2 \otimes a)\ b$, as:

$$f|^{\circ}g\ (0, a)\ b = f\ a\ b$$
$$f|^{\circ}g\ (1, a)\ b = g\ a\ b$$

It can be seen from the definitions that $f|g = (0 \otimes f)|^{\circ}(1 \otimes g)$.

5.5 Modelling irreversible quantum computations in Q

In this section a mathematical model for the category **FQC**, of irreversible finite quantum computations, is presented. **FQC** morphisms include the notion of garbage, which are extra qubits required for the computation to be carried out, but which are

discarded at the end. The garbage is removed by measuring it, which results in a probabilistic state, called a mixed state. Mixed states and the 'projection' of garbage out of a computation cannot be modelled using the concepts discussed so far, as they only deal with pure states and with irreducible state spaces. To model **FQC** in general, a mathematical formalism for mixed states and operations between them has to be provided, and this is outlined in section 5.5.1.

5.5.1 Density matrices and superoperators

The usual canonical way of representing probabilistic quantum states is to use the density matrix notation. Given a quantum system that can be in a number of possible pure states $|\phi_i\rangle$, with respective probabilities p_i, the density matrix for this state is given by the equation:

$$\rho \equiv \sum_i p_i \left|\phi_i\right\rangle \left\langle\phi_i\right| \tag{5.1}$$

From this it follows that a pure quantum state can be embedded into its density matrix form by taking its outer product, defined in section 3.2. To interpret irreversible computations, mixed states have to be modelled, which arise as the result of a measurement. This is achieved using density matrices. A mixed state of size a is represented as a positive map $\rho \in \mathbf{FinVec_2}\ a\ a$, such that $||\rho|| = 1$, which is another definition of a density matrix. The probability that a density matrix ρ is in state v is λ, if $\rho\ v = \lambda \times v$. In mathematical terms, v is an eigenvector of ρ with the eigenvalue λ. The trace condition ensures that this is a probability distribution on the vectors, in any orthonormal base.

Morphisms between density matrices are called *superoperators*. A superoperator is a trace preserving, completely positive, linear operator between density matrices. In the density matrix interpretation the trace of the matrix gives the sum of the probabilities of each pure state, hence the requirement that morphisms between density matrices are trace preserving. Formally, a linear map $f \in \mathbf{FinVec_2}\ (a \otimes a)\ (b \otimes b)$ can be interpreted as an operator on density matrices by using $\mathbf{FinVec_2}\ a\ a \simeq \mathbf{C_2}\ (a \otimes a)$. (Note that this is actually a slight simplification, as it should be $\mathbf{C_2}(a \otimes a^{\perp})$, however, as the objects of this category are natural numbers, the two definitions are equivalent.)

An operator defined in this way is positive if it preserves positivity, by definition. It is completely positive if $f \otimes (c \otimes c) \in \mathbf{FinVec_2}\ ((a \otimes c) \otimes (a \otimes c))\ ((b \otimes c) \otimes (b \otimes c))$ is positive for any $c \in \mathbb{N}$. A completely positive operator is a superoperator if it is trace preserving. The category of superoperators is defined as $\mathbf{Q}$, (pronounced *Super*, for superoperator), where the objects are again the natural numbers, and its morphisms are superoperators. In other words $s \in \mathbf{Q}\ a\ b$ is given by $s \in \mathbf{FinVec_2}\ (a \otimes a)\ (b \otimes b)$, which are completely positive and norm-preserving. The tensor product on superoperators is given by the tensor product of the underlying vector space, with a trivial permutation.

An isometry f between pure states can be similarly be lifted to a superoperator $\widehat{f}$. Formally, given $f \in \mathbf{Q}^{\circ}\ a\ b$, a superoperator $\widehat{f} \in \mathbf{Q}\ a\ b$ can be defined as follows: given a density matrix $\rho \in \mathbf{FinVec_2}\ a\ a$, the lifting given by

$$\widehat{f}\ \rho = f\ \circ\ \rho\ \circ\ f^{\dagger} \in \mathbf{FinVec_2}\ b\ b$$

is constructed using the isomorphism $\mathbf{FinVec}\ a\ a \simeq \mathbf{C}\ (a \otimes a)$. This gives rise to:

$$\widehat{f} \in \mathbf{FinVec_2}\ (a \otimes a)\ (b \otimes b)$$

which is completely positive and trace preserving.

5.5.2 Interpreting measurement as a superoperator

In contrast to the strict model of quantum computing, it is possible to define a superoperator which "forgets" part of a quantum state. This can be thought of as a form of projection, but is correctly termed a *partial trace*, as part of the computation is traced out by measuring it and calculating the effect on the retained parts of the computation. Using the partial trace allows garbage to be properly dealt with, by removing it from the computation. Formally, measurements are interpreted as a partial trace, defined as $\mathrm{Tr}_{(a,g)} \in \mathbf{Q}\ (a \otimes g)\ a$ where

$$\mathrm{Tr}_{(a,g)} \in \mathbf{FinVec_2}\ ((a \otimes g) \otimes (a \otimes g))\ (a \otimes a)$$

$$\begin{aligned}\mathrm{Tr}_{(a,g)} = \lambda(a, g)\ (a', g') \rightarrow\ & \mathbf{if} \quad g \equiv g' \\ & \mathbf{then}\ \mathit{return}\ (a, a') \\ & \mathbf{else}\ \ \lambda(\_, \_) \rightarrow 0\end{aligned}$$

Classically, the product space of a and g is given by the Cartesian product of the finite sets a and g, $(a, g) \in a \times g$, and the standard projection operation $\pi_0(a, g)$ can be used to project out the garbage to give the reduced state a. The partial trace is the quantum mechanical equivalent of this operation, which correctly interprets the effects of measurement of a quantum state. It is clear from this definition that the principle of deferred measurement (see section 3.10.1) holds. Thus garbage can be measured and removed as soon as it is created (as soon as it becomes garbage), or as a final step of the computation – as in the operational semantics of QML given in chapter 7.

5.5.3 FQC as superoperators

Using superoperators, the partial trace defined above, and the interpretation of $\mathbf{FQC}^\circ$ defined previously, $(h, g, \phi) \in \mathbf{FQC}\ a\ b$ can be interpreted as $[\![(h, g, \phi)]\!] \in \mathbf{Q}\ a\ b$ using $[\![(h, \phi)]\!] \in \mathbf{Q}^\circ\ a\ (b \otimes g)$, which is embedded into $\mathbf{FQC}$ as $\widehat{[\![(h, \phi)]\!]} \in \mathbf{Q}\ a\ (b \otimes g)$, and finally the garbage is removed using the partial trace, giving:

$$[\![(h, g, \phi)]\!] \in \mathbf{Q}\ a\ b$$
$$[\![(h, g, \phi)]\!] = \mathrm{Tr}_{(b,g)}\ \widehat{[\![(h, \phi)]\!]}$$

The "morphisms as superoperators" interpretation can be shown as the following commuting square:

$$\begin{array}{ccc} a \otimes h & \xrightarrow{[\![\phi]\!]} & b \otimes g \\ {\scriptstyle (-,0^h)}\uparrow & & \downarrow{\scriptstyle \mathrm{tr}_g} \\ a & \xrightarrow[f]{} & b \end{array}$$

This is the same picture as given for the classical "morphisms as functions" interpretation, in section 4.7, except with the projection of the garbage (π_g) replaced by a partial trace (tr_g), and all functions replaced by superoperators.

It follows from the Kraus representation theorem [53] that this interpretation is full, *i.e.* any superoperator can be realised as a morphism in $\mathbf{FQC}$.

5.5.4 Extensional equality of **FQC**

Analogously to the classical case, the extensional equality of **FQC** morphism is understood by interpreting them as superoperators. A morphism $(h, g, \phi) \in \mathbf{FQC}\ a\ b$ is interpreted as $\mathrm{tr}_g \circ [\![\phi]\!] \circ \otimes 0^h \in \mathbf{Q}\ a\ b$, where $[\![\phi]\!] \in \mathbf{Q}\ (h \otimes a)\ (g \otimes b)$ is the superoperator associated to the unitary operator, given by interpreting the reversible circuit ϕ, where $\otimes 0^h \in \mathbf{Q}\ a\ (a \otimes h)$ initialises the heap, and $\mathrm{tr}_g \in \mathbf{Q}\ (g \otimes b)\ b$ is a partial trace, which traces out the garbage.

The definition of extensional equality for **FQC** follows from the above interpretation: two **FQC** morphisms are extensionally equivalent if this interpretation gives rise to the same superoperator for both morphisms. This is the same picture as given for classical extensional equality, in section 4.7.1, except using the "morphism as superoperators" interpretation in the quantum case, rather than the classical "morphisms as functions" interpretation.

The mapping $\mathbf{FQC} \to \mathbf{Q}$ *is functorial*

The interpretation of **FQC** morphisms as morphisms in **Q**, given by $[\![\cdot]\!]$, is actually functorial. The fact that $[\![\cdot]\!] \in \mathbf{FQC}\ a\ b \to \mathbf{Q}\ a\ b$ is functorial follows from the fact the properties of a functor are satisfied, especially composition in **FQC** as defined in section 5.2, are preserved by $[\![\cdot]\!]$. This is shown by the following proposition:

Proposition 2. $[\![\cdot]\!] \in \mathbf{FQC} \to \mathbf{Q}$ is a strict monoidal functor, that is:

$$[\![\mathrm{id}]\!] = \mathrm{id}$$

$$[\![f \circ g]\!] = [\![f]\!] \circ [\![g]\!]$$

$$[\![f \otimes g]\!] = [\![f]\!] \otimes [\![g]\!]$$

Proof. Both 1. and 3. follow directly from monoidal identities. The only interesting case is 2. which follows from the fact that the diagram shown in figure 5.3 commutes. □

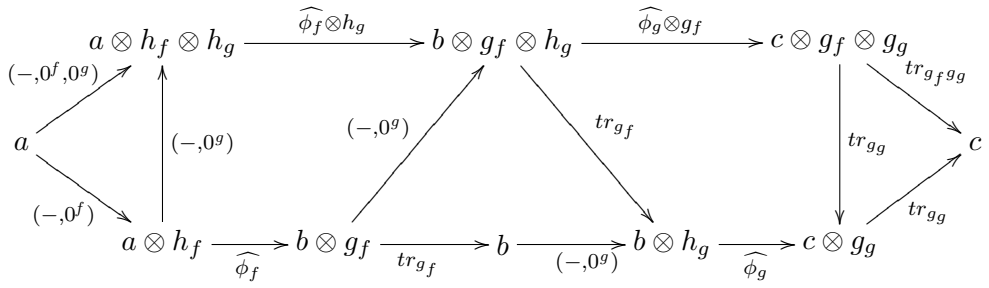


FIGURE 5.3: Composition is preserved by lifting **FQC** to **Q**. The path along the top of the diagram shows $[\![f \circ g]\!]$, while the lower path shows $[\![f]\!] \circ [\![g]\!]$, and this diagram commutes.

The functor given by $[\![\cdot]\!]$ here should be compared to that given in the case of **FCC**, which maps **FCC** objects and morphisms into objects and morphisms of the category **FinSet**, given in section 4.7. The $[\![\cdot]\!]$ functors in both cases are very similar, with the Cartesian product $\times$ on sets and the projection function in the classical case being replaced by the tensor product $\otimes$ and the partial trace superoperator in the quantum case above.

5.6 Comparing FCC with FQC

The development of **FQC** above follows directly the development of **FCC** given in chapter 4. In each case the development begins by formalising the notions of circuits as categories, presented in chapter 2 in the case of classical computing, and chapter 3 for quantum computing. The categories $\mathbf{FCC}^{\simeq}$ and $\mathbf{FQC}^{\simeq}$ are similar, as the underlying circuits are similar, as discussed in section 3.4. The categories $\mathbf{FCC}^{\simeq}$ and $\mathbf{FQC}^{\simeq}$, and the derived categories **FCC** and **FQC** all have as their objects natural numbers which represent the finite sets upon which the computations, which are the categorical morphisms, are performed. These similarities lead to the similar definitions of each category. The table in figure 5.4 shows how concepts from **FCC** have been implemented in **FQC**, by lifting each concept into the quantum mechanical notion of Hilbert spaces, and is an extension of the table given previously in figure 3.4.

Classical Case (**FCC**)	Quantum Case (**FQC**)
Finite sets	Finite dimensional Hilbert spaces
Cartesian product ($\times$)	Tensor product ($\otimes$)
Bijections	Unitary operators
Functions	Superoperators
Injective functions	Isometries
Projections	Partial trace

FIGURE 5.4: Table showing the analogous concepts of **FCC** and **FQC**

5.7 FQC in Haskell

The Haskell implementation of **FQC** is essentially identical to the implementation of **FCC** given in section 4.8. An **FQC** morphism is defined as $(h, g, \phi) \in \mathbf{FQC}\ a\ b$ where $a, b, h, g \in \mathbb{N}$ and $\phi \in \mathbf{FQC}^{\simeq}$. a and b give the size of the input and the output to the morphism, and h and g give the size of any heap and garbage required to compute ϕ. Using the quantum *Circ* datatype presented in section 3.5 to model $\mathbf{FQC}^{\simeq}$ objects in Haskell, a datatype encapsulating morphisms of **FQC** can be defined as:

$$\textbf{data FQC} = \mathbf{FQC}\{\, a, b, h, g \in \mathit{Int}, \phi \in \mathit{Circ} \,\}$$

The only differences with the definition of **FCC** are the renaming of the type from **FCC** to **FQC**, and the use of the quantum rather than classical *Circ* datatype.

As in the implementation of **FCC**, this datatype can be used to generate computations that are not in **FQC**, so a validation function, *validFQC*, is again required, which is essentially identical to the **FCC** validation function *validFCC*:

$$\begin{aligned}
&\mathit{validFQC} \in \mathbf{FQC} \to \mathit{Error}\ \mathbf{FQC} \\
&\mathit{validFQC}\ f = \textbf{do let}\ \mathit{ah} = \mathit{OK}\ (a\ f + h\ f) \\
&\qquad\qquad\qquad\qquad\ \ \mathit{bg} = \mathit{OK}\ (b\ f + g\ f) \\
&\qquad\qquad\qquad\quad \mathit{eguard}\ (\mathit{ah} \equiv \mathit{bg})\ \texttt{"Input and output mismatch"} \\
&\qquad\qquad\qquad\quad \mathit{eguard}\ (\mathit{arity}\ (\phi\ f) \equiv \mathit{ah})\ \texttt{"Circuit arity mismatch"} \\
&\qquad\qquad\qquad\quad \mathit{return}\ f
\end{aligned}$$

If the circuit is invalid the appropriate error message will be propagated through the

arity function by the *Error* monad. An $\mathbf{FQC}^\circ$ morphism is an **FQC** morphism where $g = 0$, and an $\mathbf{FQC}^\simeq$ morphism is an $\mathbf{FQC}^\circ$ morphism where $h = 0$.

5.8 Q in Haskell

The categories $\mathbf{Q}^\simeq$, $\mathbf{Q}^\circ$ and **Q** can all be defined in Haskell using the implementation of basic linear algebra concepts presented in section 3.2. The category $\mathbf{Q}^\simeq$ of unitary operations is already implemented as the datatype *Mat*, with the restriction that the matrices have to be unitary; $U^\dagger U = UU^\dagger = I$. The unitary condition cannot be enforced by the type, so must be ensured by the programmer. Therefore, $u \in \mathbf{Q}^\simeq\ a$ is interpreted in Haskell as $Mat\ a\ a\ u$.

Haskell types of $\mathbf{Q}^\circ$ and **Q** follow the definition *Mat*, as they are also modelled as matrices. $i \in \mathbf{Q}^\circ\ a\ b$ can be interpreted in Haskell as $\mathbf{Isom}\{\, inI = a, outI = b, funI = i\,\}$, where **Isom** is defined as:

$$\mathbf{data\ Isom} = \mathbf{Isom}\{\, inI \in Int, outI \in Int, funI \in [\mathbb{N}_2] \to [\mathbb{N}_2] \to \mathbb{C}\}$$

which is essentially identical to the definition of *Mat*. The requirement that for all vectors v, w, $\langle v|w\rangle = \langle i\ v|i\ w\rangle$ has to be checked manually, as again it cannot be enforced. The function *funI* gives the action of the isometry i is a matrix, hence the *VEC* class of operations (adjoint, tensor product, inner-product, outer-product, and scalar-product) can be instantiated using the *Mat* instantiation, as can the bind operation ($\gg\!=$) which gives composition of isometries as matrix multiplication:

$$\begin{aligned}
&\mathbf{instance}\ VEC\ \mathbf{Isom}\ \mathbf{Isom}\ \mathbf{where}\\
&\quad adjoint\ (\mathbf{Isom}\ a\ b\ i) = \mathbf{Isom}\ a\ b\ i^\dagger\\
&\qquad \mathbf{where}\ f^\dagger\ x\ y = (f\ y\ x)^*\\
&\quad \mathbf{Isom}\ a_1\ b_1\ i_1 \otimes\ \ \mathbf{Isom}\ a_2\ b_2\ i_2 = \mathbf{Isom}\ a_3\ b_3\ i_3\\
&\qquad \mathbf{where}\ Mat\ a_3\ b_3\ i_3 = Mat\ a_1\ b_1\ i_1 \otimes Mat\ a_2\ b_2\ i_2\\
&\quad \mathbf{Isom}\ a_1\ b_1\ i_1\langle\cdot\rangle\ \ \mathbf{Isom}\ a_2\ b_2\ i_2 = Mat\ a_1\ b_1\ i_1\langle\cdot\rangle Mat\ a_2\ b_2\ i_2\\
&\quad \mathbf{Isom}\ a_1\ b_1\ i_1\rangle\cdot\langle\ \ \mathbf{Isom}\ a_2\ b_2\ i_2 = \mathbf{Isom}\ a_3\ b_3\ i_3\\
&\qquad \mathbf{where}\ Mat\ a_3\ b_3\ i_3 = Mat\ a_1\ b_1\ i_1\rangle\cdot\langle Mat\ a_2\ b_2\ i_2\\
&\quad x\qquad\qquad \$*\ \mathbf{Isom}\ a\ b\ i\quad = \mathbf{Isom}\ a\ b\ (\lambda\vec{b}_a \to \lambda\vec{b}_b \to x \times (i\ \vec{b}_a\ \vec{b}_b))\\
&\mathbf{instance}\ Bind\ Vec\ \mathbf{Isom}\ \mathbf{where}
\end{aligned}$$

$$v \gg\!= (\mathbf{Isom}\ a\ b\ f) = v \gg\!= (Mat\ a\ b\ f)$$

instance *Bind* **Isom Isom where**

$\mathbf{Isom}\ a_1\ b_1\ f_1 \gg\!= \mathbf{Isom}\ a_2\ b_2\ f_2 = \mathbf{Isom}\ a\ b\ f$

where $Mat\ a\ b\ f = Mat\ a_1\ b_1\ f_1 \gg\!= Mat\ a_2\ b_2\ f_2$

See section 3.2.2 for details of these implementations.

The category **Q** can be translated in a similar way with $s \in \mathbf{Q}\ a\ b$ becoming **Super** $a\ b\ s$, where **Super** is interpreted in Haskell as:

$\mathbf{data\ Super} = \mathbf{Super}\{\, inS \in Int, outS \in Int, funS \in [\mathbb{N}_2] \to [\mathbb{N}_2] \to \mathbb{C}\}$

An isometry $i \in \mathbf{Q}^\circ\ a\ b$ can be lifted into a superoperator, as defined in section 5.5.1, by the operation *isom2super*:

$isom2super \in \mathbf{Isom} \to \mathbf{Super}$

$isom2super\ i = \mathbf{Super}\ m\ n\ f\ \mathbf{where\ Isom}\ m\ n\ f = i \otimes (adjoint\ i)$

The *VEC* class can be instantiated with reference to the instantiation of **Isom**, by lifting each operation to **Super**. In the cases of all functions except the tensor product $\otimes$ this leads to the correct definition. The tensor product on **Super**, however, is not the same as the definition for **Isom**; the input has to be rearranged as the type of the tensor product on superoperators is: $\otimes \in \mathbf{FinVec_2}\ (a \otimes a')\ (b \otimes b') \to \mathbf{FinVec_2}\ (c \otimes c')\ (d \otimes d') \to \mathbf{FinVec_2}\ ((a \otimes c) \otimes (a' \otimes c'))\ ((b \otimes d) \otimes (b' \otimes d'))$. This can be seen by lifting the definition of $\otimes$ on **Isom** to **Super**, and is implemented in Haskell as:

$$\begin{aligned}
&\mathbf{Super}\ at\ bt\ s_1 \otimes \mathbf{Super}\ ct\ dt\ s_2 \\
&\quad = \mathbf{Super}\ (at + ct)\ (bt + dt) \\
&\qquad\qquad (\lambda bac \to \lambda bbd \to \\
&\qquad\qquad\quad \mathbf{let}\ (a, c, a', c') = tyPerm\ at\ ct\ bac \\
&\qquad\qquad\qquad\ (b, d, b', d') = tyPerm\ bt\ dt\ bbd \\
&\qquad\qquad\quad \mathbf{in}\ (s_1\ (a +\!\!+ a')\ (b +\!\!+ b')) \times (s_2\ (c +\!\!+ c')\ (d +\!\!+ d')))
\end{aligned}$$

where the auxiliary function *tyPerm* performs the appropriate permutation of the input. Composition of superoperators is defined as matrix multiplication, as the lifting of the isometric bind leaves the definition unchanged:

instance *Bind Dens* **Super where**

$$d \gg\!= \mathbf{Super}\ a\ b\ f = d \gg\!= (Mat\ (a \otimes a)\ (b \otimes b)\ f)$$

instance *Bind* **Super Super where**

$$\mathbf{Super}\ a_1\ b_1\ f_1 \gg\!= \mathbf{Super}\ a_2\ b_2\ f_2 = \mathbf{Super}\ a\ b\ f$$

where $Mat\ (a \otimes a)\ (b \otimes b)\ f = Mat\ (a_1 \otimes a_1)\ (b_1 \otimes b_1)\ f_1$
$\gg\!= Mat\ (a_2 \otimes a_2)\ (b_2 \otimes b_2)\ f_2$

The reason for the $a \otimes a$ and $b \otimes b$ that appear in these definitions, where $\otimes = +$, is that **Super** models the underlying $\mathbf{FinVec_2}$ object; recall from section 5.5.1 that $s \in \mathbf{Q}\ a\ b$ is given by $s \in \mathbf{FinVec_2}\ (a \otimes a)\ (b \otimes b)$. Density matrices are simply modelled as matrices, with the type synonym **type** $Dens = Mat$.

A superoperator for computing the partial trace, defined in section 5.5.2, can now be implemented in Haskell. The function *tr* takes two integers as input, the size of the output of the reversible computation, and the size of the garbage, and returns a superoperator which traces out the garbage to give the reduced state:

$$tr\ \in Int \to Int \to \mathbf{Super}$$

$$tr\ m\ n = \mathbf{Super}\ m\ n$$
$$(\lambda \vec{b}_a \to \mathbf{let}\ (ab_1, ab_2) = splitAt\ m\ \vec{b}_a$$
$$(a_1, b_1) \quad = splitAt\ n\ ab_1$$
$$(a_2, b_2) \quad = splitAt\ n\ ab_2$$
$$\mathbf{in\ if}\ b_1 \equiv b_2 \quad \mathbf{then}\ funV\ (vreturn\ (a_1 +\!\!+ a_2))$$
$$\mathbf{else}\ funV\ (vzero))$$

The use of this trace function, along with other functions defined in this section, enables a function to be defined which converts **FQC** morphisms into superoperators encoded in the **Super** datatype:

$$fqc2super\ \in \mathbf{FQC} \to \mathbf{Super}$$

$$fqc2super\ (\mathbf{FQC}\ a\ b\ h\ g\ \phi) = \mathbf{Super}\ a\ b\ ((tr\ b\ g)\ \circ\ superop\ \circ\ (heap\ a\ h))$$
$$\mathbf{where}\ superop = isom2super\ (comp\ \phi)$$

The function *fqc2super* makes use of the quantum circuit compile function $comp \in Circ \to \mathbf{Isom}$, updated to **Isom** from section 3.8, and a simple function *heap*, which initialises the heap to 0. Note this follows the definition given in section 5.5.3 of the mapping from **FQC** to **Q**. If the morphism is a strict $\mathbf{FQC}^\circ$ morphism, then a

similar function, *fqco2isom*, can be used to generate a isometry, which must ensure $h = 0$. The function *fqco2isom* is otherwise identical to *fqc2super*, but with the call to *isom2super* removed.

5.9 Summary

This chapter makes precise the notion of a reversible quantum computation, informally described in chapter 3, as objects and morphisms of the category $\mathbf{FQC}^{\simeq}$. From $\mathbf{FQC}^{\simeq}$, a category of irreversible quantum computations, Finite Quantum Computations (**FQC**) is derived, that interprets heap and garbage correctly. The category $\mathbf{FQC}^{\circ}$ of strict (no garbage) computations is also introduced. Though these categorical definitions, this chapter presents a formalism of the standard model of quantum computation – the quantum circuit model. Similarities and differences between **FQC** constructions and its classical analogue **FCC** are also highlighted and discussed.

Each of the three categories $\mathbf{FQC}^{\simeq}$, $\mathbf{FQC}^{\circ}$ and **FQC** are given mathematical interpretations, via functors into the categories $\mathbf{Q}^{\simeq}$, $\mathbf{Q}^{\circ}$, and **Q**, which are full. This gives a denotation to the operational understanding of quantum computations presented in the category **FQC** and its subcategories, and allows a definition of extensional equality to be defined for **FQC** morphisms, using the notions of partial trace and superoperators.

The chapter closes with implementations of **FQC** and **Q**, and useful operations, in Haskell. This follows from, and uses, the implementations of linear algebra constructs and quantum circuits, presented in sections 3.2 and 3.5.

Chapter 6

QML: A functional quantum programming language

In this chapter the functional quantum programming language QML will be introduced. The language QML features both quantum data structures, using the connective $\otimes$, and quantum control structures, in particular a quantum conditional **if**$^\circ$. The conditional **if**$^\circ$ analyses quantum data without measuring, and hence without changing the data. This is a different approach to most other proposed quantum languages discussed in section 1.4, which consider classical control structures only, *i.e.* where quantum data can only be processed using combinators corresponding to quantum circuits or by measurement. The motivations of the approach taken in this thesis and other design details of QML are presented, followed by the syntax of the language QML, and the structural and typing rules.

6.1 Quantum data and control

QML's type system is based on *strict linear logic*; that is linear logic with contraction, but without implicit weakening. This is in contrast to Selinger and Valiron [72], whose language has an affine type system, without implicit contraction. The absence of contraction in their language is motivated by the no-cloning property of quantum states. QML's type system allows implicit contraction, which is possible as contraction is not modelled by cloning, but by *sharing* as in classical functional

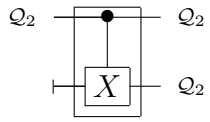


FIGURE 6.1: ϕ_δ: Implementing sharing by CNOT

programming languages. Indeed, on the level of reversible circuits, either classical or quantum, sharing can be realised using a conditional-not circuit, ϕ_δ, with the second input to the gate initialised with $|0\rangle$, as shown in figure 6.1. The circuit is, as expected, the diagonal for classical states, as it maps $|0\rangle$ to $|00\rangle$ and $|1\rangle$ to $|11\rangle$. It does not clone quantum states, such as $\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$, but shares them. The circuit would output $\frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|11\rangle$ and not $\frac{1}{\sqrt{2}}((|0\rangle + |1\rangle)(|0\rangle + |1\rangle))$, which corresponds to cloning. The contraction as sharing interpretation has been independently suggested by Arrighi and Dowek [5].

It should be noted that contraction is base dependant. For example, the function δ, defined as CNOT, does not commute with the Hadamard transform; *i.e.* δ is not natural.

6.2 Controlling weakening

One of the fundamental concepts of the design of QML is that it is not contraction which has to be carefully controlled, but weakening. The reason is that a quantum bit cannot be forgotten without first measuring it, and the measurement may affect other parts of the computation; for example, it will change qubits which are entangled with the qubit being discarded.

As an example, consider the following simple program, which appears to swap two qubits:

$$\begin{aligned} &\mathit{swap} \in \mathcal{Q}_\mathbf{2} \otimes \mathcal{Q}_\mathbf{2} \multimap \mathcal{Q}_\mathbf{2} \otimes \mathcal{Q}_\mathbf{2} \\ &\mathit{swap}\ p = \mathbf{let}\ (x, y) = p \\ &\qquad\qquad \mathbf{in}\ \ (y, x) \end{aligned}$$

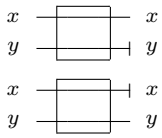


FIGURE 6.2: Interpreting variables as projections

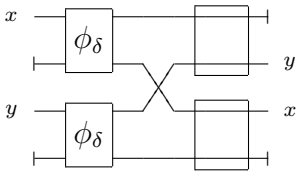


FIGURE 6.3: *swap* using projections

If a conventional type system is used, with rules such as

$$\frac{\Gamma \vdash t : \sigma \quad \Gamma \vdash u : \tau}{\Gamma \vdash (t, u) : \sigma \otimes \tau}$$

$$\frac{}{\Gamma, x : \sigma \vdash x : \sigma}$$

where the variables x and y are interpreted by projections, as in figure 6.2, then *swap* would be implemented as the circuit shown in figure 6.3, where the each variable is copied before one of each is projected out. This is essentially how a conventional functional language is implemented; the stack is used as temporary, easily disposable data storage. However, in a quantum context this implementation does not result in the desired state. Quantum states in the *computational basis* are properly swapped by the circuit given in figure 6.3. For example, $|01\rangle$ is correctly mapped to $|10\rangle$. However, a quantum superposition such as the state $\frac{1}{\sqrt{2}}((|0\rangle - |1\rangle)(|0\rangle + |1\rangle))$ is mapped to one of the basis states $|00\rangle, |01\rangle, |10\rangle, |11\rangle$ with equal probability. The *swap* operation has resulted in a completely random probabilistic state.

In order to remove these undesired effects, QML uses a strict linear type system,

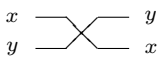


FIGURE 6.4: A quantum *swap* circuit that gives the desired behaviour

with the rules given previously substituted for the following:

$$\frac{\Gamma \vdash^{\circ} t : \sigma \quad \Delta \vdash^{\circ} u : \tau}{\Gamma \otimes \Delta \vdash^{\circ} (t, u) : \sigma \otimes \tau}$$

$$\frac{}{x : \sigma \vdash x : \sigma}$$

where the operation $\Gamma \otimes \Delta$ allows the context to be split. As a consequence, the program *swap* is now interpreted as the circuit given in figure 6.4 which behaves as a swap operation would be expected to, even on quantum states. For example, the state $\frac{1}{\sqrt{2}}((|0\rangle - |1\rangle)(|0\rangle + |1\rangle))$, is mapped to $\frac{1}{\sqrt{2}}((|0\rangle + |1\rangle)(|0\rangle - |1\rangle))$ using the new interpretation.

Throughout this thesis simple normalisation factors, such as $\frac{1}{\sqrt{2}}$, will be omitted to increase legibility.

6.3 Other sources of measurement

Weakening is not the only possible source of measurement in the context of quantum computing. Consider, for example, the following definition of negation:

$$mnot \in \mathcal{Q}_2 \multimap \mathcal{Q}_2$$

$$mnot\ x = \textbf{if}\ x\ \textbf{then qfalse}$$
$$\textbf{else qtrue}$$

If the classical control paradigm is followed, then branching over a qubit requires measurement. As a consequence, the interpretation of *mnot* does not work on superpositions as might reasonably be expected. For example, the input $(|0\rangle - |1\rangle)$ is mapped to either $|0\rangle$ or $|1\rangle$ with equal probability, because the classical **if** collapses the superposition, making it a probabilistic operation. What is meant by a reasonable

expectation for a quantum negation operator would be that, given $(|0\rangle - |1\rangle)$, it would return $(|1\rangle - |0\rangle)$. Indeed, in QML this behaviour can be implemented by using $\mathbf{if}^\circ$:

$$qnot \in \mathcal{Q}_\mathbf{2} \multimap \mathcal{Q}_\mathbf{2}$$

$$qnot\ x = \mathbf{if}^\circ\ x\ \mathbf{then\ qfalse}$$
$$\mathbf{else\ \ qtrue}$$

which is the quantum control structure that allows branching over quantum data, without measuring the control data. However, the classical conditional **if** cannot always simply be replaced by the quantum conditional $\mathbf{if}^\circ$. Consider the conditional swap program given by:

$$cswap \in \mathcal{Q}_\mathbf{2} \multimap \mathcal{Q}_\mathbf{2} \otimes \mathcal{Q}_\mathbf{2} \multimap \mathcal{Q}_\mathbf{2} \otimes \mathcal{Q}_\mathbf{2}$$

$$cswap\ c\ p = \mathbf{if}\ c\ \mathbf{then}\ swap\ p$$
$$\mathbf{else}\ \ p$$

If both components of p are the same then *cswap* effectively discards the control qubit. However, discarding quantum data is not possible without measurement, hence **if** cannot be replaced by $\mathbf{if}^\circ$ in this fragment. The only way to avoid this is to include the control qubit in the output:

$$cswap \in \mathcal{Q}_\mathbf{2} \multimap \mathcal{Q}_\mathbf{2} \otimes \mathcal{Q}_\mathbf{2} \multimap \mathcal{Q}_\mathbf{2} \otimes \mathcal{Q}_\mathbf{2} \otimes \mathcal{Q}_\mathbf{2}$$

$$cswap\ c\ p = \mathbf{if}^\circ\ c\ \mathbf{then}\ (\mathbf{qtrue}, swap\ p)$$
$$\mathbf{else}\ \ (\mathbf{qfalse}, p)$$

6.4 The design of QML

The design of the language QML is based on the considerations given above: QML has both quantum data and quantum control, QML allows contraction but controls weakening and implicit measurement. QML uses a strict linear logic with an explicit weakening operator, but with implicit contractions. This is justified by the fact that the meaning of a program can be affected by changing the weakenings, but not by moving contractions. Two conditional operators are also introduced: **if**, which measures a qubit; and $\mathbf{if}^\circ$, which does not measure, but which does require that the branches are orthogonal. This requirement is reflected by introducing an

orthogonality judgement on QML terms which invoke **if**$^\circ$.

The design of QML is also motivated by consideration of the operational semantics: the category **FQC**. In the same way that the category **FQC** of quantum computations was defined by analogy to the category **FCC** of classical computations, the language QML is designed to be as close to classical formalisms as possible. Indeed, a classical sublanguage of QML could be defined which uses **FCC** as its operational semantics, rather than **FQC**.

6.5 Syntax and typing rules

The syntax and typing rules of QML are based on strict linear logic: contractions are implicit, while weakenings are an explicit operation which correspond to measurements. QML's types are first order and finite. There are no recursive types, so, for example, a type of quantum natural numbers cannot be defined.

QML's type constructor is the tensor product, $\otimes$, which corresponds to a product type. Qubits and superpositions of single qubits are primitive. QML has two *if* constructs: **if**, which measures a qubit in the data it analyses (the classical–if); and **if**$^\circ$, which does not measure, but requires that the results will always exist in orthogonal subspaces (the quantum–if). The proofs of orthogonality can be inserted automatically by the compiler, and hence do not feature in the syntax of terms.

The Greek symbols σ, τ, ρ are used to vary over QML types, which are given by the following grammar:

$$\sigma = \mathcal{Q}_{\mathbf{1}} \mid \mathcal{Q}_{\mathbf{2}} \mid \sigma \otimes \tau$$

An infinite supply of concrete variable names is assumed, and x, y, z will be used to vary over names. 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 it is assumed that every variable appears at most once. Contexts correspond to functions from a finite set of variables to types.

To define the syntax of expressions constants $\kappa, \iota \in \mathbb{C}$ are also used, and function variables are used to refer to previously defined QML programs. The terms of QML

consist of those of a first-order functional language, extended with quantum data, a quantum control structure, and a measurement operator. The grammar of QML terms is defined as:

$$
\begin{array}{lll}
(\textit{Variables}) & x, y, \ldots \in & \textit{Vars} \\
(\textit{Prob. amplitudes}) & \kappa, \iota, \ldots \in & \mathbb{C} \\
(\textit{Patterns}) & p, q \quad ::= & x \mid (x, y) \\
(\textit{Terms}) & t, u \quad ::= & x \mid x^{\vec{y}} \\
 & & \mid () \mid (t, u) \\
 & & \mid \mathbf{let}\ p = t\ \mathbf{in}\ u \\
 & & \mid \mathbf{if}\ \ t\ \mathbf{then}\ u\ \mathbf{else}\ u' \\
 & & \mid \mathbf{if}^{\circ}\ t\ \mathbf{then}\ u\ \mathbf{else}\ u' \\
 & & \mid \mathbf{qfalse}^{\vec{y}} \mid \mathbf{qtrue}^{\vec{y}} \mid \mathbf{0} \\
 & & \mid \kappa \times t \quad \mid t + u
\end{array}
$$

Here, the vector notation $\vec{a}$ is used for sequences of syntactic objects. Formally, it corresponds to the following meta notation:

$$\vec{a} = \epsilon \mid a\ \vec{a}$$

Quantum data is modelled using the constructs $\kappa \times t$, $t + u$, with the constant $\mathbf{0}$ forming terms with a probability amplitude of zero. The term $\kappa \times t$, where κ is a complex number, associates the probability amplitude κ with the term t. The term $t + u$ describes a quantum superposition of t and u. Quantum superpositions are first class values, and can be used in a conditional to give quantum control. For example,

$$\mathbf{if}^{\circ}\ (\mathbf{qtrue} + \mathbf{qfalse})\ \mathbf{then}\ t\ \mathbf{else}\ u$$

evaluates both t and u and combines the results in a quantum superposition.

As an example of forming superpositions, the term $(\frac{1}{\sqrt{2}}) \times \mathbf{qfalse} + (\frac{1}{\sqrt{2}}) \times \mathbf{qtrue}$ is an equal superposition of **qfalse** and **qtrue**. Normalisation factors that are equal are sometimes omitted, and can then be inferred to be equal; the previous example becoming simply $\mathbf{qfalse} + \mathbf{qtrue}$.

A QML program is a sequence of function definitions $\vec{d}$, where a function definition d is given by $f\ \Gamma = t : \tau$. A Haskell style syntax will be used to present program examples, using $\multimap$ instead of $\rightarrow$ in the definition, inspired by the notation for linear

implication from linear logic. For example:

$$f : \sigma_1 \multimap \sigma_2 \,..\, \multimap \sigma_n \multimap \tau$$
$$f\ x_1\ x_2 \,..\, x_n = t$$

gives the following translation:

$$f\ (x_1 : \sigma_1, x_2 : \sigma_2, .., x_n : \sigma_n) = t : \tau$$

The basic typing judgements are:

Typing of terms

$$\vec{d}; \Gamma \vdash t : \sigma$$

Typing of strict terms

$$\vec{d}; \Gamma \vdash^{\circ} t : \sigma$$

Orthogonality

$$t \perp u$$

Well-typed programs

$$\vdash \vec{d}$$

Since all the typing rules presented above only pass $\vec{d}$, this will be omitted from rule definitions, writing $\Gamma \vdash t : \sigma$ (or $\Gamma \vdash^{\circ} t : \sigma$) instead, with the exception of the application rule (see section 6.6). To avoid repetition, the schematic judgements $\Gamma \vdash^{a} t : \sigma$ will be used, where $a \in \{-, \circ\}$.

The term $\Gamma \vdash t : \sigma$ means that term t has type σ under context Γ, and the term $\Gamma \vdash^{\circ} t : \sigma$ is used for measurement-free, *i.e.* strict, terms. Strict terms can be embedded into non-strict terms, $\vdash^{\circ}$ into $\vdash$, by induction over the structure of the derivations. This can be summarised by the following rule:

$$\frac{\Gamma \vdash^{\circ} t : \sigma}{\Gamma \vdash t : \sigma}$$

which is applicable, as an embedding for each strict term in QML to a non-strict can easily be derived from application of the non-strict rule. The embedding rule above is not formally introduced to reduce ambiguity in the derivations of terms. If the

embedding rule was defined as above then multiple interpretations of a term could be derived.

The operator $\otimes$ is introduced to map pairs of contexts to contexts:

$$\begin{aligned} \Gamma, x : \sigma \otimes \Delta, x : \sigma &= (\Gamma \otimes \Delta), x : \sigma \\ \Gamma, x : \sigma \otimes \Delta &= (\Gamma \otimes \Delta), x : \sigma \quad \text{if } x \notin \operatorname{dom} \Delta \\ \bullet \otimes \Delta &= \Delta \end{aligned}$$

This operation is partial, as it is only well-defined if the two contexts do not assign different types to the same variable. It will be implicitly assumed that whenever this operation is used it is well-defined.

6.5.1 Structural rules

There are two rules for interpreting a variable in QML, a strict and a non-strict version, which is the case for many QML typing rules. The strict variable rule requires the context to contain only the variable being used, and is given by:

$$\frac{}{x : \sigma \vdash^{\circ} x : \sigma}\ \mathbf{var}^{\circ}$$

In contrast, the impure variable rule is marked by a set of variables over which it is weakened, and the context must contain the variables to be weakened. The non-strict variable rule is given by:

$$\frac{}{\Gamma, x : \sigma \vdash x^{\operatorname{dom}\Gamma} : \sigma}\ \mathbf{var}$$

where $\operatorname{dom}\Gamma$ is the set of variables defined in Γ, which corresponds to a functional reading of contexts.

The strict variable rule can be simply embedded into the non-strict variable rule by a simple translation, where the strict rule $x : \sigma \vdash^{\circ} x : \sigma$ becomes:

$$\bullet, x : \sigma \vdash x^{\operatorname{dom}\bullet} : \sigma$$

which can be shown to hold once all rules have been defined.

The let-rule can now be introduced, which gives the building blocks used to define first-level programs:

$$\frac{\begin{array}{c}\Gamma \vdash^a t : \sigma \\ \Delta, x : \sigma \vdash^a u : \tau\end{array}}{\Gamma \otimes \Delta \vdash^a \mathbf{let}\, x = t \,\mathbf{in}\, u : \tau}\,\mathbf{let}$$

which is strict if the terms t and u are strict, as denoted in the rule by the presence of the strictness variable $a \in \{\circ, -\}$. A strict **let** derivation can be embedded into the non-strict case in the same was as for the variable rule.

By combining the let rule with the applicable embedding, a more convenient form of the let rule can be defined as:

$$\frac{\begin{array}{c}\Gamma \vdash^a t : \sigma \\ \Delta, x : \sigma \vdash^b u : \tau\end{array}}{\Gamma \otimes \Delta \vdash^{a \sqcap b} \mathbf{let}\ x = t\ \mathbf{in}\ u : \tau}$$

where $\circ \sqcap \circ = \circ$, and is non-strict otherwise.

Using the let rule and variable rules defined above, it can now be shown how weakenings can affect the meaning of a program. Consider the program defined as:

$$y : \mathcal{Q}_2 \vdash \mathbf{let}\, x = y \,\mathbf{in}\, x^{\{\}} : \mathcal{Q}_2$$

which is interpreted as the identity, and is free from measurement. However, a similar program, defined as:

$$y : \mathcal{Q}_2 \vdash \mathbf{let}\, x = y \,\mathbf{in}\, x^{\{y\}} : \mathcal{Q}_2$$

makes use of weakening, which is interpreted as a measurement, causing superpositions to collapse; thus it is not the identity on superpositions.

6.5.2 Products ($\otimes$)

The rules for the product $\otimes$ are the standard rules taken from linear logic. The unit for the product, denoted $\mathcal{Q}_\mathbf{1}$, carries no information, so instead of an elimination rule an implicit weakening rule is allowed in this case. The $\mathcal{Q}_\mathbf{1}$ introduction and weakening

rules are defined as:

$$\frac{}{\bullet \vdash^{\circ} () : \mathcal{Q}_1}\ \mathcal{Q}_1 \text{ intro} \qquad \frac{\Gamma, x : \mathcal{Q}_1 \vdash^{a} t : \sigma}{\Gamma \vdash^{a} t : \sigma}\ \mathcal{Q}_1 \text{ weak}$$

Note that the weakening rule for $\mathcal{Q}_\mathbf{1}$ preserves strictness, as no information is lost.

The product introduction rule follows the standard pattern, using pairs to denote products, and is defined as:

$$\frac{\Gamma \vdash^{a} t : \sigma \quad \Delta \vdash^{a} u : \tau}{\Gamma \otimes \Delta \vdash^{a} (t, u) : \sigma \otimes \tau}\ \otimes \text{ intro}$$

As with the **let** rule, the strict product introduction rule can be trivially embedded into the non-strict form.

The elimination rule for the product is a pattern-matching variant of the let rule:

$$\frac{\begin{array}{c}\Gamma \vdash^{a} t : \sigma \otimes \tau \\ \Delta,\ x : \sigma, y : \tau \vdash^{a} u : \rho\end{array}}{\Gamma \otimes \Delta \vdash^{a} \mathbf{let}\ (x, y) = t\ \mathbf{in}\ u : \rho}\ \otimes \text{ elim}$$

which again admits embedding. Via the trivial embedding, the following variation of the pair elimination rule can be defined:

$$\frac{\begin{array}{c}\Gamma \vdash^{a} t : \sigma \otimes \tau \\ \Delta, x : \sigma, y : \tau \vdash^{b} u : \rho\end{array}}{\Gamma \otimes \Delta \vdash^{a \sqcap b} \mathbf{let}\ (x, y) = t\ \mathbf{in}\ u : \rho}$$

The pair introduction and let-pair elimination rules can be used to implement the swap operations discussed in section 6.2. The first presented approximation of *swap* is based on multiplicative rules that measure the qubits whilst swapping them; this can be implemented in QML as:

$$p : \mathcal{Q}_2 \otimes \mathcal{Q}_2 \vdash \mathbf{let}\ (x, y) = p\ \mathbf{in}\ (y^{\{p\}}, x^{\{p\}}) : \mathcal{Q}_2 \otimes \mathcal{Q}_2$$

where the pair p is weakened, and hence its components are measured. The second *swap* operation, which has the correct semantics of swapping without measurement,

is given by the following QML program:

$$p : \mathcal{Q}_2 \otimes \mathcal{Q}_2 \vdash \textbf{let } (x, y) = p \textbf{ in } (y^{\{\}}, x^{\{\}}) : \mathcal{Q}_2 \otimes \mathcal{Q}_2$$

where there is no weakening, as the variables are marked with the empty set, and hence there is no measurement.

6.5.3 Conditionals (**if** and **if**$^\circ$)

As already discussed, in QML there are two kinds of conditional operation, and both are variations of the standard *if... then... else...* construct. The first, denoted **if**, measures the conditional qubit c, and hence performs classical control:

$$\frac{\begin{array}{c}\Gamma \vdash c : \mathcal{Q}_2 \\ \Delta \vdash t, u : \sigma\end{array}}{\Gamma \otimes \Delta \vdash \textbf{if } c \textbf{ then } t \textbf{ else } u : \sigma}\ \textbf{if}$$

The rule for the classical control operator **if** is similar to the usual rule for *if*, and can be used to implement a form of negation:

$$\begin{array}{l} mnot \quad : \mathcal{Q}_{\mathbf{2}} \multimap \mathcal{Q}_{\mathbf{2}} \\ mnot\ x = \textbf{if } x \textbf{ then qfalse} \\ \qquad\qquad\quad \textbf{else qtrue} \end{array}$$

As this program measures the qubit x it does not swap the probability amplitudes of the qubit, as might be expected of a quantum negation operator, and only acts as true negation on the computational basis states of either **qtrue** or **qfalse**. This is termed the *classical–if* and gives classical control. To avoid the measurement, and to implement a quantum negation operation, a measurement-free version of *if* must be used. The measurement-free, or strict, *if* rule, called **if**$^\circ$, is introduced by the following rule:

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

The rule for **if**$^\circ$ relies upon the orthogonality judgement $t \perp u$, which is defined for terms of the same type and context, $\Gamma \vdash t, u : \alpha$, and is fully discussed in section

6.5.5. Intuitively, $t \perp u$ holds if the outputs of t and u are always orthogonal. The $\mathbf{if}^\circ$ operation is termed the *quantum–if* or *strict–if*, as it provides quantum control, and produces no garbage (and hence there is no measurement). The strict $\mathbf{if}^\circ$ derivation given above can also be simply embedded into a non-strict derivation.

Using the measurement-free version $\mathbf{if}^\circ$, standard reversible and hence quantum operations such as *qnot* can be implemented:

$$\begin{aligned} &\mathit{qnot} : \mathcal{Q}_\mathbf{2} \multimap \mathcal{Q}_\mathbf{2} \\ &\mathit{qnot}\ x = \mathbf{if}^\circ\ x\ \mathbf{then}\ \mathbf{qfalse} \\ &\qquad\qquad\qquad\quad \mathbf{else}\ \ \mathbf{qtrue} \end{aligned}$$

and the conditional-not *cnot*:

$$\begin{aligned} &\mathit{cnot} : \mathcal{Q}_\mathbf{2} \multimap \mathcal{Q}_\mathbf{2} \multimap \mathcal{Q}_\mathbf{2} \otimes \mathcal{Q}_\mathbf{2} \\ &\mathit{cnot}\ c\ x = \mathbf{if}^\circ\ c\ \mathbf{then}\ (\mathbf{qtrue}, \mathit{qnot}\ x) \\ &\qquad\qquad\qquad\quad \mathbf{else}\ \ (\mathbf{qfalse}, x) \end{aligned}$$

and finally the Toffoli operator, discussed in section 2.4.1, which is essentially a conditional-*cnot*:

$$\begin{aligned} &\mathit{toff} : \mathcal{Q}_\mathbf{2} \multimap \mathcal{Q}_\mathbf{2} \multimap \mathcal{Q}_\mathbf{2} \multimap \mathcal{Q}_\mathbf{2} \otimes (\mathcal{Q}_\mathbf{2} \otimes \mathcal{Q}_\mathbf{2}) \\ &\mathit{toff}\ c\ x\ y = \mathbf{if}^\circ\ c\ \mathbf{then}\ (\mathbf{qtrue}, \mathit{cnot}\ x\ y) \\ &\qquad\qquad\qquad\quad \mathbf{else}\ \ (\mathbf{qfalse}, (x, y)) \end{aligned}$$

6.5.4 Superpositions

In order to exploit quantum parallelism, it has to be possible to create superpositions such as $\mathbf{qtrue} + \mathbf{qfalse}$, which is actually a shorthand for $(\frac{1}{\sqrt{2}}) \times \mathbf{qtrue} + (\frac{1}{\sqrt{2}}) \times \mathbf{qfalse}$. If one of the coefficients is zero then that term may be omitted, for example $(-1) \times \mathbf{qtrue}$ is shorthand for $(-1) \times \mathbf{qtrue} + 0 \times \mathbf{qfalse}$, which constructs a qubit which behaves like **qtrue** if measured, but has a different phase.

The rule for generating a superposition of terms is defined as:

$$\frac{\begin{array}{l}\Gamma \vdash^\circ t, u : \sigma \\ |\lambda|^2 + |\lambda'|^2 = 1 \qquad t \perp u\end{array}}{\Gamma \vdash^\circ (\lambda) \times t + (\lambda') \times u}\ \mathbf{sup}^\circ$$

where $\lambda, \lambda' \in \mathbb{C}$ and describe the complex-valued probability amplitudes of the superposition. The condition $|\lambda|^2 + |\lambda'|^2 = 1$ ensures that the coefficients describe a probability distribution.

The QML implementation of the Hadamard operator makes use of this rule in the branches of its conditional:

$$\begin{aligned} &\mathit{had} \quad : \mathcal{Q}_\mathbf{2} \multimap \mathcal{Q}_\mathbf{2} \\ &\mathit{had}\ x = \mathbf{if}^\circ\ x\ \mathbf{then}\ (-1) \times \mathbf{qtrue} + \mathbf{qfalse} \\ &\qquad\qquad\quad \mathbf{else}\ \ \mathbf{qtrue} + \mathbf{qfalse} \end{aligned}$$

which, as already noted, omits simple normalisation factors that can be inferred. Section 6.5.5 discusses the orthogonality judgement $t \perp u$, and in this case it is clear that $\mathbf{qtrue} \perp \mathbf{qfalse}$, hence the use of $\mathbf{if}^\circ$ is permitted.

The binary rule for superpositions presented above is sufficient to produce any n-ary superposition. Superpositions of more than two terms are created by nesting of the superposition rule, for example $(t + u + v)$ would become $(\frac{1}{3} \times t + (\frac{2}{3} \times (u + v)))$.

The constants **qtrue** and **qfalse** can also allow weakening to occur, in a similar manner to the method by which the non-strict variable rule allows weakening, defined in section 6.5.1. If the constants are not marked with a set of variables to be weakened over, then it is assumed that the set is empty. This rule also allows weakening over single qubit superpositions, interpreting $\mathbf{qfalse}^{\vec{x}} + \mathbf{qtrue}^{\vec{y}}$ as $(\mathbf{qfalse} + \mathbf{qtrue})^{\vec{x}+\vec{y}}$. The non-strict qubit introduction rule is given as:

$$\frac{|\lambda|^2 + |\lambda'|^2 = 1}{\Gamma \vdash (\lambda \times \mathbf{qfalse} + \lambda' \times \mathbf{qtrue})^{\operatorname{dom}\Gamma} : \mathcal{Q}_2}\ \mathbf{sup}$$

where $\operatorname{dom}\Gamma$ is the set of variables defined in Γ. The embedding rule is also applicable in this case, as the strict rule, restricted to qubits, can be embedded into the non-strict rule by a simple application of the non-strict rule, as in the case of **var**.

6.5.5 Orthogonality ($\perp$)

The type system presented so far correctly tracks the use of variables and prevents variables from being weakened inappropriately; 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

$$\textbf{if}^\circ\ x\ \textbf{then qtrue else qtrue}$$

This expression appears, syntactically at least, to use the variable x. However, given the semantics of $\textbf{if}^\circ$, which returns a superposition of the branches, the expression in this case returns **qtrue** without using any information about x. In order to maintain the invariant that all measurements are explicit, the type system should reject the above expression. More precisely, the expression $\textbf{if}^\circ\ x\ \textbf{then}\ t\ \textbf{else}\ u$ 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. The branches of a superposition should also be orthogonal for similar reasons, and this is why the typing rules for superpositions (sup) and the quantum conditional ($\textbf{if}^\circ$) include the judgement $t \perp u$.

Mathematically, two terms, t, u, are orthogonal if their inner-product is equal to zero, $\langle t|u\rangle = 0$. If this is the case then the judgement $t \perp u$ is true, but if the inner-product yields any other value then t is not orthogonal to u. In the presentation of an equational theory for (the pure fragment of) QML [2] (published in QPL 2005 [71]), the orthogonality judgements are replaced by an inner-product judgement on terms, to much the same effect. However, the inner-product approach appears to be more informative and flexible, and gives a method of reasoning about orthogonality, hence in future versions of QML this method may be adopted for all terms. Two QML terms are orthogonal if their inner-product, defined on terms in figure 6.5, is equal to zero. If the inner-product is greater than zero, or undefined, then the terms are defined to be non-orthogonal.

The inner-product judgements defined over QML terms in figure 6.5 can be approximated by the following rules:

$$\frac{}{\textbf{qtrue} \perp \textbf{qfalse}} \qquad \frac{}{\textbf{qfalse} \perp \textbf{qtrue}}$$

$$
\begin{array}{rcl@{\qquad}rcl}
\langle \mathrm{t}|\mathrm{t}\rangle & = & 1 \text{ if } \mathrm{t} \neq \mathbf{0} & \langle \lambda \times t + \lambda' \times t' \mid u\rangle & = & \lambda^* \times \langle t|u\rangle + \lambda'^* \times \langle t'|u\rangle \\
\langle \mathbf{qfalse}|\mathbf{qtrue}\rangle & = & 0 & \langle t \mid \kappa \times u + \kappa' \times u'\rangle & = & \kappa \times \langle t|u\rangle + \kappa' \times \langle t|u'\rangle \\
\langle \mathbf{qtrue}|\mathbf{qfalse}\rangle & = & 0 & & & \\
 & & & \langle \lambda \times t|u\rangle & = & \lambda^* \langle t|u\rangle \\
\langle \mathbf{0}|\mathbf{qtrue}\rangle & = & 0 = \langle \mathbf{qtrue}|\mathbf{0}\rangle & \langle t|\lambda \times u\rangle & = & \lambda \langle t|u\rangle \\
\langle \mathbf{0}|\mathbf{qfalse}\rangle & = & 0 = \langle \mathbf{qfalse}|\mathbf{0}\rangle & \langle t + t'|u\rangle & = & \langle t|u\rangle + \langle t'|u\rangle \\
\langle \mathbf{0}|x\rangle & = & 0 = \langle x|\mathbf{0}\rangle & \langle t|u + u'\rangle & = & \langle t|u\rangle + \langle t|u'\rangle \\
 & & & & & \\
\langle (t,t') \mid (u,u')\rangle & = & \langle t|u\rangle \times \langle t'|u'\rangle & \langle t|u\rangle & = & \textit{undefined} \text{ otherwise}
\end{array}
$$

FIGURE 6.5: Inner-products and orthogonality of terms

$$\frac{t \perp u}{(t,v) \perp (u,w)} \perp \mathbf{pair}_0 \qquad \frac{t \perp u}{(v,t) \perp (w,u)} \perp \mathbf{pair}_1$$

$$\frac{t \perp u \qquad \lambda_0^* \kappa_0 = -\lambda_1^* \kappa_1}{\lambda_0 \times t + \lambda_1 \times u \perp \kappa_0 \times t + \kappa_1 \times u} \perp \mathbf{sup}$$

The first two axioms above state that the basic states of **qtrue** and **qfalse** are orthogonal. The third and fourth rule state that pairs of terms can be orthogonal, so long as one component of the pair is orthogonal to the corresponding component in the other pair. The final rule defines when superpositions of terms can be orthogonal.

As an example, these rules are used in the typing of the Hadamard operation discussed earlier. The use of $\mathbf{if}^\circ$ in QML programs is valid only if the two branches are orthogonal, hence, for the Hadamard operation, it is required that:

$$((-1) \times \mathbf{qtrue} + \mathbf{qfalse}) \perp (\mathbf{qtrue} + \mathbf{qfalse})$$

which, with normalisation factors reintroduced, is shorthand for the term:

$$((-\frac{1}{\sqrt{2}}) \times \mathbf{qtrue} + \frac{1}{\sqrt{2}} \times \mathbf{qfalse}) \perp (\frac{1}{\sqrt{2}} \times \mathbf{qtrue} + \frac{1}{\sqrt{2}} \times \mathbf{qfalse})$$

This can be found to be orthogonal by application of the $\perp$ **sup** rule.

The rules for orthogonality given so far are incomplete, as is shown by the final *undefined* case in figure 6.5. For example, conditional expressions are not allowed to be

orthogonal, even though semantically this may be the case. Orthogonality judgements for strict conditionals can be added by considering the extra inner-product rules shown in figure 6.6, which can be approximated by the following (less informative)

$$\begin{array}{rcl} \langle t | \mathbf{if}^\circ\ (\alpha \times \mathbf{qfalse} + \beta \times \mathbf{qtrue})\ \mathbf{then}\ u\ \mathbf{else}\ u' \rangle & = & \beta \langle t|u\rangle + \alpha \langle t|u'\rangle \\ \langle \mathbf{if}^\circ\ (\alpha \times \mathbf{qfalse} + \beta \times \mathbf{qtrue})\ \mathbf{then}\ u\ \mathbf{else}\ u' | t \rangle & = & \beta \langle u|t\rangle + \alpha \langle u'|t\rangle \end{array}$$

FIGURE 6.6: Inner-products and orthogonality of $\mathbf{if}^\circ$ terms

orthogonality judgements:

$$\frac{t \perp u \qquad t' \perp u'}{t \perp \mathbf{if}^\circ\ c\ \mathbf{then}\ u\ \mathbf{else}\ u'} \perp \mathbf{if}_0^\circ \qquad \frac{t \perp u \qquad t' \perp u'}{\mathbf{if}^\circ\ c\ \mathbf{then}\ u\ \mathbf{else}\ u' \perp t} \perp \mathbf{if}_1^\circ$$

The orthogonality judgements may be extended in this, and other, ways by adding rules in future.

6.6 QML Programs

In QML, programs are definitions of terms in contexts. Passing the programs explicitly through the rules has been omitted, as discussed in section 6.5. The axioms also require that the program is well-typed. Well-typed programs can be constructed by the following rule:

$$\frac{\vdash \vec{d} \qquad \vec{d};\ \Gamma \vdash t : \sigma}{\vdash \vec{d}, f\ \Gamma = t : \sigma}$$

Note that recursion is not allowed, unlike Selinger's QPL [69]. See section 8.3 for a brief discussion of how recursion could be interpreted in QML.

Previously defined functions can be used via the application rule:

$$\frac{\vec{d} = \vec{d}', f\ (x_1 : \sigma_1, .., x_n : \sigma_n) = t : \tau \quad \vec{d}; \Gamma \vdash t_1 : \sigma_1, \ldots t_n : \sigma_n}{\vec{d}; \Gamma \vdash f\ \vec{t} : \tau}\ \text{app}$$

Two QML programs will now be presented as examples: a variant of the Deutsch algorithm, and an encoding of quantum teleportation in QML.

6.6.1 The Deutsch Algorithm in QML

The Deutsch algorithm is presented in section 3.9 as an method of discovering whether a classical function on Booleans is constant, by querying that function only once. To avoid having to resort to higher-order oracles, the algorithm presented here solves the analogous problem of deciding whether two qubits, which are assumed to be classical, are equal, with the property that each branch of the program only queries one of the input bits. This non-oracle variation of the Deutsch algorithm can be implemented in QML as the following program:

$$
\begin{aligned}
&\mathit{deutsch} : \mathcal{Q}_{\mathbf{2}} \multimap \mathcal{Q}_{\mathbf{2}} \multimap \mathcal{Q}_{\mathbf{2}} \\
&\mathit{deutsch}\ a\ b = \mathbf{let}\ (x, y) = \\
&\qquad\qquad \mathbf{if}^{\circ}\ \mathbf{qfalse} + \mathbf{qtrue} \\
&\qquad\qquad \mathbf{then}\ (\mathbf{qtrue}, \mathbf{if}^{\circ}\ a \\
&\qquad\qquad\qquad\qquad \mathbf{then}\ (\mathbf{qfalse} + (-1) \times \mathbf{qtrue}, (\mathbf{qtrue}, b)) \\
&\qquad\qquad\qquad\qquad \mathbf{else}\ \ ((-1) \times \mathbf{qfalse} + \mathbf{qtrue}, (\mathbf{qfalse}, b))) \\
&\qquad\qquad \mathbf{else}\ (\mathbf{qfalse},\ \mathbf{if}^{\circ}\ b \\
&\qquad\qquad\qquad\qquad \mathbf{then}\ ((-1) \times \mathbf{qfalse} + \mathbf{qtrue}, (a, \mathbf{qtrue})) \\
&\qquad\qquad\qquad\qquad \mathbf{else}\ \ (\mathbf{qfalse} + (-1) \times \mathbf{qtrue}, (a, \mathbf{qfalse}))) \\
&\qquad\qquad \mathbf{in}\ \mathit{had}\ x
\end{aligned}
$$

where $\mathit{had}\ x$ is the previously defined Hadamard operator.

The program above is complicated by the need to store both input qubits in a temporary structure computed by $\mathbf{if}^{\circ}$, which is actually unnecessary; it can be assumed that these bits are classical, and hence they can be used without further measurement. If QML was extended to include classical data, this program could be simplified as shown in figure 6.7. Classical bits could be incorporated into QML using structures similar to those used in Selinger's QPL [69]. Extensions to QML such as these are discussed in chapter 9.

The Deutsch algorithm is in fact related to the Shor algorithm, by the quantum Fourier transform (QFT). The Hadamard operations that prepare the necessary superpositions in the Deutsch algorithm are an example of the QFT. The Shor algorithm makes use of the QFT (actually the inverse: $\mathrm{QFT}^{\dagger}$) in a more complicated manner.

$$
\begin{aligned}
&\mathit{deutsch} : \mathbb{N}_2 \multimap \mathbb{N}_2 \multimap \mathcal{Q}_{\mathbf{2}} \\
&\mathit{deutsch}\ a\ b = \mathbf{let}\ (x, y) = \mathbf{if}^{\circ}\ \mathbf{qfalse} + \mathbf{qtrue} \\
&\qquad\qquad\qquad\qquad \mathbf{then}\ (\mathbf{qtrue}, \mathbf{if}\ a \\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{then}\ \mathbf{qfalse} + (-1) \times \mathbf{qtrue} \\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{else}\ \ (-1) \times \mathbf{qfalse} + \mathbf{qtrue}) \\
&\qquad\qquad\qquad\qquad \mathbf{else}\ (\mathbf{qfalse},\ \mathbf{if}\ b \\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{then}\ (-1) \times \mathbf{qfalse} + \mathbf{qtrue} \\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{else}\ \ \mathbf{qfalse} + (-1) \times \mathbf{qtrue}) \\
&\qquad\qquad \mathbf{in}\ \mathit{had}\ x
\end{aligned}
$$

FIGURE 6.7: The Deutsch Algorithm implemented in QML, extended with classical data

Shor's algorithm and the QFT are presented in appendix A, with a detailed analysis of both of these techniques, and an implementation of the QFT in QML is defined.

6.6.2 Quantum teleportation

The quantum teleport protocol, described in section 3.10, allows a qubit to be *teleported* to a partner with whom an EPR pair has previously been shared, using only two bits of classical information. The separation of the partners and the classical computation cannot currently be formalised in QML, but a function *tel*, which encodes what happens to the teleported qubit, can be implemented. The correctness of the teleport protocol can be verified by showing that *tel* is extensionally equivalent to the identity function.

As an auxiliary, the Pauli-Z operation is implemented in QML as:

$$
\begin{aligned}
&z : \mathcal{Q}_{\mathbf{2}} \multimap \mathcal{Q}_{\mathbf{2}} \\
&z\ x = \mathbf{if}\ x\ \mathbf{then}\ (-1) \times \mathbf{qtrue} \\
&\qquad\qquad \mathbf{else}\ \ \mathbf{qfalse}
\end{aligned}
$$

Making use of this auxiliary function, the QML program *tel* can be implemented as:

$$
\begin{aligned}
&\mathit{tel} : \mathcal{Q}_{\mathbf{2}} \multimap \mathcal{Q}_{\mathbf{2}} \\
&\mathit{tel}\ x = \mathbf{let}\ (a, b)\ \ = (\mathbf{qfalse}, \mathbf{qfalse}) + (\mathbf{qtrue}, \mathbf{qtrue})
\end{aligned}
$$

$$
\begin{aligned}
(a', x') &= \mathit{cnot}\ a\ x \\
b' &= \textbf{if}\ a'\ \textbf{then}\ \mathit{qnot}\ b\ \textbf{else}\ b \\
b'' &= \textbf{if}\ \mathit{had}\ x'\ \textbf{then}\ z\ b'\ \textbf{else}\ b' \\
\textbf{in}\ \ b''
\end{aligned}
$$

This is a QML implementation of the circuit given for the teleportation algorithm in section 3.10. The language CQP of Communicating Quantum Processes [28] allows the modelling of channels, and the separation of the two components of the EPR pair, in an implementation of the teleportation algorithm.

6.7 Coproducts in QML

In a previous version of QML (presented in reference [3], referred to as QML$^{\oplus}$ for the remainder of this section) the language included the notion of a tensorial coproduct, denoted by $\oplus$. The types of QML$^{\oplus}$ were generated by $\mathcal{Q}_{\mathbf{1}}$, $\sigma \otimes \tau$, and $\sigma \oplus \tau$, where σ and τ quantify types. Qubits were not primitive, but defined as $\mathcal{Q}_{\mathbf{2}} = \mathcal{Q}_{\mathbf{1}} \oplus \mathcal{Q}_{\mathbf{1}}$. The coproduct allows any finite type to be directly represented in this way; not just limited to $\mathcal{Q}_{\mathbf{2}}$. The introduction rules used for $\oplus$ were the usual coproduct rules, a left and a right injection:

$$\frac{\Gamma \vdash^{a} s : \sigma}{\Gamma \vdash^{a} \mathrm{inl}\ s : \sigma \oplus \tau}\ \oplus\mathrm{intro_l} \qquad \frac{\Gamma \vdash^{a} t : \tau}{\Gamma \vdash^{a} \mathrm{inr}\ t : \sigma \oplus \tau}\ \oplus\mathrm{intro_r}$$

The coproduct type was interpreted as $\sigma \oplus \tau = \mathcal{Q}_{\mathbf{2}} \otimes |\sigma \sqcup \tau|$, where $|\sigma \sqcup \tau|$ could store a value of either $|\sigma|$ or $|\tau|$, by padding the smaller type. Using the coproduct and injection rules, **qfalse** and **qtrue** were defined in QML$^{\oplus}$ as:

$$\textbf{qtrue} = \mathrm{inl}() : \mathcal{Q}_2 \qquad \textbf{qfalse} = \mathrm{inr}() : \mathcal{Q}_2$$

omitting the weakening property of QML$^{\oplus}$.

Instead of primitive **if** and **if**$^{\circ}$ rules, QML$^{\oplus}$ implemented two $\oplus$-elimination rules: **case**, which provided classical-control and is a generalisation of **if**; and a quantum-control operation **case**$^{\circ}$, which generalised **if**$^{\circ}$. The measuring $\oplus$-elimination rule is

similar to the standard coproduct elimination rule, and is given as:

$$\frac{\begin{array}{c}\Gamma \vdash c : \sigma \oplus \tau \\ \Delta,\, x : \sigma \vdash t : \rho \\ \Delta,\, y : \tau \vdash u : \rho\end{array}}{\Gamma \otimes \Delta \vdash \mathbf{case}\ c\ \mathbf{of}\ \{\mathbf{inl}\ x \Rightarrow t \mid \mathbf{inr}\ y \Rightarrow u\} : \rho}\ \oplus\text{elim}$$

The **if** rule would then be derived from this as:

$$\mathbf{if}\ b\ \mathbf{then}\ t\ \mathbf{else}\ u = \mathbf{case}\ b\ \mathbf{of}\ \{\mathbf{inl}\ \_ \Rightarrow t \mid \mathbf{inr}\ \_ \Rightarrow u\}$$

The strict case is introduced by:

$$\frac{\begin{array}{c}\Gamma \vdash^{a} c : \sigma \oplus \tau \\ \Delta,\ x : \sigma \vdash^{\circ} t : \rho \\ \Delta,\ y : \tau \vdash^{\circ} u : \rho \quad t \perp u\end{array}}{\Gamma \otimes \Delta \vdash^{a} \mathbf{case}^{\circ}\ c\ \mathbf{of}\ \{\mathbf{inl}\ x \Rightarrow t \mid \mathbf{inr}\ y \Rightarrow u\} : \rho}\ \oplus\text{elim}^{\circ}$$

and from this the measurement-free $\mathbf{if}^{\circ}$ can be derived:

$$\mathbf{if}^{\circ}\ b\ \mathbf{then}\ t\ \mathbf{else}\ u = \mathbf{case}^{\circ}\ b\ \mathbf{of}\ \{\mathbf{inl}\ \_ \Rightarrow t \mid \mathbf{inr}\ \_ \Rightarrow u\}$$

The problem with coproducts, and hence $\text{QML}^{\oplus}$, is that the branches of a $\mathbf{case}^{\circ}$ operation can be of different size, and this was dealt with in the semantics of $\text{QML}^{\oplus}$ by padding the type of the smaller branch. The padding of one type in this way could lead the garbage becoming entangled with the useful output in some way. This could happen, for example, by branching over $\mathcal{Q}_1 \otimes \mathcal{Q}_2$. The garbage, which is created by padding, may indirectly measure the qubit which is being branched over. Consequently, this approach is not compositional, and is therefore rejected.

In the version of QML presented in this thesis, this problem does not occur as the coproduct has been removed, with qubits now primitive types, and **case** and $\mathbf{case}^{\circ}$ have been replaced by their simpler derivations of **if** and $\mathbf{if}^{\circ}$, and the strict $\mathbf{if}^{\circ}$ does admit for any garbage to be produced. In future versions of QML coproducts may be reintroduced, possibly limited to classical types to remove the problem of the unexpected decoherence; refer to the future work section 8.4.

6.8 Summary

This chapter has introduced the functional quantum programming language QML. The design and motivation for the language is first explored, followed by the definitions of the syntax of QML, and QML functions and programs. The structural and typing rules of QML terms are also presented, including the rules governing the quantum conditional $\mathbf{if}^{\circ}$, which requires an orthogonality judgement that guarantees the branches are observably different. This chapter includes two QML implementations of quantum algorithms that have been discussed in earlier chapters, and closes with a discussion of a variation of **QML** that included coproducts, and why they are not included in this presentation.

Chapter 7

Operational semantics of QML

An operational semantics of a programming language such as QML describes how a program is interpreted as a sequence of computational steps. In the case of QML, the operational semantics presented here is defined by presenting a translation of QML derivations to morphisms in the category **FQC**. Morphisms in the category **FQC** are expressed as quantum circuits, which gives the computational steps required to compute a QML program. This is achieved by introducing a category of QML terms, and then defining a mapping from the category **QML** to the category **FQC** for each QML term. In this way the operational semantics of QML provides an implementation of the language in the well understood physical model of quantum circuits. This is slightly different from the usual view of an operational semantics as a form of term rewriting system; in this thesis the term operational semantics is understood to mean a (theoretically) realisable semantics, describing a system which can be physically executed to produce an output: a sequence of quantum gates.

A denotational semantics for QML, factored through the operational semantics, is also presented. For every QML term, the denotational semantics gives a superoperator in the category **Q**, via the **FQC** morphism produced by the operational semantics. The denotational semantics allows mathematical tools to be used to reason about programs and optimisations, and allows a definition of extensional equality of QML terms.

A Haskell implementation of the operational semantics is also discussed, which

acts as a compiler for QML programs. A current implementation[1] of the compiler can be found at the project website [30].

7.1 The category of QML terms

The set of typed QML terms can be organised in a categorical structure. The objects of this category are contexts, and the homset between the objects Γ and Δ, denoted $\mathbf{QML}\ \Gamma\ \Delta$, consists of all the terms t such that $\Gamma \vdash t : \overline{\Delta}$, where the operation $\bar{\cdot}$ on contexts views the context as a type. The mapping from context to types is naturally defined as:

$$\begin{aligned} \overline{\bullet} &= \mathcal{Q}_1 \\ \overline{\Gamma, x:\sigma} &= \overline{\Gamma} \otimes \sigma \end{aligned}$$

For every context Γ, the identity morphism $I_\Gamma \in \mathbf{QML}\ \Gamma\ \Gamma$ is defined as:

$$\begin{aligned} I_\bullet &= () \\ I_{\Gamma, x:\sigma} &= (I_\Gamma, x) \end{aligned}$$

In order to express composition in the category **QML**, the following auxiliary definition is required:

$$\mathbf{let}^*\ \bullet = u\ \mathbf{in}\ t \qquad \equiv t$$

$$\mathbf{let}^*\ \Gamma, x:\sigma = u\ \mathbf{in}\ t \equiv \mathbf{let}\ (x_r, x) = u\ \mathbf{in}\ \mathbf{let}^*\ \Gamma = x_r\ \mathbf{in}\ t$$

Using the definition of $\mathbf{let}^*$, given $u \in \mathbf{QML}\ \Gamma\ \Delta$ and $t \in \mathbf{QML}\ \Delta\ \Theta$, the composition $t \circ u \in \mathbf{QML}\ \Gamma\ \Theta$ is given by the term $\mathbf{let}^*\ \Gamma = u\ \mathbf{in}\ t$.

The category $\mathbf{QML}^\circ$ is defined as a subcategory of **QML**, and consists of all the strict terms t; $\mathbf{QML}^\circ\ \Gamma\ \Delta = \{\, t | \Gamma \vdash^\circ t : \overline{\Delta} \}$

The structure of the category **QML** is used extensively by Altenkirch, Grattage, Vizzotto and Sabry in the development of a sound and complete equational theory for a pure fragment of QML [2].

[1] The code for the compiler is currently under development, as both QML and the operational semantics are active areas of research. Refer to section 8.4.

7.2 Interpretation of judgements

In order to be able to interpret judgements, the sizes of types and contexts must first be defined. The function $|\cdot|$ gives the size of a type:

$$\begin{aligned} |\mathcal{Q}_\mathbf{1}| &= 0 \\ |\mathcal{Q}_\mathbf{2}| &= 1 \\ |\sigma \otimes \tau| &= |\sigma| \otimes |\tau| \\ &= |\sigma| + |\tau| \end{aligned}$$

Contexts correspond to the tensor product of their component types, hence:

$$|\Gamma| = |\overline{\Gamma}|$$

where $\bar{\cdot}$ maps context to types, as defined in section 7.1. The size function $|\cdot|$ will be frequently omitted, and just Γ will be written for $|\Gamma|$, and σ written for $|\sigma|$. It is clear from the context which is intended. The interpretations of a context Γ and a type σ are therefore given by:

$$\begin{aligned} [\![\Gamma]\!] &= |\Gamma| \\ [\![\sigma]\!] &= |\sigma| \end{aligned}$$

There are two kinds of derivations in QML: strict programs in $\mathbf{QML}^\circ$, and non-strict, or impure programs in $\mathbf{QML}$. These can both be given an operational semantics by translating derivations into the appropriate category of quantum computations: $\mathbf{FQC}^\circ$ in the strict case, and otherwise $\mathbf{FQC}$. Strict QML derivations $\frac{d}{\Gamma \vdash^\circ t:\sigma}$ are interpreted by the semantic function given by:

$$[\![d]\!]^\circ_{\mathrm{Op}} \in \mathbf{QML}^\circ\ \Gamma\ \sigma \to \mathbf{FQC}^\circ\ \Gamma\ \sigma$$

and non-strict QML derivations $\frac{d}{\Gamma \vdash^\circ t:\sigma}$ are interpreted by:

$$[\![d]\!]_{\mathrm{Op}} \in \mathbf{QML}\ \Gamma\ \sigma \to \mathbf{FQC}\ \Gamma\ \sigma.$$

Given $\Gamma \vdash^\circ t : \sigma$ and $\Gamma' \vdash^\circ u : \sigma$ a derivation $\frac{d}{t \perp u}$ is interpreted as a structure $[\![d]\!]^\perp_{\mathrm{Op}} = (c, l, r, \psi)$, where:

$c \in \mathbb{N}$

$l \in \mathbf{FQC}^{\circ}\ \Gamma\ c$

$r \in \mathbf{FQC}^{\circ}\ \Gamma'\ c$

$\psi \in \mathbf{FQC}^{\simeq}\ \sigma\ (\mathcal{Q}_{\mathbf{2}} \otimes c)$

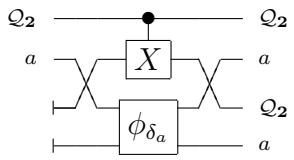
The semantics of programs $\vdash \vec{d}$ is given by an assignment of circuits to function names, following the standard interpretation. What is interesting is the interpretation of terms, and operations on the contexts of terms.

7.3 Interpreting operations on contexts

Interpreting the operation $\otimes$ on contexts is not as straightforward as it may fist appear. Using $\Gamma \otimes \Delta$ allows a variable to be used several times, allowing contraction. This is interpreted using

$$\delta = (\mathcal{Q}_2, \phi_\delta) \in \mathbf{FQC}^{\circ}\ \mathcal{Q}_2\ (\mathcal{Q}_2 \otimes \mathcal{Q}_2)$$

where $\phi_\delta = \mathrm{id}|qnot$, which is the conditional-not operation. The operation δ can be iterated to contract registers of any size: given $a \in \mathbb{N}$, the operation $\delta_a \in \mathbf{FQC}^{\circ}\ a\ (a \otimes a)$ is defined in the base case as $\delta_0 = \mathit{wires}\ \mathrm{id}$. Otherwise, $\delta_{a \otimes \mathcal{Q}_2}$ is constructed from δ_a by the following circuit:

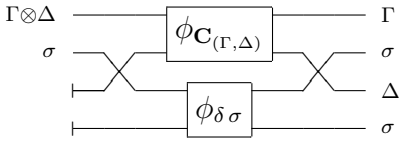


Given Γ, Δ such that $\Gamma \otimes \Delta$ is well-defined, the context-splitting operation $\mathbf{C}$ can be constructed as:

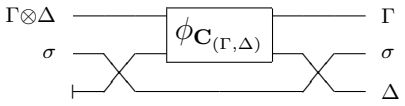
$$\mathbf{C}_{(\Gamma,\Delta)} \in \mathbf{FQC}^{\circ}\ |\Gamma \otimes \Delta|\ (|\Gamma| \otimes |\Delta|)$$

by induction over the definition of $\Gamma \otimes \Delta$. The explicit use of $|\cdot|$ in this formula is essential, as the definition of $\otimes$ is different depending on whether it acts on contexts or on their sizes. The action of $\mathbf{C}$ is defined by the following case analysis:

$\Gamma, x : \sigma \otimes \Delta, x : \sigma = (\Gamma \otimes \Delta), x : \sigma$



$\Gamma, x : \sigma \otimes \Delta = (\Gamma \otimes \Delta), x : \sigma$**, if** $x \notin$ **dom** Δ



$\bullet \otimes \Delta = \Delta$

7.4 Interpreting QML rules in FQC

The compilation of rules into **FQC** morphisms is presented using the circuit diagrams defined in section 5.2. The interpretation of the embedding of strict terms into non strict terms is invisible as a circuit, as $\mathbf{FQC}^{\circ}\ a\ b \subseteq \mathbf{FQC}\ a\ b$. Explicit references to the heap, h, and garbage, g, are omitted for clarity, and the circuits refer to the reversible circuit ϕ_t arising from the interpretation of $\frac{d}{\Gamma \vdash t : \sigma}$, $[\![d]\!]_{\mathrm{Op}} \in \mathbf{FQC}\ \Gamma\ \sigma$. These conventions will be followed in all interpretations and their diagrams.

7.4.1 Interpreting the structural rules

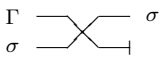
The structural rules of QML are given in section 6.5.1, and the circuits arising from these rules are presented now. The strict variable rule is interpreted in **FQC** as the identity morphism:

$$\frac{}{x : \sigma \vdash^{\circ} x : \sigma}\ \mathbf{var}^{\circ}$$

Note that this interpretation gives a strict morphism. The non-strict variable rule has a similar interpretation, with the variable passing straight through, but with

all the variables in the context being moved into the garbage, which is the **FQC** interpretation of weakening:

$$\frac{}{\Gamma, x : \sigma \vdash x^{\operatorname{dom}\Gamma} : \sigma}\ \mathbf{var}$$



The operational semantics of the variable rules are therefore given as the **FQC** morphisms $\textsc{var}^{\circ}_{\mathrm{Op}}$ and $\textsc{var}_{\mathrm{Op}}$, where $[\![x : \sigma \vdash^{\circ} x : \sigma]\!]^{\circ}_{\mathrm{Op}}$ is given by $\textsc{var}^{\circ}_{\mathrm{Op}} \in \mathbf{FQC}^{\circ}\ \sigma\ \sigma$, defined as $(0, 0, \phi)$, where ϕ is the first circuit above ($\mathbf{var}^{\circ}$), and where $[\![\Gamma, x : \sigma \vdash x^{\operatorname{dom}\Gamma} : \sigma]\!]_{\mathrm{Op}}$ is given by $\textsc{var}_{\mathrm{Op}} \in \mathbf{FQC}\ (\Gamma, \sigma)\ \sigma$, defined as $(0, |\Gamma|, \psi)$ where ψ is the second circuit above ($\mathbf{var}$).

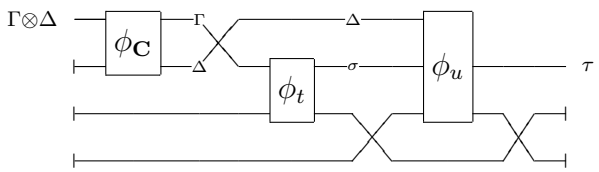
The let-rule is actually a scheme of two rules depending on value of the strictness variable $a \in \{\circ, -\}$, given by:

$$\frac{\begin{array}{c}\Gamma \vdash^{a} t : \sigma \\ \Delta, x : \sigma \vdash^{a} u : \tau\end{array}}{\Gamma \otimes \Delta \vdash^{a} \mathbf{let}\, x = t\, \mathbf{in}\, u : \tau}\ \mathbf{let}$$

The operational semantics for the let-rule, $[\![\Gamma \otimes \Delta \vdash^{a} \mathbf{let}\, x = t\, \mathbf{in}\, u : \tau]\!]^{a}_{\mathrm{Op}}$ is given by:

$$\frac{\begin{array}{c}\mathbf{t} \in \mathbf{FQC}^{a}\ \Gamma\ \sigma \\ \mathbf{u} \in \mathbf{FQC}^{a}\ (\Delta \otimes \sigma)\ \tau\end{array}}{\begin{array}{l}\textsc{let}^{a}_{\mathrm{Op}}\ \mathbf{t}\ \mathbf{u} \in \mathbf{FQC}^{a}\ (\Gamma \otimes \Delta)\ \tau \\ \textsc{let}^{a}_{\mathrm{Op}}\ \mathbf{t}\ \mathbf{u} = (h_{\mathbf{C}} + h_{\mathbf{t}} + h_{\mathbf{u}}, g_{\mathbf{t}} + g_{\mathbf{u}}, \phi)\end{array}}$$

where $[\![t]\!]^{a}_{\mathrm{Op}} = \mathbf{t}$ and $[\![u]\!]^{a}_{\mathrm{Op}} = \mathbf{u}$, which are the interpretations of the sub-terms as **FQC** morphisms, $h_{\mathbf{t}}, g_{\mathbf{t}}$ and $h_{\mathbf{u}}, g_{\mathbf{u}}$ are the heap and garbage required by $\mathbf{t}$ and $\mathbf{u}$, respectively, and $h_{\mathbf{C}}$ is the heap used by the context operation $\mathbf{C}$. ϕ, the circuit given by the **FQC** morphism $\textsc{let}^{a}_{\mathrm{Op}}$, makes use of the context operation $\mathbf{C}$, defined in section 7.3, to create the appropriate contexts for each sub-circuit:



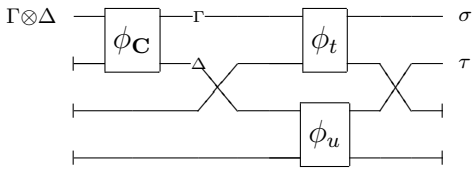
The operational semantics given by $\textsc{let}^a_{\text{Op}}$ is uniform in a, and the circuit generated is strict if both sub-derivations of the original rule are strict. This can be seen in the diagram as ϕ_t and ϕ_u are the only sources of garbage; if t is strict then ϕ_t produces no garbage, and if u is strict then ϕ_u produces no garbage. Hence, if neither sub-circuit produces any garbage, then the entire circuit is strict.

7.4.2 Interpreting products ($\otimes$)

The QML product rules are explained in section 6.5.2, with rules for $\mathcal{Q}_\mathbf{1}$ introduction and weakening, and $\otimes$ introduction and elimination. As the rules for $\mathcal{Q}_\mathbf{1}$ carry no information, their interpretation as circuits in **FQC** is invisible; they are modelled by no wires and the empty circuit.

The interpretation of $\otimes$ introduction is only slightly more interesting, as it simply merges the components of the pairing using parallel composition:

$$\frac{\Gamma \vdash^a t : \sigma \quad \Delta \vdash^a u : \tau}{\Gamma \otimes \Delta \vdash^a (t, u) : \sigma \otimes \tau} \otimes \text{intro}$$



As with the let-rule interpretation above, the contexts for the sub-circuits are prepared using the **C** operation. Note also how the circuit preserves strictness if both sub-derivations are strict, again like the let-rule interpretation. The operational semantics of strict and non-strict pairing, $[\![\Gamma \otimes \Delta \vdash^a (t, u) : \sigma \otimes \tau]\!]^a_{\text{Op}} = \textsc{pair}^a_{\text{Op}}$, is given by the following rule:

$$\frac{\mathbf{t} \in \mathbf{FQC}^a\ \Gamma\ \sigma \qquad \mathbf{u} \in \mathbf{FQC}^a\ \Delta\ \tau}{\begin{array}{l}\textsc{pair}^a_{\text{Op}}\ \mathbf{t}\ \mathbf{u} \in \mathbf{FQC}^a\ (\Gamma \otimes \Delta)\ (\sigma \otimes \tau) \\ \textsc{pair}^a_{\text{Op}}\ \mathbf{t}\ \mathbf{u} = (h_\mathbf{C} + h_\mathbf{t} + h_\mathbf{u}, g_\mathbf{t} + g_\mathbf{u}, \phi)\end{array}}$$

where $[\![t]\!]^a_{\text{Op}} = \mathbf{t}$, $[\![u]\!]^a_{\text{Op}} = \mathbf{u}$, heap and garbage are interpreted as before, and ϕ is given by the circuit above.

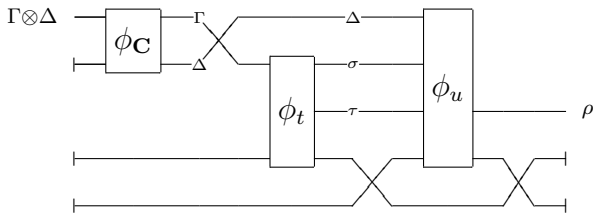
The circuit interpretation of the $\otimes$ elimination rule is almost identical to the interpretation of the let-rule, due to the fact that $\sigma \otimes \tau$ is interpreted as concatenation of wires:

$$\frac{\begin{array}{c}\Gamma \vdash^a t : \sigma \otimes \tau \\ \Delta,\, x : \sigma, y : \tau \vdash^a u : \rho\end{array}}{\Gamma \otimes \Delta \vdash^a \mathbf{let}\ (x, y) = t\ \mathbf{in}\ u : \rho}\ \otimes \text{elim}$$

The operational semantics of $\otimes$ elimination is given by $\textsc{letp}^a_{\text{Op}}$, which interprets $[\![\Gamma \otimes \Delta \vdash^a \mathbf{let}\ (x, y) = t\ \mathbf{in}\ u : \rho]\!]^a_{\text{Op}}$ by the following rule:

$$\frac{\begin{array}{c}\mathbf{t} \in \mathbf{FQC}^a\ \Gamma\ (\sigma \otimes \tau) \\ \mathbf{u} \in \mathbf{FQC}^a\ (\Delta \otimes \sigma \otimes \tau)\ \rho\end{array}}{\begin{array}{l}\textsc{letp}^a_{\text{Op}}\ \mathbf{t}\ \mathbf{u} \in \mathbf{FQC}^a\ (\Gamma \otimes \Delta)\ \rho \\ \textsc{letp}^a_{\text{Op}}\ \mathbf{t}\ \mathbf{u} = (h_{\mathbf{C}} + h_{\mathbf{t}} + h_{\mathbf{u}}, g_{\mathbf{t}} + g_{\mathbf{u}}, \phi)\end{array}}$$

where $[\![t]\!]^a_{\text{Op}} = \mathbf{t}$, $[\![u]\!]^a_{\text{Op}} = \mathbf{u}$, heap and garbage are interpreted as usual, and ϕ is given by the following circuit:



7.4.3 Interpreting conditionals (**if** and **if**°)

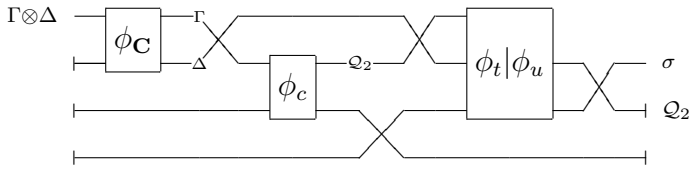
The rules for the classical conditional, **if**, and the quantum conditional, **if**°, are introduced in section 6.5.3. In this section the interpretation of these rules as circuits will be presented, starting with the classical **if** rule:

$$\frac{\begin{array}{c}\Gamma \vdash c : \mathcal{Q}_2 \\ \Delta \vdash t, u : \sigma\end{array}}{\Gamma \otimes \Delta \vdash \mathbf{if}\ c\ \mathbf{then}\ t\ \mathbf{else}\ u : \sigma}\ \mathbf{if}$$

The operational semantics of $[\![\Gamma \otimes \Delta \vdash \textbf{if } c \textbf{ then } t \textbf{ else } u : \sigma]\!]_{\text{Op}}$ is given by IF_{Op}, which is interpreted using the quantum circuit conditional primitive:

$$\mathbf{t|u} = (h_{\mathbf{t|u}}, 0, \phi_t|\phi_u) \in \mathbf{FQC}\ (\mathcal{Q}_\mathbf{2} \otimes \Delta)\ (\mathcal{Q}_\mathbf{2} \otimes \sigma)$$

where the output from the derivation of c is used as the control qubit, to give the following circuit, ψ:



Note how the control qubit c is moved to the top of the conditional circuit, and that after the conditional circuit, the qubit c is placed into the garbage. Placing c into the garbage induces the measurement, and the use of the **if** rule always induces at least one qubit of garbage. Therefore a program using **if** can never be strict. Note that this circuit implicitly makes use of the principle of deferred measurement (see section 3.10.1) to allow garbage, which is measured, to be passed through the computation, without altering the meaning. The interpretation of IF_{Op} is given by:

$$\frac{\begin{array}{c}\mathbf{c} \in \mathbf{FQC}\ \Gamma\ \mathcal{Q}_\mathbf{2} \\ \mathbf{t} \in \mathbf{FQC}\ \Delta\ \sigma \\ \mathbf{u} \in \mathbf{FQC}\ \Delta\ \sigma\end{array}}{\begin{array}{l}\text{IF}_{\text{Op}}\ \mathbf{c}\ \mathbf{t}\ \mathbf{u} \in \mathbf{FQC}\ (\Gamma \otimes \Delta)\ \sigma \\ \text{IF}_{\text{Op}}\ \mathbf{c}\ \mathbf{t}\ \mathbf{u} = (h_\mathbf{C} + h_\mathbf{c} + h_{\mathbf{t|u}}, g_\mathbf{c} + 1, \phi)\end{array}}$$

where $[\![c]\!]_{\text{Op}} = \mathbf{c}$, $[\![t]\!]_{\text{Op}} = \mathbf{t}$, $[\![u]\!]_{\text{Op}} = \mathbf{u}$, and ϕ is the circuit given above. $h_\mathbf{C}$ is the heap required by the context operation $\mathbf{C}$, used to copy any variables used by both sub-circuits, and $h_{\mathbf{t|u}}$ is the maximum of the heap of $\mathbf{t}$ and $\mathbf{u}$.

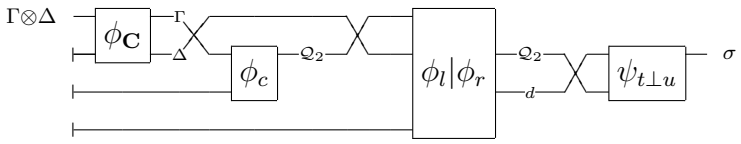
The operational interpretation of the quantum control $\textbf{if}^\circ$ rule is more complicated, due to the requirement that the two branches must be orthogonal. Recall the $\textbf{if}^\circ$ rule is defined as:

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

In order to compile this rule, the orthogonality judgement $t \perp u$ first has to be interpreted, which yields the data $(d, l, r, \psi_{t \perp u})$. How to interpret orthogonality judgements is described in section 7.4.4. The interpretation of $\mathbf{if}^\circ$ follows the same pattern as the interpretation of $\mathbf{if}$, but makes use of $\phi_l, \phi_r \in \mathbf{FQC}^{\simeq}\ d$ and the orthogonality circuit $\psi_{t \perp u}$ in the final stages. The orthogonality judgement circuit $\psi_{t \perp u}$ allows the two branches of the quantum conditional to be distinguished, ensuring that the computation is reversible, thus guaranteeing the orthogonality of the branches. The interpretation $[\![\Gamma \otimes \Delta \vdash^\circ \mathbf{if}^\circ\ c\ \mathbf{then}\ t\ \mathbf{else}\ u : \sigma]\!]^\circ_{\mathrm{Op}}$ is therefore given by the following rule:

$$\frac{\begin{array}{c}\mathbf{c} \in \mathbf{FQC}^\circ\ \Gamma\ \mathcal{Q}_2 \\ \mathbf{t} \in \mathbf{FQC}^\circ\ \Delta\ \sigma \\ \mathbf{u} \in \mathbf{FQC}^\circ\ \Delta\ \sigma\end{array}}{\begin{array}{l}\mathrm{IF}^\circ_{\mathrm{Op}}\ \mathbf{c}\ \mathbf{t}\ \mathbf{u} \in \mathbf{FQC}^a\ (\Gamma \otimes \Delta)\ \sigma \\ \mathrm{IF}^\circ_{\mathrm{Op}}\ \mathbf{c}\ \mathbf{t}\ \mathbf{u} = (h_{\mathbf{C}} + h_{\mathbf{l|r}}, 0, \phi)\end{array}}$$

where $[\![c]\!]^\circ_{\mathrm{Op}} = \mathbf{c}$, $[\![t]\!]^\circ_{\mathrm{Op}} = \mathbf{t}$, $[\![u]\!]_{\mathrm{Op}} = \mathbf{u}$, $h_{\mathbf{l|r}}$ is the heap required by $\phi_l|\phi_r$, and with the $\mathbf{FQC}^{\simeq}$ circuit ϕ given by:



The use of the data from the orthogonality judgement allows this circuit to preserve strictness, as the qubit c used as the conditional control qubit is now utilised by $\psi_{t \perp u}$, and no component produces any garbage output. Hence, the interpretation of $\mathbf{if}^\circ$ is free from measurement and the associated wavefunction collapse.

7.4.4 Interpreting superpositions

In order to compile the superposition rule $\mathbf{sup}^\circ$, introduced in section 6.5.4, it is first reduced by a simple syntactic translation to the problem of generating an arbitrary single qubit state. The $\mathbf{if}^\circ$ rule is used to generate superpositions of terms, as in the

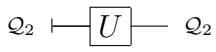
following rule:

$$\frac{\begin{array}{c}\Gamma \vdash^{\circ} t, u : \sigma \\ |\lambda|^2 + |\lambda'|^2 = 1 \qquad t \perp u\end{array}}{\Gamma \vdash^{\circ} (\lambda) \times t + (\lambda') \times u \equiv \mathbf{if}^{\circ}\ (\lambda \times \mathbf{qtrue} + \lambda' \times \mathbf{qfalse})\ \mathbf{then}\ t\ \mathbf{else}\ u}$$

The quantum circuit rotation primitive *rot* is used to generate an arbitrary single qubit superposition by rotating 0, the heap initialisation, to the required value. This is achieved by applying the unitary matrix given by:

$$U = \begin{pmatrix} \lambda & \lambda' \\ \lambda' & -\lambda \end{pmatrix}$$

to the rotation primitive, *rot* U, giving the following circuit:



The rotation given by the matrix U is only one possible rotation, but as this acts on the heap, initialised to 0, this choice is as good as any other. The operational semantics for the evaluation of $[\![\Gamma \vdash^{\circ} \lambda \times \mathbf{qtrue} + \lambda' \times \mathbf{qfalse} : \mathcal{Q}_2]\!]^{\circ}_{\mathrm{Op}}$ is therefore given as:

$$\begin{array}{l}\textsc{sup}^{\circ}_{\mathrm{Op}}\ \lambda\ \lambda' \in \mathbf{FQC}^{\circ}\ 0\ \mathcal{Q}_{\mathbf{2}} \\ \textsc{sup}^{\circ}_{\mathrm{Op}}\ \lambda\ \lambda' = (1, \mathit{rot}\ U)\end{array}$$

where U is the unitary transform defined above.

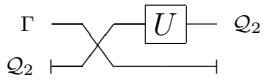
The non-strict constant weakening rule, **sup**:

$$\frac{|\lambda|^2 + |\lambda'|^2 = 1}{\Gamma \vdash (\lambda \times \mathbf{qfalse} + \lambda' \times \mathbf{qtrue})^{\mathrm{dom}\,\Gamma} : \mathcal{Q}_2}\ \mathbf{sup}$$

is interpreted in a similar manner to the non-strict variable rule, and the matrix U is generated in the same way as for the strict case **sup**. Hence the interpretation of $[\![\Gamma \vdash (\lambda \times \mathbf{qfalse} + \lambda' \times \mathbf{qtrue})^{\mathrm{dom}\,\Gamma} : \mathcal{Q}_2]\!]_{\mathrm{Op}}$ is given by the morphism:

$$\begin{array}{l}\textsc{sup}_{\mathrm{Op}}\ \lambda\ \lambda' \in \mathbf{FQC}\ \Gamma\ \mathcal{Q}_{\mathbf{2}} \\ \textsc{sup}_{\mathrm{Op}}\ \lambda\ \lambda' = (1, |\Gamma|, \phi)\end{array}$$

where ϕ is given by the following circuit:



which generates the appropriate superposition using a single heap qubit and places the variables of Γ into the garbage, which is how weakening is modelled.

7.4.5 Interpreting orthogonality ($\perp$)

The interpretation of the orthogonality judgement $t \perp u$ is the most complicated translation from QML to **FQC**. The intention is to derive a circuit which can tell the two orthogonal terms apart. Given $\Gamma \vdash^{\circ} t, u : \rho$, the derivation $\frac{d}{t \perp u}$ is given by $[\![d]\!]^{\perp}_{\mathrm{Op}} = (c, l, r, \psi)$, where:

$c \in \mathbb{N}$

$l \in \mathbf{FQC}^{\circ}\ \Gamma\ c$

$r \in \mathbf{FQC}^{\circ}\ \Gamma'\ c$

$\psi \in \mathbf{FQC}^{\simeq}\ \sigma\ (\mathcal{Q}\mathbf{2} \otimes c)$

by induction over the derivation. Each case is defined below:

Orthogonality of pure qubits:

$$\frac{}{\textbf{qfalse} \perp \textbf{qtrue}} \qquad \frac{}{\textbf{qtrue} \perp \textbf{qfalse}}$$

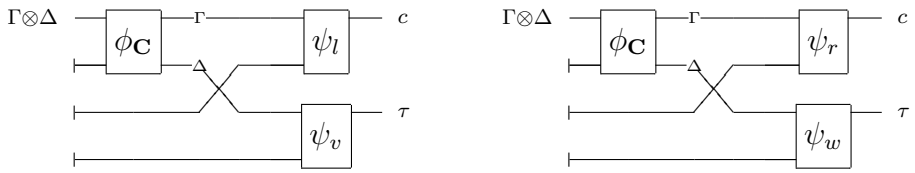
The axioms above state that **qtrue** $\perp$ **qfalse**, and vice-versa. To interpret both of these cases, $c = 0$, and l, r both are the empty **FQC** morphism: **FQC** 0 0. The definition of the circuit ψ, is the only difference between these cases:

with **qfalse** $\perp$ **qtrue** on the left, and **qtrue** $\perp$ **qfalse** on the right.

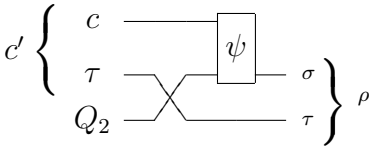
Orthogonality of pairs:

$$\frac{t \perp u}{(t,v) \perp (u,w)} \perp \mathbf{pair}_0 \qquad \frac{t \perp u}{(v,t) \perp (w,u)} \perp \mathbf{pair}_1$$

To interpret the orthogonality of pairs, let $\Gamma \vdash^{\circ} (t,v),(u,w) : \sigma \otimes \tau$ and let (c,l,r,ψ) be the interpretation of $t \perp u$. From this, the interpretation of $(t,v) \perp (u,w)$ as (c',l',r',ψ') can be constructed. In this case, the value $c' = c \otimes |\tau|$, and l', r' are constructed by semantically pairing l, r with v, w:



The definition of ψ' in both cases is given by the following diagram:



Orthogonality of superpositions:

$$\frac{t \perp u \qquad \lambda_0^* \kappa_0 = -\lambda_1^* \kappa_1}{\lambda_0 \times t + \lambda_1 \times u \perp \kappa_0 \times t + \kappa_1 \times u}$$

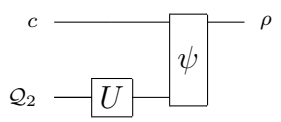
Given the interpretation of $t \perp u$ as (c,l,r,ψ), the interpretation of this rule is (c,l,r,ψ'), where the definition of ψ' relies on $\phi \in \mathbf{FQC}^{\simeq}\ \mathcal{Q}_{\mathbf{2}}$, which is defined as:

$$\phi = rot\ U$$

where U is given by the unitary matrix:

$$U = \begin{pmatrix} \lambda_0 & \lambda_1 \\ \kappa_0 & \kappa_1 \end{pmatrix}$$

Using this rotation, the circuit ψ' is defined as:



where ρ is the output type of the derivation $t \perp u$.

Recall from the discussion of the orthogonality judgement in section 6.5.5 that the definition of orthogonality given here is not complete, and that other rules may be added. This is indeed an area of ongoing research (see section 8.4).

7.5 A denotational semantics for QML

The purpose of a denotational semantics is to provide a formal interpretation of a program, by assigning a mathematical object (called a denotation) to each term in a language. The key difference between denotational semantics and operational semantics is that while the latter is primarily concerned with how a computation is realised, the denotational semantics is concerned with the meaning of the computation. The denotational semantics of QML gives a mathematical framework to QML that supports reasoning and allows optimisation principles to be applied, and therefore facilitates the expression and analysis of proofs and theorems.

The operational semantics for QML is presented as a translation from QML terms into **FQC** morphisms, in section 7.4. A denotational semantics for QML is written as a translation of QML terms into superoperators in the category **Q**, which is introduced in section 5.5.1. The translation from QML terms to the category **Q** uses the operational semantics to first translate a QML term into an **FQC** morphism. The **FQC** morphism produced by the semantic function $[\![\cdot]\!]_{\mathrm{Op}}$ is then translated to a superoperator using the functor defined in section 5.5. This mapping, denoted $[\![\cdot]\!]$, maps $\mathbf{FQC}^\circ$ morphisms into isometries in $\mathbf{Q}^\circ$, and **FQC** morphisms into superoperators in **Q**. Therefore, strict QML terms, which are mapped to $\mathbf{FQC}^\circ$ morphisms by the operational semantics, are mapped to isometries in $\mathbf{Q}^\circ$ by the denotational semantics, and non-strict QML terms are mapped to superoperators in **Q**. Non-strict terms produce garbage, and in the denotational semantics garbage is managed using the partial trace superoperator, hence the interpretation of non-strict terms is given in the category **Q** of superoperators, which allows garbage to be modelled. Strict terms produce no garbage, and can be interpreted in the category $\mathbf{Q}^\circ$ of isometries. An isometry, f, can be lifted to a superoperator, $\widehat{f}$, as defined in section 5.5.1.

Following from this interpretation, strict QML derivations $\frac{d}{\Gamma \vdash^{\circ} t : \sigma}$ are therefore interpreted by the semantic function:

$$[\![d]\!]^{\circ}_{\mathrm{D}} \in \mathbf{QML}^{\circ}\ \Gamma\ \sigma \to \mathbf{Q}^{\circ}\ \Gamma\ \sigma$$
$$[\![d]\!]^{\circ}_{\mathrm{D}} = [\![[\![d]\!]^{\circ}_{\mathrm{Op}}]\!]$$

which gives an isometry, while non-strict QML derivations $\frac{d}{\Gamma \vdash t : \sigma}$ are interpreted by:

$$[\![d]\!]_{\mathrm{D}} \in \mathbf{QML}\ \Gamma\ \sigma \to \mathbf{Q}\ \Gamma\ \sigma$$
$$[\![d]\!]_{\mathrm{D}} = [\![[\![d]\!]_{\mathrm{Op}}]\!]$$

which results in a superoperator. The action of the denotational semantics is summarised in figure 7.1, which shows the relationships between **QML**, **QML**$^{\circ}$, **FQC**, **FQC**$^{\circ}$, **Q**$^{\circ}$ and **Q**. The embeddings shown in figure 7.1 between **QML** and **QML**$^{\circ}$

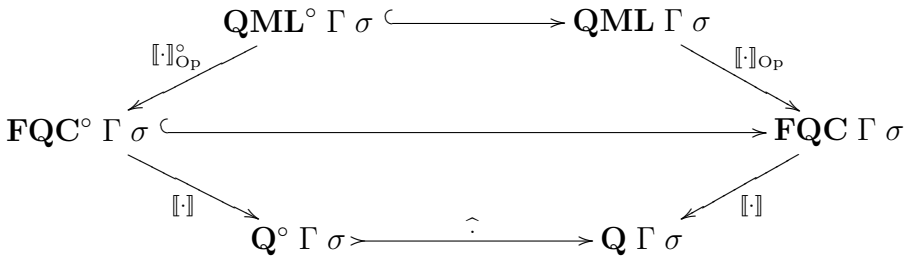


FIGURE 7.1: This diagram shows how the denotational semantics of QML is defined: by first applying the operational semantics and then interpreting the resulting morphism as either an isometry or a superoperator.

and between **FQC** and **FQC**$^{\circ}$ are trivial, as **QML**$^{\circ} \subset$ **QML** (see section 7.1) and **FQC**$^{\circ} \subset$ **FQC** (see section 5.2.1). Full details of the functor $[\![\cdot]\!] \in \mathbf{FQC} \to \mathbf{Q}$ are given in sections 5.3 through to 5.5.4.

7.5.1 Extensional equality

The definition of the denotational semantics presented in this section also leads to a definition of extensional equality for QML programs. Two QML programs are defined as extensionally equivalent if the interpretation of the programs by the denotational semantics gives rise to equal superoperators. This is the same as the definition of extensional equality for **FQC** morphisms, as it this definition which is being exploited, which is discussed in section 5.5.4.

An alternative definition of the denotational semantics, outlined in section 8.1, gives a direct interpretation of QML terms in the category **Q**, without factoring through the operational semantics. A direct denotational semantics can be defined in the same way as the operational semantics, by defining for each term in QML a mapping directly to **Q**. The direct denotational semantics can then be used to show compositionality, which is also discussed in section 8.1.

7.5.2 Implementing the denotational semantics in Haskell

In section 5.8 the Haskell function *fqc2super* $\in$ **FQC** $\rightarrow$ **Super** is defined, which translates **FQC** morphisms into superoperators. The composition of this function with the Haskell implementation of the operational semantics, which is given in section 7.6, gives an implementation of the denotational semantics presented in section 7.5. If the QML term is strict, then the function *fqco2isom* $\in$ $\mathbf{FQC}^{\circ} \rightarrow$ **Isom** should be used, as the operational semantics will produce a strict morphism. This could be lifted to a superoperator by use of the *isom2super* function, also defined in section 5.8.

7.6 Compiling QML in Haskell

This section will explore how QML, and a compiler from QML into **FQC** (or more accurately, into typed quantum circuits), can be implemented in Haskell. The first step will be to define the terms of the language QML in Haskell, and to introduce the notion of a typed circuit, which extends the implementation of **FQC** in Haskell with contexts and types. A compiler function from the Haskell QML syntax to typed quantum circuits is then defined, which follows the derivation of circuits given in section 7.4. Code for the QML compiler outlined here will be made available from the project website [30].

7.6.1 Typed quantum circuits

Before typed QML terms and typed quantum circuits in Haskell can be introduced, a Haskell implementation of types and contexts is required. These follow the definition of types and contexts given in section 6.5, and are implemented as the following Haskell datatypes:

$$\begin{aligned}&\textbf{data } Ty &&= \mathcal{Q}_\mathbf{1} \mid \mathcal{Q}_\mathbf{2} \mid Ty \otimes Ty\\ &\textbf{type } TCon &&= [Ty]_R\\ &\textbf{type } Con &&= [(String, Ty)]_R\end{aligned}$$

where $[\cdot]_R$ denotes a reverse-list. $TCon$ is the type of type-contexts, while Con gives the type of named-type contexts. $Ty, TCon$ and Con have a function $size$, which returns in each case the size of the type, and the sum of the type sizes, as defined in section 7.2.

A typed quantum circuit is essentially a Haskell **FQC** morphism, as defined in section 5.7. In **FQC** the input a and output b are defined as natural numbers. However, in a typed quantum circuit the full context input to a program is stored, which has a size equal to a, and the output size is replaced by its type, which has a size equal to b. Typed quantum circuits are implemented simply as:

$$\textbf{data } TCirc = TCirc\{\, inT \in TCon, outT \in Ty, hpS, gbS \in Int, circ \in Circ\,\}$$

where $Circ$ denotes the quantum circuit datatype defined in section 3.5. Storing the context and type information used by a circuit simplifies the design of the compiler.

A typed circuit can be translated into an **FQC** morphism by the following function:

$$\begin{aligned}&tyc2fqc &&\in TCirc \to \mathbf{FQC}\\ &tyc2fqc\ (TCirc\ a\ b\ h\ g\ \phi) &&= \mathbf{FQC}\ |a|\ |b|\ h\ g\ \phi\end{aligned}$$

where $|\cdot|$ is the context/type size function. Composing the function $tyc2fqc$ with the $validFQC$ function, defined in section 5.7, gives a new function which tests the validity of a typed circuit, given by the Haskell function:

$$\begin{aligned}&validTCirc &&\in TCirc \to Error\ TCirc\\ &validTCirc\ tc &&= \textbf{do}\ (validFQC \circ tyc2fqc)\ tc\\ &&&\quad\ return\ tc\end{aligned}$$

7.6.2 QML syntax in Haskell

QML programs are written using the syntax given in section 6.5. An example is the familiar Hadamard function that has been used several times, which is given as:

$$
\begin{aligned}
&\mathit{had} \quad : \mathcal{Q}_2 \multimap \mathcal{Q}_2 \\
&\mathit{had}\ x = \mathbf{if}^\circ\ x\ \mathbf{then}\ (-1) \times \mathbf{qtrue} + \mathbf{qfalse} \\
&\qquad\qquad\quad \mathbf{else}\ \ \mathbf{qtrue} + \mathbf{qfalse}
\end{aligned}
$$

A QML parser, which is not defined here, would take this code and translate it into a Haskell representation of the QML syntax. The Haskell version of the syntax is based on the (simplified) QML typing rules used to define the operational semantics of QML, given in section 7.4. QML terms are given in Haskell by the datatype shown in figure 7.2, which includes comments explaining the mapping to the QML operational semantics.

$$
\begin{aligned}
&\mathbf{data}\ \mathit{Tm} = \mathit{Atom}\ \mathit{At}\ [\mathit{Name}]_R && \text{-- Atomic weakening} \\
&\quad \mid \mathit{Pair}\ \ \mathit{Tm}\ \mathit{Tm} && \text{-- } \textsc{pair}^a_{\mathrm{Op}} \\
&\quad \mid \mathit{Let}\ \ \mathit{Name}\ \mathit{Tm}\ \mathit{Tm} && \text{-- } \textsc{let}^a_{\mathrm{Op}} \\
&\quad \mid \mathit{LetP}\ \ \mathit{Name}\ \mathit{Name}\ \mathit{Tm}\ \mathit{Tm} && \text{-- } \textsc{letp}^a_{\mathrm{Op}} \\
&\quad \mid \mathit{If}\ \ \mathit{Tm}\ \mathit{Tm}\ \mathit{Tm} && \text{-- } \textsc{if}_{\mathrm{Op}} \\
&\quad \mid \mathit{If}^\circ\ \ \mathit{Tm}\ \mathit{Tm}\ \mathit{Tm}\ \mathit{Orth} && \text{-- } \textsc{if}^\circ_{\mathrm{Op}} \\
&\mathbf{data}\ \mathit{At}\ \ = \mathit{Var}\ \mathit{Name} && \text{-- } \textsc{var}^\circ_{\mathrm{Op}} \text{ and } \textsc{var}_{\mathrm{Op}} \\
&\quad \mid \mathit{Sup}\ \mathbb{C}\ \mathbb{C} && \text{-- } \textsc{sup}^\circ_{\mathrm{Op}}\ \lambda\ \lambda' \text{ and } \textsc{sup}_{\mathrm{Op}}\ \lambda\ \lambda'
\end{aligned}
$$

FIGURE 7.2: The syntax of QML terms, rendered as a Haskell datatype. The comments give the mapping from each Haskell type to the operational semantics.

The atomic terms *Var* and *Sup* in figure 7.2 represent QML terms that can be weakened. If the list of variable names passed to the *Atom* constructor is empty, then no weakening takes place and the strict form of the rule is used. Otherwise the appropriate weakening rule is used, with the variables in the list used to represent the set of variables to be weakened. The If° term includes an *Orth* type, which is the type of the orthogonality judgement discussed in section 7.6.9. The *Var* term includes a *Name* argument, which refers to a type in the context of the program.

Finally, the *Sup* term represents a single qubit superposition, and takes two complex numbers as arguments. The translation from a superposition into *Sup* is defined as:

$$\lambda \times \mathbf{qfalse} + \lambda' \times \mathbf{qtrue} \equiv \mathit{Sup}\ \lambda\ \lambda'$$

Returning to the Hadamard operation example, passing the function *had* to a parser returns the following Haskell representation of QML terms:

$$\mathit{If}^{\circ}\ (\mathit{Atom}\ (\mathit{Var}\ \texttt{"x"})\ [\cdot]_R)\ (\mathit{Sup}\ (\tfrac{1}{\sqrt{2}})\ (-\tfrac{1}{\sqrt{2}}))$$
$$(\mathit{Sup}\ (\tfrac{1}{\sqrt{2}})\ (\tfrac{1}{\sqrt{2}}))\ (\mathit{OSup}\ \mathit{OBit}_0)$$

where $(\mathit{OSup}\ \mathit{OBit}_0)$ is an orthogonality judgement, which is required whenever If° is used; see section 7.6.9.

7.6.3 Defining the compiler input and output

The output of the Haskell QML compiler is an **FQC** object represented as a typed quantum circuit, along with a list of all the variables used by the circuit. This is represented by the output type *Comp*, defined as:

$$\mathbf{data}\ \mathit{Comp} = \mathit{Comp}\{\, \mathit{uVar} \in [\mathit{Name}]_R, \mathit{fqc} \in \mathit{TCirc}\}$$

The list of variables used, *uVar*, is kept to ensure type correctness; a computation is not correct unless every variable passed to it in the program context has been used. If a variable is not used it could interfere with the computation, possibly inducing a measurement. Hence the type system must ensure that all variables that are defined are actually used. Since a program may contain multiple function definitions, the final output type of the compiler is given by the type *Code*:

$$\mathbf{type}\ \mathit{Code} = \mathit{Env}\ \mathit{Comp}$$

where *Env* is the type of named environments, given by:

$$\mathbf{data}\ \mathit{Env}\ a = \mathit{Env}\{\, \mathit{unEnv} \in [(\mathit{Name}, a)]_R\}$$

$$\mathit{elookup} \in \mathit{Eq}\ a \Rightarrow \mathit{Name} \to (\mathit{Env}\ a) \to \mathit{Error}\ a$$

$$\mathit{elookup}\ n\ (\mathit{Env}\ \vec{x}) = \mathit{slookup}\ n\ \vec{x}$$

where *elookup* is the *Env* lookup function, which uses *slookup*, which is the lookup function on $[\cdot]_R$.

The input to the compiler is a QML program, represented by the type *Prog*, which is an environment of function definitions, *FDef*. In turn, a function definition is a

function signature, *FSig*, which contains the required context with the return type of the function, and the QML term that defines the action of the function. This is encapsulated by the following Haskell types:

$$\textbf{type}\ \mathit{Prog} = \mathit{Env}\quad \mathit{FDef}$$

$$\textbf{data}\ \mathit{FDef} = \mathit{FDef}\ \mathit{FSig}\ \mathit{Tm}$$

$$\textbf{data}\ \mathit{FSig} = \mathit{FSig}\ \ \mathit{Con}\ \ \mathit{Ty}$$

As an example of a QML program rendered in Haskell, consider the QML quantum negation function *qnot*:

$$\mathit{qnot} : \mathcal{Q}_{\mathbf{2}} \multimap \mathcal{Q}_{\mathbf{2}}$$

$$\mathit{qnot}\ x = \textbf{if}^{\circ}\ x\ \textbf{then qfalse}$$

$$\textbf{else}\ \ \textbf{qtrue}$$

This would be realised by a suitable parser in Haskell as:

$$\mathit{ex1} \in \mathit{Prog}$$

$$\begin{aligned}\mathit{ex1} = \mathit{Env}\ [\ (\texttt{"qnot"}, \mathit{FDef}\ (\mathit{FSig}\ [(\texttt{"x"}, \mathcal{Q}_{\mathbf{2}})]\ \mathcal{Q}_{\mathbf{2}})\\ (\mathit{If}^{\circ}\ (\mathit{Atom}\ (\mathit{Var}\ \texttt{"x"})\ [\cdot]_R)\\ (\mathit{Sup}\ 1\ 0)\\ (\mathit{Sup}\ 0\ 1)\ \mathit{OBit}_0))\ ]_R\end{aligned}$$

Applying the QML to FQC compiler to this program would produce the following output:

$$\begin{aligned}\mathit{Code}\ [\ \mathit{Comp}\{\, \mathit{uVar} = [\ \texttt{"x"}\ ]_R,\\ \mathit{fqc}\quad = \mathit{TCirc}\ [\ \mathcal{Q}_{\mathbf{2}}\ ]_R\ \mathcal{Q}_{\mathbf{2}}\ 0\ 0\ \mathit{Not}\,\}]_R\end{aligned}$$

These definitions provide enough structure to build the definition of the Haskell QML compiler.

7.6.4 Implementing the operational semantics in Haskell

A compiler for QML programs expressed in the Haskell data types defined in section 7.6.3 is defined by the function *compileProg*. This function makes use of an auxiliary function *compileTm* to compile each QML term, and a function *checkCompProg* which ensures the type correctness of the compiled program. The function *compileProg* is a recursive function which takes as input a QML program implemented in Haskell, and

is defined as:

$$
\begin{aligned}
&\mathit{compileProg} \ \in \mathit{Prog} \to \mathit{Error}\ (\mathit{Env}\ \mathit{Comp}) \\
&\mathit{compileProg}\ (\mathit{Env}\ [\cdot]_R) = \mathit{OK}\ (\mathit{Env}\ [\cdot]_R) \\
&\mathit{compileProg}\ (\mathit{Env}\ (p : (f, (\mathit{FDef}\ (\mathit{FSig}\ \Gamma\ \sigma)\ \mathit{cmp})))) = \\
&\quad \mathbf{do}\ c \quad \leftarrow \mathit{compileProg}\ (\mathit{Env}\ p) \\
&\qquad \mathit{cmp}' \leftarrow \mathit{compileTm}\ c\ \Gamma\ \mathit{cmp}\ (\mathit{Just}\ \sigma) \\
&\qquad \mathit{checkCompProg}\ \mathit{cmp}'\ \Gamma\ \sigma \\
&\qquad \mathit{return}\ (\mathit{Env}\ ((\mathit{unEnv}\ c) : (f, \mathit{cmp}')))
\end{aligned}
$$

This function is mainly concerned with error checking and preparing the input for the *compileTm* function, which is where the actual work of the compilation takes place. The function *checkCompProg* checks that the output from the *compileTm* function uses all the variables in the context Γ, and that the output type of the typed circuit produced by the compilation matches the output type expected, σ.

The function that performs the majority of the compilation, *compileTm*, has the type:

$$\mathit{compileTm} \in \mathit{Code} \to \mathit{Con} \to \mathit{Tm} \to \mathit{Maybe}\ \mathit{Ty} \to \mathit{Error}\ \mathit{Comp}$$

The type *Code* contains the functions that have already been compiled by previous calls of *compileProg*, and are available for use by the compiler. *compileTm* also takes as input the current context, the term to be compiled, and the expected type (if known, hence the use of the *Maybe* type) of the term, and it returns either an error or a *Comp* object. The function is defined using pattern matching over the type of terms, *Tm*, so there is a separate function for each of the term forms given by the grammar for QML. For each term in the grammar of QML, the *compileTm* function constructs a typed circuit following the interpretation presented in section 7.4. The definitions of a sample of these interpretations in Haskell shall now be explored.

7.6.5 Compiling the structural rules in Haskell

The interpretations of the variable rules in Haskell are straightforward translations of the circuits shown in section 7.4.1. The operational semantics of the strict variable rule, given by $\textsc{var}^{\circ}_{\mathrm{Op}}$, is interpreted by the *compileTm* function as:

$$
\begin{aligned}
&\mathit{compileTm}\ \_\ \Gamma\ (\mathit{Atom}\ (\mathit{Var}\ x)\ [\cdot]_R)\ \mathit{mTy} = \\
&\quad \mathbf{do}\ \sigma \leftarrow \mathit{slookup}\ x\ \Gamma \\
&\qquad \mathit{treturn}\ \mathit{mTy}\ (\mathit{Comp}\ [x]_R\ (\mathit{TCirc}\ [\sigma]_R\ \sigma\ 0\ 0\ (\mathit{Wire}\ [0\,..\,|\sigma|-1])))
\end{aligned}
$$

To compile a strict variable, first the type of the variable x is looked up in the context Γ, and is called σ (to match the circuit of $\textsc{var}_{\text{Op}}$). The $[\cdot]_R$ lookup function, *slookup*, returns an Error type if the variable does not exist. The interpretation of the strict variable rule is the identity morphism on the type σ, and this is exactly the typed circuit produced: it has no heap or garbage, takes a context containing only σ as input, and outputs the type σ, with the reversible quantum circuit defined as id_σ. This exactly matches $\textsc{var}^{\circ}_{\text{Op}}$ given in section 7.4.1, which is:

$$\sigma \text{ ——— } \sigma$$

with no heap or garbage.

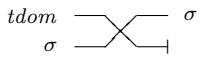
The only variable used is $x \in \sigma$, and this is the only content of the *uVar* parameter of the *Comp* type. The monadic return function *treturn* is a small extension of the usual *return* function, which confirms the type returned by the compilation matches the expected type *mTy* passed to the *compileTm* function:

$$
\begin{aligned}
&\mathit{treturn} && \in \mathit{Maybe}\ \mathit{Ty} \to \mathit{Comp} \to \mathit{Error}\ \mathit{Comp} \\
&\mathit{treturn}\ \mathit{mT}\ \mathit{cmp} \mid \mathit{mT} \equiv \mathit{Nothing} && = \mathit{return}\ \mathit{cmp} \\
&\qquad\qquad \mid |\mathit{mT}| \equiv |(\mathit{outT} \circ \mathit{fqc})\ \mathit{cmp}| && = \mathit{return}\ \mathit{cmp} \\
&\qquad\qquad \mid \mathit{otherwise} && = \mathit{Error}\ (\texttt{"Type error: "} +\!\!+\ \mathit{cmp})
\end{aligned}
$$

The non-strict variable rule compilation, $\textsc{var}_{\text{Op}}$, is implemented in much the same way as the implementation of $\textsc{var}^{\circ}_{\text{Op}}$ above, where *dom* is the list of variables to be weakened:

$$
\begin{aligned}
&\mathit{compileTm}\ \_\ \Gamma\ (\mathit{Atom}\ (\mathit{Var}\ x)\ \mathit{dom})\ \mathit{mTy} = \\
&\quad \mathbf{do}\ \sigma \quad \leftarrow \mathit{slookup}\ x\ \Gamma \\
&\qquad \mathit{tdom} \leftarrow \mathit{populate}\ \Gamma\ \mathit{dom} \\
&\qquad \mathbf{let}\ \mathit{ld} \quad = \mathit{length}\ \mathit{dom} \\
&\qquad\quad \mathit{iTy} = \mathit{tdom} : \sigma \\
&\qquad\quad \vec{p} \quad = \mathit{permFromTy}\ \mathit{iTy}\ ([\mathit{ld}] +\!\!+ [0\,..\,\mathit{ld}-1]) \\
&\qquad \mathit{treturn}\ \mathit{mTy}\ (\mathit{Comp}\ (\mathbf{dom} : x)\ (\mathit{TCirc}\ \mathit{iTy}\ \sigma\ 0\ |\mathit{tdom}|\ (\mathit{Wires}\ \vec{p})))
\end{aligned}
$$

The value of x is again looked up in the context Γ, and called σ. The function *populate* then creates the context of the variables to be weakened, given by the list *dom*, by looking up the types in Γ. The input type is calculated to be the variables to be weakened, *tdom*, plus the variable σ, while the output type is only σ, with the context *tdom* being placed in the garbage. This is achieved with the permutation *Wires* $\vec{p}$, where $\vec{p}$ is defined by the function *permFromTy*, which calculates the bijection required to move the wires representing σ from the bottom of the circuit to the top. Diagrammatically, this is shown as:



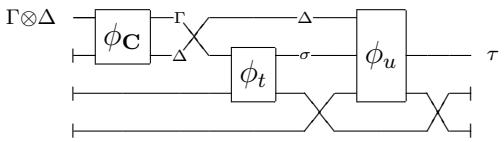
which follows the circuit $\textsc{var}_{\mathrm{Op}}$ presented in section 7.4.1, where $tdom = \Gamma$.

The final structural rule to be compiled in Haskell is the let-rule, given by $\textsc{let}^{a}_{\mathrm{Op}}$. The compilation of a **let** term proceeds by first compiling the subterms t and u. The contexts required by these terms is then calculated, and from this the circuit for context operation, **C**, is calculated. The circuits are then all wired together as prescribed by the circuit definition, and the result is returned along with the list of used variables, which are all the variables of t and u, minus the variable x, which is only locally declared. Each line of the Haskell code for this compilation is explained by the following comment:

```
compileTm c Γ (Let x t u) mTy =
  do ct ← compileTm c Γ t Nothing      -- Compile term t
     let Γ'    = Γ : (x, (outT ∘ fqc) ct)   -- Add x ∈ σ to context
     cu ← compileTm c Γ' u mTy         -- Compile term u, using Γ'
     (tC, uC) ← pSubCon Γ (uVar ct) (sfilter (≢ x) (uVar cu))
          -- Calculate contexts for subterms, removing x
     let fdC = deltaCon tC uC          -- Calculate deltaCon (C)
         (ft, fu)   = (fqc ct, fqc cu)     -- Get t and u circuits
         (h_t, h_u) = (hpS ft, hpS fu)     -- Get heap sizes
         (g_t, g_u) = (gbS ft, gbS fu)     -- Get garbage sizes
         (sg, sd)   = (size tC, size uC)   -- Get context sizes
         (bt, bu)   = ((size ∘ outT) ft, (size ∘ outT) fu)
```

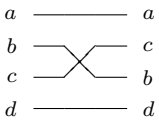
$$
\begin{aligned}
&\text{-- Get output sizes}\\
(hS, gS) &= (h_{\mathbf{t}} + h_{\mathbf{u}} + hpS\ fdC, gbS\ ft + gbS\ fu)\\
&\text{-- Calculate full heap and garbage sizes}\\
\phi \quad &= Seq\ (Seq\ (Seq\ (Seq\ (Seq\ \ (Par\ (circ\ fdC)\\
&\qquad\qquad (Wire\ [0\,..\,h_{\mathbf{t}} + h_{\mathbf{u}} - 1]))\\
&\qquad\qquad (permN\ 0\ sg\ sd\ (h_{\mathbf{t}} + h_{\mathbf{u}})))\\
&\qquad\qquad (Par\ (Par\ (idC\ sd)\ (circ\ ft))\\
&\qquad\qquad (idC\ h_{\mathbf{u}})))\\
&\qquad\qquad (permN\ (sd + bt)\ g_{\mathbf{t}}\ h_{\mathbf{u}}\ 0))\\
&\qquad\qquad (permN\ bu\ g_{\mathbf{u}}\ g_{\mathbf{t}}\ 0))\\
&\qquad\qquad (Par\ (circ\ fu)\ (idC\ g_{\mathbf{t}}))\\
&\qquad\qquad \text{-- Generate the } \textsc{let}^{a}_{\mathrm{Op}} \text{ circuit}\\
&\qquad\qquad treturn\ mTy\ (Comp\ (uniqueVars\ (uVar\ ct)\ (sfilter\ (\not\equiv x)\ (uVar\ cu)))\\
&\qquad\qquad\qquad (TCirc\ (inT\ fdC)\ (outT\ fu)\ hS\ gS\ \phi))
\end{aligned}
$$

The majority of the code used to compile a **let** rule is concerned with calculating the sizes of various types and contexts in order to correctly wire the circuit ϕ. The actual quantum circuit ϕ corresponds directly to the definition given in section 7.4.1 of the $\textsc{let}^{a}_{\mathrm{Op}}$, which is given as the circuit below:



The compile function for the let rule makes use of several auxiliary functions. The function *pSubCon* takes the lists of variables used by the two subcircuits and the context Γ, and from this information generates the contexts required for the subcircuits, by looking up the types in Γ. The function *deltaCon* generates the circuit for the **C** operation on the context, and is explored in section 7.6.7. *uniqueVars* is a simple function that lists which variables have been used at least once in the entire computation, excepting the variable x which is filtered out, as x is only in scope inside this let-block. Lastly, *permN a b c d* generates a permutation which swaps two groups of wires, and is best explained with a diagram, where $a \in \mathbb{N}$ represents the first a wires

as a group, $b \in \mathbb{N}$ is the next b wires, $c \in \mathbb{N}$ is the next c wires, and $d \in \mathbb{N}$ is the final group of d wires. The action is given as:



For example, $\mathit{permN}\ 1\ 2\ 3\ 1 = \mathit{Wire}\ [0,3,4,5,1,2,6]$; the action essentially swaps the b and c wire groups. It is computing these permutations which gives rise to most of the calculations required to implement the QML operational semantics.

7.6.6 Compiling the superposition rules in Haskell

The rules **sup** and **sup**$^\circ$ are essential to QML, as without them there is no data, and no superpositions can be expressed. The interpretations of the operational semantics of these rules for superpositions, given by $\textsc{sup}^\circ_{\mathrm{Op}}$ and $\textsc{sup}_{\mathrm{Op}}$ in section 7.4.4, are very similar to the interpretation of variables given in section 7.6.5. The strict superposition rule, **sup**, is interpreted by:

$$[\![\Gamma \vdash^\circ \lambda \times \mathbf{qtrue} + \lambda' \times \mathbf{qfalse} : \mathcal{Q}_2]\!]^\circ_{\mathrm{Op}} = \textsc{sup}^\circ_{\mathrm{Op}}\ \lambda\ \lambda' \in \mathbf{FQC}^\circ\ 0\ \mathcal{Q}_2$$

as the circuit:

$$\text{where}\quad U = \begin{pmatrix} \lambda & \lambda' \\ \lambda' & -\lambda \end{pmatrix}$$

In Haskell, this is translated into the following *compileTm* definition:

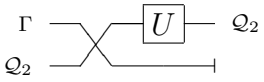
```
compileTm _ _ (Atom (Sup λ λ') [·]_R) mTy =
  treturn mTy (Comp [·]_R (TCirc [·]_R Q2 1 0 (Rot (λ, λ') (λ, (−λ')))))
```

which simply performs the rotation U, taking the input qubit from the heap.

The non-strict variable rule is given the interpretation:

$$[\![\Gamma \vdash (\lambda \times \mathbf{qfalse} + \lambda' \times \mathbf{qtrue})^{\mathrm{dom}\,\Gamma} : \mathcal{Q}_2]\!]_{\mathrm{Op}} = \textsc{sup}_{\mathrm{Op}}\ \lambda\ \lambda' \in \mathbf{FQC}\ \Gamma\ \mathcal{Q}_2$$

where $\textsc{sup}_{\mathrm{Op}}$ is given by the following circuit:



The SUP_{Op} morphism has an interpretation in Haskell similar to that of the VAR_{Op} morphism, and is given as:

$$
\begin{aligned}
&\mathit{compileTm}\ \_\ \Gamma\ (\mathit{Atom}\ (\mathit{Sup}\ \lambda\ \lambda')\ \mathit{dom})\ \mathit{mTy} = \\
&\quad \mathbf{do}\ \mathit{tdom} \leftarrow \mathit{populate}\ \Gamma\ \mathit{dom} \\
&\qquad \mathbf{let}\ \mathit{ld} = \mathit{length}\ \mathit{dom} \\
&\qquad\quad \vec{p}\ = \mathit{permFromTy}\ (\mathit{tdom} : \mathcal{Q}_{\mathbf{2}})\ ([\mathit{ld}] +\!\!+ [0\,..\,\mathit{ld}-1]) \\
&\qquad\quad \phi\ = \mathit{Seq}\ (\mathit{Par}\ (\mathit{idC}\ |\mathit{tdom}|) \\
&\qquad\qquad\qquad\qquad (\mathit{Rot}\ (\lambda, \lambda')\ (\lambda', -\lambda))) \\
&\qquad\qquad\qquad (\mathit{Wire}\ \vec{p}) \\
&\qquad \mathit{treturn}\ \mathit{mTy}\ (\mathit{Comp}\ \mathit{dom}\ (\mathit{TCirc}\ \mathit{tdom}\ \mathcal{Q}_{\mathbf{2}}\ 1\ |\mathit{tdom}|\ \phi))
\end{aligned}
$$

This code follows the pattern of the VAR_{Op} implementation to weaken the variables of *dom* (by moving them to the garbage), and also performs the rotation given by U, to implement the SUP_{Op} circuit given above. Note that $\mathit{idC}\ x$, where $x \in \mathbb{N}$, is shorthand for the identity circuit on x qubits: $\mathit{Wires}\ [0\,..\,(x-1)]$.

All of the QML terms compiled so far have not made use of the $\otimes$ operation on contexts. In order to implement any more of the operational semantics of QML in Haskell, the action of this function must be defined as a circuit, as outlined in section 7.6.7.

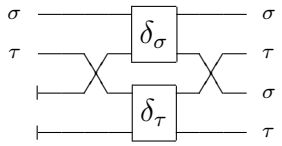
7.6.7 Haskell operations on contexts

In this section the operations on contexts described in section 7.3 will be implemented in Haskell. The first function considered will be the *deltaTy* function, which is implemented in Haskell as a circuit which takes as input a QML type, and outputs a circuit which creates a copy of that type by repeated use of the controlled-not operation (*cnotC*). The function *deltaTy* is presented by pattern matching on the QML type parameter, and is defined as:

$$
\begin{aligned}
&\mathit{deltaTy} && \in \mathit{Ty} \to \mathit{TCirc} \\
&\mathit{deltaTy}\ \mathcal{Q}_{\mathbf{1}} && = \mathit{TCirc}\ [\mathcal{Q}_{\mathbf{1}}]_R\ (\mathcal{Q}_{\mathbf{1}} \otimes \mathcal{Q}_{\mathbf{1}})\ 0\ \ 0\ (\mathit{Wires}\ [\,]) \\
&\mathit{deltaTy}\ \mathcal{Q}_{\mathbf{2}} && = \mathit{TCirc}\ [\mathcal{Q}_{\mathbf{2}}]_R\ (\mathcal{Q}_{\mathbf{2}} \otimes \mathcal{Q}_{\mathbf{2}})\ 1\ \ 0\ \mathit{cnotC} \\
&\mathit{deltaTy}\ i@(\sigma \otimes \tau) && = \mathit{TCirc}\ [i]_R\ \ (i \otimes i)\ \ \ |i|\ 0\ \phi
\end{aligned}
$$

$$\begin{aligned}
&\textbf{where } \phi = (Seq\ (Seq\ (Wires\ \vec{p})\\
&\qquad\qquad (Par\ ((circ \circ deltaTy)\ \sigma)\\
&\qquad\qquad\quad ((circ \circ deltaTy)\ \tau)))\\
&\qquad (Wires\ \vec{p}))\\
&\vec{p} = permFromTy\ [(\sigma : \tau : \sigma : \tau)]_R\ [0, 2, 1, 3]
\end{aligned}$$

The first case is trivial, as $\mathcal{Q}_\mathbf{1}$ and $\mathcal{Q}_\mathbf{1} \otimes \mathcal{Q}_\mathbf{1}$ carry no information. The second case is simply the controlled not operation on a single qubit, defined as $cnotC = \mathrm{id}|qnot$. The final case makes two recursive calls to the *deltaTy* function, in order to create copies of the types σ and τ, and pairs the copies in the following circuit:



which uses $|\sigma \otimes \tau|$ heap qubits.

From this definition a function that copies lists of types can be developed, which is useful in the definition of the compiler. Lists of QML types are given the Haskell type $TCon \in [Ty]_R$, and the function which performs the copying is called *deltaTCon*. This is defined as:

$$\begin{aligned}
&deltaTCon && \in TCon \to TCirc\\
&deltaTCon\ [\cdot]_R && = TCirc\ [\cdot]_R\ [\cdot]_R\ 0\ 0\ (Wires\ [\,])\\
&deltaTCon\ [\sigma]_R && = deltaTy\ t\\
&deltaTCon\ tc@(\vec{t} : \tau) && = TCirc\ tc\ oTc\ |tc|\ 0\ \phi\\
&\quad \textbf{where } \phi && = (Seq\ (Seq\ (Wire\ \vec{p})\\
&\qquad\qquad\qquad (Par\ ((circ \circ deltaTCon)\ \vec{t})\\
&\qquad\qquad\qquad\quad ((circ \circ deltaTy)\ \tau)))\\
&\qquad\qquad (Wire\ \vec{p}))\\
&\qquad oTc && = (tcon2ty\ tc) \otimes (tcon2ty\ tc)\\
&\qquad \vec{p} && = permFromTy\ (tc \otimes tc)\ shuf\\
&\qquad shuf && = [0\,..\,st-1] +\!\!+ [st+1\,..\,2\times st] +\!\!+ [st, 2\times st+1]\\
&\qquad st && = slength\ \vec{t}
\end{aligned}$$

This function works in a similar way to *deltaTy*. If the type context is empty, then the empty typed circuit is returned. If the type context contains only one type then *deltaTy* is used to generate the typed circuit that copies that type. Otherwise, *deltaTy* is used to copy the type at the end of the list, τ, and a recursive call to *deltaTCon* is used to copy the remainder of the list, $\vec{t}$. These are then paired up correctly by the circuit ϕ, as in the case of *deltaTy*.

The final operation is a Haskell implementation of the **C** operation on contexts, called *deltaCon*, and implements the $\otimes$ operation on contexts, which is defined as:

$$\begin{aligned} \Gamma, x : \sigma \otimes \Delta, x : \sigma &= (\Gamma \otimes \Delta), x : \sigma \\ \Gamma, x : \sigma \otimes \Delta &= (\Gamma \otimes \Delta), x : \sigma, \text{ if } x \notin \operatorname{dom} \Delta \\ \bullet \otimes \Delta &= \Delta \end{aligned}$$

The *deltaCon* function takes two contexts as its arguments, Γ and Δ, and returns the typed circuit $\mathbf{C}_{(\Gamma,\Delta)} \in \mathbf{FQC}^{\circ}\ |\Gamma \otimes \Delta|\ (|\Gamma| \otimes |\Delta|)$, as defined in section 7.3. *deltaCon* has three base cases which shall be looked at first:

$$\begin{aligned} &\mathit{deltaCon} && \in \mathit{Con} \to \mathit{Con} \to \mathit{TCirc} \\ &\mathit{deltaCon}\ [\cdot]_R\ [\cdot]_R && = \mathit{TCirc}\ [\cdot]_R\ [\cdot]_R\ 0\ 0\ (\mathit{Wires}\ [\,]) \\ &\mathit{deltaCon}\ [\cdot]_R\ \vec{y} && = \mathit{TCirc}\ (\mathit{con2tcon}\ \vec{y})\ (\mathit{con2ty}\ \vec{y})\ 0\ 0\ (\mathit{idC}\ |\vec{y}|) \\ &\mathit{deltaCon}\ \vec{y}\ \ [\cdot]_R && = \mathit{TCirc}\ (\mathit{con2tcon}\ \vec{y})\ (\mathit{con2ty}\ \vec{y})\ 0\ 0\ (\mathit{idC}\ |\vec{y}|) \end{aligned}$$

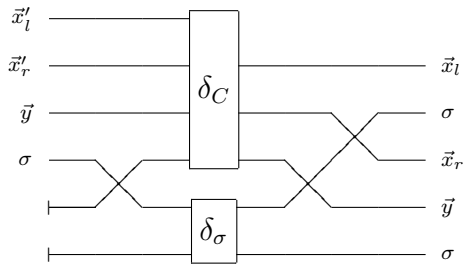
These cases state that if one context is empty then the circuit is simply the identity on the non-empty context. If both are empty, the empty circuit is returned. *con2tcon* is an auxiliary function that converts a list of variable name and type pairs (*Con*), into a list of types (*TCon*), and *con2ty* converts a context into a single type by tensoring all the types in the context together. The final case is given as:

$$\begin{aligned} &\mathit{deltaCon}\ \vec{x}\ (\vec{y} : (y, s)) = \\ &\quad \textbf{case}\ \mathit{splitAtN}\ (y, s)\ \vec{x}\ \textbf{of} \\ &\qquad \mathit{Nothing} \to \mathit{TCirc}\ (a : s)\ (b \otimes s)\ h\ 0\ (\mathit{Seq}\ (\mathit{Wire}\ \vec{p}) \\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{Par}\ \mathit{sub}\ (\mathrm{id}\ [s]_R))) \\ &\qquad\quad \textbf{where}\quad \mathit{TCirc}\ a\ b\ h\ \_\ \mathit{sub} = \mathit{deltaCon}\ \vec{x}\ \vec{y} \\ &\qquad\qquad\quad\ \ \mathit{iTy} = (a : s) \otimes \mathit{mkHeap}\ h \\ &\qquad\qquad\quad\ \ \mathit{sa}\ \ = \mathit{slength}\ a \end{aligned}$$

$$
\begin{aligned}
\vec{p} \quad &= permFromTy\ iTy \\
&\quad ([0\,..\,sa-1] +\!\!+ [sa+1\,..\,sa+h, sa]) \\
Just\ (\vec{x}_l, \vec{x}_r) \rightarrow\ &TCirc\ (a:s)\ oTy\ (h + size\ s)\ 0 \\
&\quad (Seq\ (Seq\ (Wire\ \vec{p}) \\
&\quad\quad (Par\ sub \\
&\quad\quad\quad (circ\ (deltaTy\ s)))) \\
&\quad\quad (Wire\ \vec{p}'))
\end{aligned}
$$

$$
\begin{aligned}
\textbf{where} \quad & TCirc\ a\ \_\ h\ \_\ sub = deltaCon\ (\vec{x}_l \otimes \vec{x}_r)\ \vec{y} \\
& iTy \quad = (a:s) \otimes mkHeap\ (h+1) \\
& oTy \quad = tcon2ty\ (dCout\ \vec{x}_l\ \vec{x}_r\ \vec{y}\ s) \\
& (\vec{p}, \vec{p}') = dCPerms\ iTy\ \vec{x}_l\ \vec{x}_r\ \vec{y}\ s
\end{aligned}
$$

If both contexts passed to the *deltaCon* contain types, then the fourth and final pattern is matched. This pattern examines the last name and type of the second context, and performs case analysis on the contexts using the auxiliary *splitAtN* function, which attempts to find the variable in the first context, Γ. If the variable is not found in Γ then the *Nothing* case is entered, which calls the *deltaCon* function on Γ and the remainder of Δ, and correctly wires up the subcircuit generated by this call. If the *splitAtN* function does find the variable y in the context Γ, then Γ is split at this point. The variable $y : s$ is then copied, using *deltaTy*, and the remainder of Γ and Δ are passed to *deltaCon* to generate the subcircuit δ_C, and the circuits are all then wired together as shown in the following circuit:



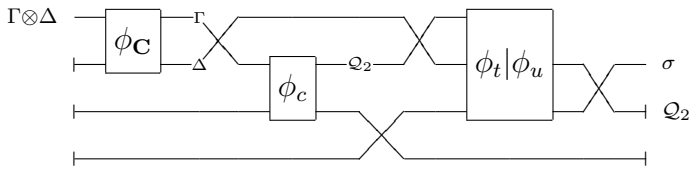
which correctly implements $\otimes$ on contexts. The auxiliary function *mkHeap* x, where $x \in \mathbb{N}$, simply generates a typed context of x qubits, for use as heap, while the function *dCPerms* calculates the permutations required to implement the circuit above. Finally, *dCout* calculates the output type of the *deltaCon* function.

7.6.8 Compiling the conditional and product rules in Haskell

The Haskell implementation of the classical conditional given by IF_{Op} follows the same pattern as the implementation of the LET^a_{Op} operation given in section 7.6.5. The compilation of an **if** term begins by compiling the subcircuits, as given by the rule for $[\![\Gamma \otimes \Delta \vdash \textbf{if } c \textbf{ then } t \textbf{ else } u : \sigma]\!]_{\text{Op}}$:

$$\frac{\begin{array}{c}\mathbf{c} \in \mathbf{FQC}^a\ \Gamma\ \mathcal{Q}_\mathbf{2} \\ \mathbf{t} \in \mathbf{FQC}^a\ \Delta\ \sigma \\ \mathbf{u} \in \mathbf{FQC}^a\ \Delta\ \sigma\end{array}}{\text{IF}_{\text{Op}}\ \mathbf{c}\ \mathbf{t}\ \mathbf{u} \in \mathbf{FQC}^a\ (\Gamma \otimes \Delta)\ \sigma}$$

The subcircuits **t** and **u** are then combined using a conditional circuit, by the auxiliary function *condOp*, to give $\mathbf{t}|\mathbf{u}$ The context operation **C** is performed by the *deltaCon* function. The results of these calculations are then combined to give the **FQC** morphism shown in the circuit below:



The interpretation of IF_{Op} in Haskell is therefore given by the following commented code:

```
compileTm c Γ (If b t u) mTy =
  do cb          ← compileTm c Γ b (Just Q₂)   -- Compile b
     ct          ← compileTm c Γ t mTy         -- Compile t
     cu          ← compileTm c Γ u mTy         -- Compile u
     (bC, tuC) ← pSubCon Γ (uVar cb) (uVar ct)
                                               -- Calculate context
                                               -- for subterms
     ftu         ← condOp (fqc ct) (fqc cu)    -- Compiles the circuit
                                               -- t|u ∈ σ ⊗ Q₂
     let fdC = deltaCon bC tuC                 -- Calculate deltaCon (C)
```

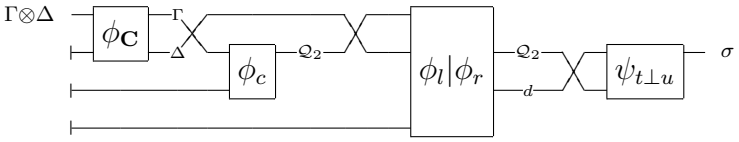
$$
\begin{array}{lll}
fb & = fqc\ cb & \text{-- Typed circuit for } b \\
gb & = gbS\ fb & \text{-- Size of garbage for } b \\
uCb & = uVar\ cb & \text{-- Context used} \\
 & & \text{-- in conditional} \\
(uCt, uCu) & = (uVar\ ct, uVar\ cu) & \text{-- Context used in branches} \\
(hb, h_{\mathbf{t|u}}) & = (hpS\ fb, hpS\ ftu) & \text{-- Heap sizes} \\
(sg, sd) & = (|bC|, |tuC|) & \text{-- Context sizes} \\
hS & = hb + h_{\mathbf{t|u}} + hpS\ fdC & \text{-- Total heap} \\
\phi & = Seq\ (Seq\ (Seq\ (Seq\ (Par\ (circ\ fdC) & \\
 & \qquad (Wire\ [0\,..\,hb + h_{\mathbf{t|u}} - 1])) & \\
 & \quad (permN\ 0\ sg\ sd\ (hb + h_{\mathbf{t|u}}))) & \\
 & \quad (Par\ (Par\ (idC\ sd)\ (circ\ fb))\ (idC\ h_{\mathbf{t|u}}))) & \\
 & \quad (permN\ (sd + 1)\ gb\ h_{\mathbf{t|u}}\ 0)) & \\
 & \quad (Par\ (circ\ ftu)\ (idC\ hb)) & \\
treturn\ mTy & \quad (Comp\ (uniqueCon\ uCb\ (uCt +\!\!+ uCu)) & \\
 & \qquad (TCirc\ (inT\ fdC)\ (outT\ ftu)\ hS\ 1\ \phi)) &
\end{array}
$$

The auxiliary function *condOp* calculates $\mathbf{t|u}$, and performs rudimentary type checking on the branches:

$$
\begin{array}{lll}
condOp \in TCirc \to TCirc \to Error\ TCirc & & \\
condOp\ tTC\ uTC & & \\
\quad = \mathbf{do\ let}\ (sit, siu) & = (|inT\ tTC|, |inT\ uTC|) & \text{-- } t, u \text{ input sizes} \\
(sot, sou) & = (|outT\ tTC|, |outT\ uTC|) & \text{-- } t, u \text{ output sizes} \\
(h_{\mathbf{t}}, h_{\mathbf{u}}) & = (hpS\ tTC, hpS\ uTC) & \text{-- } t, u \text{ heap sizes} \\
g_{\mathbf{t}} & = gbS\ tTC & \text{-- garbage size} \\
\phi_{a|b} & = Seq\ (Seq\ (permN\ 0\ sit\ 1\ h_{\mathbf{t}}) & \\
 & \qquad (Cond\ (circ\ tTC)\ (circ\ uTC))) & \\
 & \quad (permN\ 0\ 1\ sot\ 0) & \text{-- conditional circuit} \\
 & eguard\ (sot \equiv sou) & \texttt{"If: Branch output"} \\
 & eguard\ (sit + h_{\mathbf{t}} \equiv siu + h_{\mathbf{u}}) & \texttt{"If: Branch input"} \\
 & eguard\ (g_{\mathbf{t}} \equiv 0) & \texttt{"If: Garbage"}
\end{array}
$$

$$return\ (TCirc\ ((inT\ tTC) : \mathcal{Q}_{\mathbf{2}})$$
$$(outT\ tTC)\ h_{\mathbf{t}}\ 1\ \phi_{a|b})$$

The implementation in Haskell of the operational semantics of **if**°, given by $\text{IF}^{\circ}_{\text{Op}}$, is almost identical to that given for IF_{Op} above. The differences are that the **if**° term also contains an orthogonality judgement, described in section 7.4.5. Interpreting the orthogonality judgement gives the tuple (c, l, r, ψ), where $c \in \mathbb{N}$, $l \in \mathbf{FQC}^{\circ}\ \Gamma\ c$, $r \in \mathbf{FQC}^{\circ}\ \Gamma'\ c$, and $\psi \in \mathbf{FQC}^{\simeq}\ \sigma\ (\mathcal{Q}_{\mathbf{2}} \otimes c)$. These are used to create the following circuit, which is an extension of the interpretation of the **if** circuit:



The Haskell code for compiling **if**°, $compileTm\ \ c\ \ \Gamma\ \ (If^{\circ}\ \ b\ \ t\ \ u\ \ o)\ \ mTy$, is not included here, and the reader is directed to the similar code for the comparable **if** implementation. The Haskell implementation of the product rules will also be omitted, as they follow exactly the circuits presented in 7.4.2, and are translated into Haskell using the same concepts as presented for VAR, SUP, LET and IF. Code for interpreting the product rules, and for the auxiliary functions used above, will be made available from the QML project website [30].

7.6.9 Interpreting orthogonality in Haskell

Whenever an operation that requires an orthogonality is used, it is marked with an orthogonality judgement. These judgements are given by the datatype $Orth$, which follows the cases developed in section 7.4.5:

$$\begin{aligned}\textbf{data}\ Orth = {} & OBit_0 && \mid OBit_1 \\ \mid {} & OPair_0\ Orth && \mid OPair_1\ Orth \\ \mid {} & OSup\ \ Orth\end{aligned}$$

where $OBit_0$ denotes the judgement **qfalse** $\perp$ **qtrue**, $OBit_1$ denotes the judgement **qtrue** $\perp$ **qfalse**, $OSup$ denotes an orthogonal superposition, and $OPair$ denotes a product judgement that is orthogonal in either the left components or the right. There

is scope for this type to be extended in future with further orthogonality rules, such as the $\perp\mathbf{if}^\circ$ rule suggested in section 6.5.5.

The only term which uses the orthogonality judgement in the simplified QML syntax used in this section is the If° term, $IFo\ \ Tm\ \ Tm\ \ Tm\ \ Orth$. The QML parser would generate the orthogonality judgement by examining the **then** and **else** branches of the conditional; t and u in the term $\mathbf{if}^\circ\ c\ \mathbf{then}\ t\ \mathbf{else}\ u$. It compares t and u to try and apply one of the rules presented in section 6.5.5. For example, if the term $t = (\mathbf{qtrue}, x)$ and $u = (\mathbf{qfalse}, y)$, where x, y are some undefined QML terms, then these would be judged orthogonal with the judgement $Orth = OPair_0\ OBit_1$, as the pairs t, u are have left components that are orthogonal by the $OBit$ rule.

The Haskell interpretation of the orthogonality judgements follows the development presented in section 7.4.5. The function that interprets an orthogonality judgement $Orth$ returns a tuple (c, l, r, ψ) where $c \in \mathbb{N}$, $l \in \mathbf{FQC}^\circ\ \Gamma\ c$, $r \in \mathbf{FQC}^\circ\ \Gamma'\ c$, and $\psi \in \mathbf{FQC}^\simeq\ \sigma\ (\mathcal{Q}_\mathbf{2} \otimes c)$. The derivation of this data is detailed in the following case analysis:

Orthogonality of pure qubits ($OBit$):

$$\frac{}{\mathbf{qfalse} \perp \mathbf{qtrue}}\ OBit_0 \qquad \frac{}{\mathbf{qtrue} \perp \mathbf{qfalse}}\ OBit_1$$

The interpretation of the above rules, discussed in section 7.4.5, is $(0, -, -, \psi)$, where $-$ denotes the empty circuit, which can be implemented as $Wires\ [\,]$ in Haskell. In the $OBit_0$ case the interpretation of $\mathbf{qfalse} \perp \mathbf{qtrue}$ is the identity circuit on one qubit, $\psi = Wires\ [0]$; and in the case of $OBit_1$, the interpretation of $\mathbf{qtrue} \perp \mathbf{qfalse}$ is negation, $\psi = Rot\ (0,1)\ (1,0)$.

Orthogonality of pairs ($OPair_0$):

$$\frac{t \perp u}{(t, v) \perp (u, w)}\ OPair_0$$

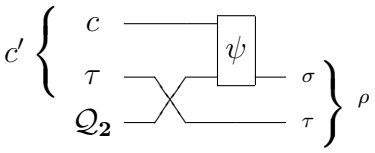
The interpretation of the orthogonality of $OPair_0\ x$ proceeds by first interpreting the orthogonality of the $Orth$ judgement x, which gives $[\![t \perp u]\!]^\perp_{\mathrm{Op}} =$

(c, l, r, ψ). Using this derivation, the interpretation of $(t, v) \perp (u, w)$ can be given as (c', l', r', ψ'). The terms l' and r' are given as:

$$l' = \mathit{Pair}\ l\ v \quad \text{and} \quad r' = \mathit{Pair}\ r\ w$$

using Haskell QML notation, which, when interpreted using *compileTm*, gives the circuits shown in section 7.4.5. The value c', which is defined as $c \otimes |\tau|$, is the output type of l' (and also r'), and can be extracted from the output of the compilation of l' (or r').

Finally, the interpretation of ψ' is given by the circuit:



which can be translated directly into a Haskell quantum circuit.

In the case of OPair_1, the values of c and ψ are calculated in exactly the same way, but the interpretation of the products l' and r' components are swapped: l' becomes $\mathit{Pair}\ v\ l$, and r' is interpreted as $\mathit{Pair}\ w\ r$.

Orthogonality of superpositions (*OSup*):

$$\frac{t \perp u \qquad \lambda_0^* \kappa_0 = -\lambda_1^* \kappa_1}{\lambda_0 \times t + \lambda_1 \times u \perp \kappa_0 \times t + \kappa_1 \times u}\ \mathit{OSup}$$

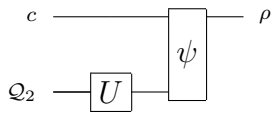
The interpretation of the *OSup* x rule above proceeds by first interpreting the orthogonality of the *Orth* judgement $t \perp u$, given as x, to derive $[\![t \perp u]\!]^{\perp}_{\mathrm{Op}} = (c, l, r, \psi')$. The evaluation of *OSup* x uses the c, l and r derived from this judgement directly to return (c, l, r, ψ). The circuit ψ' is used to define ψ, with the auxiliary circuit $\phi = \mathit{rot}\ U \in \mathbf{FQC}^{\simeq}\ \mathcal{Q}_2$, where U is given by the unitary matrix:

$$U = \begin{pmatrix} \lambda_0 & \lambda_1 \\ \kappa_0 & \kappa_1 \end{pmatrix}$$

Translated into Haskell circuit notation, ψ is given by:

$$\begin{array}{l} Seq\ (Par\ (idC\ c) \\ \qquad\qquad (Rot\ (\lambda_0, \lambda_1)\ (\kappa_0, \kappa_1))) \\ \psi' \end{array}$$

which is given by the following circuit diagram:



which matches the circuit and definitions given in section 7.4.5.

7.7 Summary

This chapter presents an operational semantics for the quantum functional programming language QML. A category of QML terms is introduced, and for each term in QML a mapping into the category **FQC** is defined. As each QML term can be mapped into an **FQC** morphism, and each **FQC** morphism defines a quantum circuit, this mapping gives an operational semantics to QML terms and programs. The interpretation of context operations is also discussed, as is the quantum control operator **if**$^\circ$ and the interpretation of the orthogonality judgements.

An interpretation of the QML term syntax is then presented in Haskell. From this a compiler for QML programs is implemented. The compiler works by mapping each Haskell QML term into a typed circuit representation of **FQC** morphisms, using the interpretations developed earlier in the chapter and implementing them using Haskell.

Chapter 8

Further research

This chapter presents some possible extensions to the research presented in this thesis. Both active areas of research and outlines of new concepts and ideas are explored. An outline of the compositionality of QML is presented in section 8.1. This is achieved by directly giving an interpretation of QML programs in the category **Q** of superoperators, then by showing that this interpretation agrees with the denotational semantics given in section 7.5, which factors through the operational semantics.

8.1 Compositionality of QML

The principle of compositionality states that the denotation of each expression in a language can be defined purely in terms of the denotations of its subexpressions. Compositionality for QML can be demonstrated by defining a direct denotational semantics for QML, which translates QML terms into superoperators without first calculating the operational semantics. It can then be shown that this agrees with the denotational semantics presented in section 7.5. This is summarised by the diagram in figure 8.1, which should be compared with figure 7.1 (in section 7.5), which shows how the denotational semantics is defined via the operational semantics. The definition of the direct denotational semantics will now be summarised.

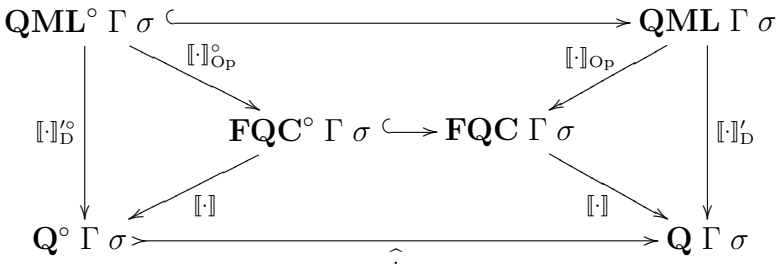


FIGURE 8.1: If this diagram commutes then QML is compositional, as outlined in section 8.2.5

8.2 A direct denotational semantics for QML

A definition of a direct denotational semantics for QML would proceed in a very similar way as the definition of the (direct) operational semantics, presented in chapter 7. The interpretation of types and contexts must again be defined before the QML judgements can be interpreted. This is achieved using the same functions defined in section 7.2 for the operational semantics, which interpret types and contexts by their sizes. The size of a type σ is given by $|\sigma|$, and the size of a context Γ is given by $|\Gamma| = |\overline{\Gamma}|$. The size function will again be omitted for clarity, when it is clear from the context that the size of the type is intended. Types and contexts are therefore interpreted in the denotational semantics thus:

$$\begin{aligned} [\![\Gamma]\!] &= |\Gamma| \\ [\![\sigma]\!] &= |\sigma| \end{aligned}$$

Both strict and non-strict QML programs are given interpretations by the direct denotational semantics. Strict $\mathbf{QML}^\circ$ program derivations are interpreted in the category $\mathbf{Q}^\circ$ of isometries, while non-strict $\mathbf{QML}$ programs are interpreted in the category $\mathbf{Q}$ of superoperators. Strict QML derivations $\frac{d}{\Gamma \vdash^\circ t:\sigma}$ are interpreted by the semantic function:

$$[\![d]\!]'^\circ_{\mathrm{D}} \in \mathbf{QML}^\circ\ \Gamma\ \sigma \to \mathbf{Q}^\circ\ \Gamma\ \sigma$$

and non-strict QML derivations $\frac{d}{\Gamma \vdash t:\sigma}$ are interpreted by:

$$[\![d]\!]'_{\mathrm{D}} \in \mathbf{QML}\ \Gamma\ \sigma \to \mathbf{Q}\ \Gamma\ \sigma.$$

The denotational semantics of programs is defined in the same way as for the operational semantics.

8.2.1 Interpreting operations on contexts

There are at least two possible interpretations of the context operation $\otimes$. A direct interpretation of the operation $\otimes$, defined in section 6.5, could be developed. Alternatively, the language could be simplified by no longer allowing the implicit contraction of variables by the $\otimes$ operation on contexts. This simplification of QML, called $\mathbf{QML}^{-}$, defines QML without implicit contraction. The operation $\otimes$ on contexts in $\mathbf{QML}^{-}$ is given by:

$$\begin{array}{lcll} \Gamma, x : \sigma \otimes \Delta & = & (\Gamma \otimes \Delta), x : \sigma & \text{if } x \notin \operatorname{dom} \Delta \\ \bullet \otimes \Delta & = & \Delta & \end{array}$$

The use of $\mathbf{QML}^{-}$ simplifies the denotational semantics, as it is now the case that the following rule holds:

$$[\![\Gamma \otimes \Delta]\!]'_{\mathrm{D}} = [\![\Gamma]\!]'_{\mathrm{D}} \otimes [\![\Delta]\!]'_{\mathrm{D}}$$

which is not true with the standard QML interpretation, as Γ and Δ may each contain a variable which will only appear once in the standard QML interpretation of $\Gamma \otimes \Delta$.

Using $\mathbf{QML}^{-}$ it is still possible to represent any terms which can be defined in **QML**, but now all contractions must be made explicit. For example, the contraction of a single qubit can be made explicit, as shown in the following example $\mathbf{QML}^{-}$ program:

$$\begin{array}{l} \delta_{\mathcal{Q}_2} \quad : \mathcal{Q}_{\mathbf{2}} \multimap \mathcal{Q}_{\mathbf{2}} \otimes \mathcal{Q}_{\mathbf{2}} \\ \delta_{\mathcal{Q}_2}\ x = \mathbf{if}^{\circ}\ x\ \mathbf{then}\ (\mathbf{qtrue}, \mathbf{qtrue}) \\ \qquad\qquad\quad \mathbf{else}\ \ (\mathbf{qfalse}, \mathbf{qfalse}) \end{array}$$

Larger types can be explicitly copied by a $\mathbf{QML}^{-}$ program such as the following, which uses pattern matching:

$$\begin{array}{ll} \delta_\sigma & : \sigma \multimap \sigma \otimes \sigma \\ \delta_\sigma\ x & = \delta_{\mathcal{Q}_2}\ x \\ \delta_\sigma\ (x, y) & = \mathbf{let}\ (x_0, x_1) = \delta_\sigma\ x \end{array}$$

$$(y_0, y_1) = \delta_\sigma\ y$$
$$\textbf{in}\ ((x_0, y_0), (x_1, y_1))$$

In this way it is possible to define a translation from **QML** into $\mathbf{QML}^-$, which would then allow the direct denotational semantics for $\mathbf{QML}^-$ to be applied to QML programs.

8.2.2 The denotational semantics of QML rules

In chapter 7, the operational semantics is presented by interpreting each rule in QML by a different **FQC** morphism. The direct denotational semantics can be defined in a similar way; by defining, for each strict term in QML, an associated isometry in $\mathbf{Q}^\circ$, and for each non-strict term, a superoperator in **Q**. Non-strict terms produce garbage, and in the denotational semantics garbage is managed using the partial trace superoperator, hence the interpretation of non-strict terms is given in the category **Q** of superoperators. Strict terms produce no garbage, and can be interpreted in the category $\mathbf{Q}^\circ$ of isometries.

Unlike with the operational semantics, presented in section 7.4, the embedding of strict terms into non-strict terms in the denotational semantics is not invisible. The embedding of the denotation of a strict term $f \in \mathbf{Q}^\circ\ a\ b$ is achieved by the lifting operation from isometries to superoperators, defined in section 5.5.1, giving $\widehat{f} \in \mathbf{Q}\ a\ b$. Each QML term initialises its own heap, and traces out its own garbage. Heap initialisation is achieved using the isometry 0^h introduced in section 5.4.1, and the removal of garbage is managed by use of the partial trace superoperator Tr, introduced in section 5.5.2.

The direct denotational semantics of several rules will now be considered, with the full definition, including a detailed analysis of orthogonality in the denotational semantics, left to future work. In this presentation the semantics of the $\otimes$ operation on contexts will be omitted, see section 8.2.1.

8.2.3 Interpreting the structural rules

The structural rules of QML are defined in section 6.5.1. The strict variable rule, $\mathbf{var}^\circ$, is interpreted by $[\![x : \sigma \vdash^\circ x : \sigma]\!]'^\circ_{\mathrm{D}}$ as $\textsc{var}^\circ_{\mathrm{D}}$, which is defined as the following isometry:

$$\textsc{var}^\circ_{\mathrm{D}} \in \mathbf{Q}^\circ\ \sigma\ \sigma$$
$$\textsc{var}^\circ_{\mathrm{D}} = \mathrm{id}_\sigma$$

This is the identity morphism, defined as $\mathrm{id}_\sigma\ a = a \in \sigma \to \sigma$.

The non-strict variable rule, $\mathbf{var}$, is interpreted by $[\![\Gamma, x : \sigma \vdash x^{\mathrm{dom}\,\Gamma} : \sigma]\!]'_{\mathrm{D}}$ as $\textsc{var}_{\mathrm{D}}$, which is defined as the following superoperator:

$$\textsc{var}_{\mathrm{D}} \in \mathbf{Q}\ (\Gamma, \sigma)\ \sigma$$
$$\textsc{var}_{\mathrm{D}} = \mathrm{Tr}_\Gamma$$

which uses the partial trace operator defined in section 5.5.2 to trace out Γ, which is the method by which garbage is dealt with in the operational semantics.

As both strict and non-strict terms are interpreted in different categories, based on the schematic variable $a \in \{\circ, -\}$, the $\mathbf{let}^a$ rule has two interpretations based on a. The rule $\mathbf{let}^a$ is given as:

$$\frac{\begin{array}{c}\Gamma \vdash^a t : \sigma \\ \Delta, x : \sigma \vdash^a u : \tau\end{array}}{\Gamma \otimes \Delta \vdash^a \mathbf{let}\, x = t\, \mathbf{in}\, u : \tau}\ \mathbf{let}^\circ$$

This rule is interpreted by the denotation function $[\![\Gamma \otimes \Delta \vdash^a \mathbf{let}\, x = t\, \mathbf{in}\, u : \tau]\!]'^a_{\mathrm{D}}$ as $\textsc{let}^a_{\mathrm{D}}$, given by:

$$\frac{\begin{array}{c}\mathbf{t} \in \mathbf{Q}^a\ \Gamma\ \sigma \\ \mathbf{u} \in \mathbf{Q}^a\ (\Delta \otimes \sigma)\ \tau\end{array}}{\begin{array}{l}\textsc{let}^\circ_{\mathrm{D}}\ \mathbf{t}\ \mathbf{u} \in \mathbf{Q}^a\ (\Gamma \otimes \Delta)\ \tau \\ \textsc{let}^\circ_{\mathrm{D}}\ \mathbf{t}\ \mathbf{u} = \mathbf{u} \circ (\mathbf{t} \otimes \mathrm{id}_\Delta)\end{array}}$$

where $\mathbf{t} = [\![t]\!]'^a_{\mathrm{D}}$ and $\mathbf{u} = [\![u]\!]'^a_{\mathrm{D}}$, and each is either an isometry or a superoperator depending on the schematic variable a.

8.2.4 Interpreting products ($\otimes$)

The QML product rules have been introduced in section 6.5.2, with the operational semantics given in section 7.4.2. In the operational semantics, the interpretation of the $\mathcal{Q}_\mathbf{1}$ introduction and weakening rules were invisible, as they were modelled by zero wires in the circuit model. In the denotational semantics the $\mathcal{Q}_\mathbf{1}$ introduction is interpreted by $[\![\bullet \vdash^\circ () : \mathcal{Q}_\mathbf{1}]\!]'^\circ_\mathrm{D}$ as $\textsc{unit}^\circ_\mathrm{D}$, where:

$$\textsc{unit}^\circ_\mathrm{D} \in \mathbf{Q}^\circ\ 0\ \mathcal{Q}_\mathbf{1}$$
$$\textsc{unit}^\circ_\mathrm{D} = \mathrm{id}_{\mathcal{Q}_\mathbf{1}}$$

Weakening of the tensor unit $\mathcal{Q}_\mathbf{1}$ is interpreted in the same way.

The rule for $\otimes$ introduction, or pairing, is given as:

$$\frac{\Gamma \vdash^a t : \sigma \quad \Delta \vdash^a u : \tau}{\Gamma \otimes \Delta \vdash^a (t, u) : \sigma \otimes \tau}\otimes \text{ intro}$$

The denotational interpretation of $\otimes$ introduction merges the components of the pairing using the appropriate tensor product to produce an isometry if the pairing is strict, and otherwise gives a superoperator. The evaluation of $[\![\Gamma \otimes \Delta \vdash^a (t, u) : \sigma \otimes \tau]\!]'^a_\mathrm{D}$ is given by $\textsc{pair}^a_\mathrm{D}$ as follows:

$$\frac{\mathbf{t} \in \mathbf{Q}^a\ \Gamma\ \sigma \qquad \mathbf{u} \in \mathbf{Q}^a\ \Delta\ \tau}{\begin{array}{l}\textsc{pair}^a_\mathrm{D}\ \mathbf{t}\ \mathbf{u} \in \mathbf{Q}^a\ (\Gamma \otimes \Delta)\ (\sigma \otimes \tau) \\ \textsc{pair}^a_\mathrm{D}\ \mathbf{t}\ \mathbf{u} = \mathbf{t} \otimes \mathbf{u}\end{array}}$$

where $[\![t]\!]'^a_\mathrm{D} = \mathbf{t}$ and $[\![u]\!]'^a_\mathrm{D} = \mathbf{u}$. If both $\mathbf{t}$ and $\mathbf{u}$ are isometries, and are therefore strict, then so is $\textsc{pair}^a_\mathrm{D}$, as given by the schematic variable a. Otherwise, $\textsc{pair}^a_\mathrm{D}$ is modelled by a superoperator.

As in the operational semantics, the denotational semantics of the $\otimes$ elimination rule is similar to the denotational interpretation of the let-rule. The $\otimes$ elimination rule is given as:

$$\frac{\begin{array}{c}\Gamma \vdash^a t : \sigma \otimes \tau \\ \Delta,\ x : \sigma, y : \tau \vdash^a u : \rho\end{array}}{\Gamma \otimes \Delta \vdash^a \mathbf{let}\ (x, y) = t\ \mathbf{in}\ u : \rho}\otimes \text{ elim}$$

This rule is interpreted in the denotational semantics as $[\![\Gamma \otimes \Delta \vdash^a \textbf{let } (x, y) = t \textbf{ in } u : \rho]\!]'^a_{\mathrm{D}}$ by $\textsc{letp}^a_{\mathrm{D}}$:

$$\frac{\mathbf{t} \in \mathbf{Q}^a\ \Gamma\ (\sigma \otimes \tau) \qquad \mathbf{u} \in \mathbf{Q}^a\ (\Delta \otimes \sigma \otimes \tau)\ \rho}{\textsc{letp}^a_{\mathrm{D}}\ \mathbf{t}\ \mathbf{u} \in \mathbf{Q}^a\ (\Gamma \otimes \Delta)\ \rho}$$
$$\textsc{letp}^a_{\mathrm{D}}\ \mathbf{t}\ \mathbf{u} = \mathbf{u} \circ (\mathbf{t} \otimes \mathrm{id}_\Delta)$$

where $[\![t]\!]'^a_{\mathrm{D}} = \mathbf{t}$ and $[\![u]\!]'^a_{\mathrm{D}} = \mathbf{u}$. As with the introduction rule, if both $\mathbf{t}$ and $\mathbf{u}$ are isometries (strict) then so is $\textsc{pair}^a_{\mathrm{D}}$; otherwise $\textsc{pair}^a_{\mathrm{D}}$ is modelled by a superoperator, again given by the schematic variable a.

The rules for superpositions and conditionals, and an interpretation of orthogonality judgements, can be defined in the same way.

8.2.5 Showing compositionality for QML

Compositionality is an important feature of any denotational semantics. The principle of compositionality, applied to programming languages, is that the denotation of a program can be constructed from the denotation of its parts. For example, given some expression such as $term_1 \oplus term_2$, compositionality would be shown by giving a meaning for the entire expression, in terms of the meaning of $term_1$ and $term_2$, where $\oplus$ is some operator in the language. In QML, this corresponds to showing that the direct denotational semantics outlined in section 8.2 agrees with the interpretation presented in section 7.5, which uses the operational semantics to generate **FQC** morphisms and then interprets these in the category **Q** of superoperators. This can be summarised as:

$$[\![d]\!]'_{\mathrm{D}} = [\![[\![d]\!]_{\mathrm{Op}}]\!]$$

where $[\![[\![d]\!]_{\mathrm{Op}}]\!]$ is the denotational semantics $[\![\cdot]\!]_{\mathrm{D}}$ presented in section 7.5, which uses the operational semantics given by $[\![\cdot]\!]_{\mathrm{Op}}$. This can be shown by proving that it holds for each element of the operational and denotational semantics. This is achieved by showing the equations such as the following hold for each object in the denotational

semantics:

$$\begin{aligned}
\mathrm{VAR}_{\mathrm{D}} &= [\![\mathrm{VAR}_{\mathrm{Op}}]\!] \\
\mathrm{VAR}^{\circ}_{\mathrm{D}} &= [\![\mathrm{VAR}^{\circ}_{\mathrm{Op}}]\!] \\
\mathrm{LET}^{a}_{\mathrm{D}}\ t\ u &= [\![[\![\mathrm{LET}^{a}_{\mathrm{Op}}\ t\ u]\!]_{\mathrm{Op}}]\!] \\
\mathrm{PAIR}^{a}_{\mathrm{D}}\ t\ u &= [\![[\![\mathrm{PAIR}^{a}_{\mathrm{Op}}\ t\ u]\!]_{\mathrm{Op}}]\!] \\
\mathrm{LETP}^{a}_{\mathrm{D}}\ t\ u &= [\![[\![\mathrm{LETP}^{a}_{\mathrm{Op}}\ t\ u]\!]_{\mathrm{Op}}]\!] \\
&\ldots
\end{aligned}$$

In many of these cases compositionality follows directly from proposition 2 in section 5.5.4, and the observation that only horizontal ($\circ$) and vertical ($\otimes$) composition are used in the definition of the interpretation of terms from their components. It also must be shown that the partial trace, which is only used at the end of the computation when interpreting via the operational semantics, commutes. This is an area of further research, as the full direct denotational semantics of QML has not yet been finalised.

8.3 Infinite data structures and recursion in QML

The version of QML defined in this thesis is finite dimensional, but future revisions could be extended to allow infinite dimensional constructions. Infinite data structures could be interpreted in infinite-dimensional vector spaces using the standard approaches taken from mathematical physics. An alternative, which is closer to potential practical implementations of quantum computational devices, is to allow quantum programs to be indexed by classical structures in a way akin to that proposed in the language Dependent ML (DML) [86]. DML is a language with dependent types where index expressions and actual programs are clearly separated. In the case of DML, this separation is required to deal with impurities in the actual programs, such as non-termination. In a dependently typed version of QML, the same approach may be used to separate the classical structure of the computation from quantum effects.

Recursion could be implemented in QML by indexing types by a classical structure (the natural numbers, for example). Recursive QML programs could then be written by recurring over the size of these structures, such as in classical circuit complexity. These structures could be generalised by other types, but would be compiled to qubits

at runtime. A scheme such as this would allow recursion over the structure of the data, which covers most forms of recursion, but is not sufficient for programs using recursion as a quantum control structure. For example, using this scheme a while-loop over a quantum data structure could not be implemented. It is as yet unknown whether this kind of recursion is required for completeness, and how to understand conditional quantum recursion is an active area of research [35].

8.4 Other areas of further research

The extension of QML to support higher-order programming is a possibly fruitful area of research. Higher-order programs could more accurately reflect how many quantum algorithms are presented. For example, the Quantum Fourier Transform (discussed in appendix A) can be parameterised by a function on quantum words. Selinger investigated this problem [70] and found that there is currently no known canonical higher-order structure on $\mathbf{Q}$. However, it may prove interesting to investigate whether the category of presheaves over $\mathbf{Q}$ would provide a sound denotational model for higher-order quantum computations. Tensor products could be interpreted by Day's construction, which is automatically closed with respect to this tensor product. There is no clear candidate for coproducts, $\oplus$, and since it is not obvious how a coproduct of higher order quantum functions may be implemented, the best choice may be not to allow this and limit $\oplus$ to first order types. These topics are worthy of further consideration.

A further possible addition includes making QML basis independent, which is useful for the expression of many algorithms, and is used in other models of quantum computation such as the one-way quantum computer. This could be achieved using a notion of *views*, such as that used by McBride and McKinna in the dependently typed language Epigram [49].

There is also scope to develop a new operational semantics for QML using the one-way quantum computer (measurement calculus) model, and for the further development of the orthogonality judgement, in order to make it complete with respect to the denotational semantics. As discussed in section 6.5.5, the direct use of an

inner-product judgement on terms would be a step towards achieving this.

Green and Altenkirch have sketched the first steps towards a theory of irreversible computation based on reversible computation [31]. Several laws are proposed for translating morphisms in the categories **FCC** and **FQC**, and it is shown that these laws are sufficient to derive van Neumann's measurement postulate. Research is ongoing in this area to try and establish a more abstract presentation of the laws and to discover whether they are in fact complete for quantum computation, *i.e.* whether the three laws presented can characterise equality of definable, irreversible quantum circuits.

Finally, in collaboration with Altenkirch, Vizzotto, and Sabry, an equational theory for a pure fragment of QML has been developed [2]. This work introduces high-level reasoning principles which could be expressed as an algebra of quantum programming. This algebra allows mathematically clear, formal correctness proofs of pure QML programs, and also introduces a normalisation algorithm for QML. In future this work could be extended to cover the full language, including measurement, to provide a normalisation function and equational reasoning principles for the entire language.

8.5 Summary

This chapter outlines topics in this thesis which could be possibly improved or extended. Included here is an outline of a direct denotational semantics for QML, directly into superoperators, and a possible method by which this semantics could be used to prove compositionality. The chapter concludes with a brief discussion of other possible areas of future research, including recursion and infinite dimensions, basis independence, a possible coproduct structure, and higher–order types.

Chapter 9

Summary and conclusions

This thesis introduces the finite quantum programming language QML, which features both quantum control and quantum data. Weakenings affect the behaviour of a quantum program, and this is identified as one of the main structural differences between quantum and classical programming. Consequently QML uses a strict linear type system where weakenings have to be made explicit. Forgetting information may affect other parts of the computation, and this necessitates the use of an orthogonality judgement. This judgement ensures that the quantum control operator **if**$^\circ$ does not irreversibly dispose of information by providing a witness to orthogonality.

The thesis begins with a review of reversible classical and reversible quantum computing, providing background information about quantum computing, and includes a description of basic linear algebra. These topics are presented in a progression from simple classical systems through to irreversible quantum computations modelled by superoperators in the category **Q**, with each development explained and accompanied by an implementation in the functional programming language Haskell [55]. Chapter 2 gives a presentation of reversible classical computation, including both theory and implementations. Chapter 3 extends chapter 2 with a discussion of reversible quantum computation, again including presentations of both the theory of quantum computation and an implementation given in terms of the quantum circuit model, which highlights the similarities and differences between classical and quantum reversible computation. The developments in these two chapters are given categorical interpretations in chapters 4 and 5, and both are extended with the concept of ir-

reversible computations based on the appropriate reversible framework. Chapter 4 gives a categorical interpretation of classical computation as morphisms in the category **FCC** (Finite Classical Computation). A mathematical interpretation of **FCC** morphisms is also presented, by defining a functor from **FCC** to the category **FinSet** of finite sets. By analogy with this interpretation of classical computation, chapter 5 presents the categorical interpretation of quantum computations as morphisms in the category **FQC** (Finite Quantum Computation), and includes a development of a mathematical interpretation of this category in terms of isometries and superoperators, which is also functorial. Using these categorical interpretations, a formalisation of the quantum circuit model is defined, which is a contribution of this thesis.

The language QML is defined in chapter 6. This chapter presents a discussion of possible interpretations of weakening and contraction in quantum programming languages, and the motivations of the design choices made in defining QML. Quantum data and quantum control as features of a language are also presented. The grammar of the language is defined, and the structural rules are stated. This chapter also includes implementations of several quantum algorithms as QML programs.

An operational semantics of QML, presented in terms of reversible quantum circuits, is given in chapter 7. These circuits model irreversible computation by using initialised heap registers at the start of the computation, and a notion of garbage which allows ancillary data to be disposed of at the end. The operational semantics has been implemented in Haskell, and is given by the category **FQC**. A denotational semantics for QML is also presented by factoring through the operational semantics, and using the translation from **FQC** morphisms to superoperators in **Q**. This follows the ideas of Selinger's denotational semantics for QPL [69]. An implementation of the operational semantics and denotational semantics of QML is also discussed, developed in Haskell.

The presentation in this thesis clearly separates the operational semantics from the denotational semantics. **FQC** morphisms (computations) are identified up to monoidal identities (in other words, isomorphic circuit diagrams), and the extensional equality of QML terms is defined in the denotational model in the category **Q**. It is also shown that the denotational semantics is derivation independent, *i.e.* different

typing derivations of the same term do not effect the interpretation up to extensional equality.

In chapter 8 a brief outline of how compositionality can be shown for QML is presented. This is achieved by defining a direct denotational semantics, and shows that replacing extensionally equivalent sub-terms results in extensionally equivalent programs. Chapter 8 also includes a discussion of other possibly interesting extensions to QML, such as introducing higher–order types and making the language basis independent.

This thesis has contributed the functional quantum programming language QML which features both basic quantum data structures and quantum control structures. In particular QML includes a quantum *if* construct which analyses quantum data without measuring, and hence without changing the data and therefore preserving any superposition or entanglement. QML thus differs from other work in quantum programming, as it allows both quantum data *and* quantum control.

The design of the language was inspired by taking a classical reversible model of computation, and exploring where a similarly designed quantum model differs. A categorical semantics of QML is presented by interpreting terms as morphisms in the category **FQC** of finite quantum computations. The **FQC** semantics gives rise to a denotational semantics in terms of superoperators, in the category **Q**, which is the accepted domain of irreversible quantum computation. The **FQC** semantics also gives rise to a compiler into quantum circuits, an accepted operational semantics for quantum programs, which is given a formal interpretation in chapter 5 The denotational semantics supports reasoning and allows optimisation principles to be applied, and therefore facilitates the expression and analysis of proofs and theorems.

9.1 List of publications

T. Altenkirch and J. Grattage. A functional quantum programming language [3]. In *20th Annual IEEE Symposium on Logic in Computer Science*, 2005. Also arXiv:quant-ph/0409065.

T. Altenkirch, J. Grattage, J. K. Vizzotto, and A. Sabry. An algebra of pure quantum programming [2]. In P. Selinger, editor. Proceedings of the 3rd International Workshop on Quantum Programming Languages [71], Electronic Notes in Theoretical Computer Science. Elsevier Science, 2005.

T. Altenkirch and J. Grattage. QML: Quantum data and control. In preparation for publication in January 2007. http://sneezy.cs.nott.ac.uk/qml

References

[1] Samson Abramsky. A structural approach to reversible computation. *Theoretical Computer Science*, 347(3):441–464, 2005.

[2] T. Altenkirch, J. Grattage, J. K. Vizzotto, and A. Sabry. An algebra of pure quantum programming. In Selinger [71].

[3] T. Altenkirch and J. J. Grattage. A functional quantum programming language. In *20th Annual IEEE Symposium on Logic in Computer Science*, 2005. Also arXiv:quant-ph/0409065.

[4] T. Altenkirch and B. Reus. Monadic presentations of lambda terms using generalized inductive types. In *Computer Science Logic*, number 1683 in LNCS, pages 453–468, 1999.

[5] P. Arrighi and G. Dowek. Operational semantics for a formal tensorial calculus. *Proceedings of the International Workshop on Quantum Programming Languages*, pages 21–38, 2004.

[6] J. Baugh, O. Moussa, C. A. Ryan, R. Laflamme, C. Ramanathan, T. F.Havel, and D. G. Cory. Solid-state NMR three-qubit homonuclear system for quantum-information processing: Control and characterization. *Physical Review A*, 73(2):art. 022305, 2006.

[7] P. Benioff. The computer as a physical system: a microscopic quantum mechanical model of computers as represented by Turing Machines. *Journal of Statistical Physics*, 22(5):563–591, 1980.

[8] P. Benioff. Quantum mechanical Hamiltonian models of Turing Machines that dissaipate no energy. *Physics Review Letters*, 48:1581–1585, 1982.

[9] S. C. Benjamin and S. Bose. Quantum computing with an always-on Heisenberg interaction. *Physical Review Letters*, 90(24):art. 247901, 2003.

[10] C. H. Bennett. Logical Reversibility of Computation. *IBM Journal of Research and Development*, 17(6):525–532, 1973.

[11] C. H. Bennett. Dissipation-error tradeoff in proofreading. *BioSystems*, 11:85–91, 1979.

[12] E. Biolatti, R. C. Iotti, P. Zanardi, and F. Rossi. Quantum information processing with semiconductor macroatoms. *Physical Review Letters*, 85(26):5647–5650, 2000.

[13] D. Bouwmeester, J. W. Pan, K. Mattle, M. Eibl, H. Weinfurter, and A. Zeilinger. Experimental quantum teleportation. *Nature*, 390(6660):575–579, 1997.

[14] K. A. Brickman, P. C. Haljan, P. J. Lee, M. Acton, L. Deslauriers, and C. Monroe. Implementation of Grover's Quantum Search Algorithm in a Scalable System, 2006. http://arxiv.org/abs/quant-ph/0510066.

[15] J. Brown. *Quest for the Quantum Computer*. Simon and Schuster, first touchstone edition edition, 2001.

[16] H. Buhrman, J. Tromp, and P. Vitányi. Time and space bounds for reversible simulation. *Lecture Notes in Computer Science*, 2076:1017–, 2001.

[17] I. Chuang, N. Gershenfeld, and M Kubinec. Experimental implementation of fast quantum searching. *Physical Review Letters*, 80(15):3408–3411, 1998.

[18] G. Cory, W. Mass, M. Price, E. Knill, R. Laflamme, W. H. Zurek, T. F. Havel, and S. S. Somaroo. Experimental Quantum Error Correction, 1998. http://arxiv.org/abs/quant-ph/9802018.

[19] V. Danos, E. Kashefi, and P. Panangaden. The Measurement Calculus. arXiv:quant-ph/0412135, 2004.

[20] L. Davidovich, N. Zagury, M. Brune, J. M. Raim, and S. Haroche. Teleportation of an atomic state between two cavities using non-local microwave fields. *Physical Review A*, 50(2):R895–R898, 1994.

[21] C. M. Dawson and M. A. Nielsen. The Solovay-Kitaev Theorem. *arXiv:quant-ph/0505030*, pages 1–15, 2005.

[22] D. Deutsch. Quantum theory, the Church-Turing principle and the universal quantum computer. *Proceedings of the Royal Society of London Ser. A*, A400:97–117, 1985. http://citeseer.ist.psu.edu/deutsch85quantum.html.

[23] A. Einstein, B. Podolsky, and N. Rosen. Can quantum-mechanical description of physical reality be considered complete? *Physical Review*, 47(10):0777–0780, 1935.

[24] L. Euler. Introductio in analysin infinitorum, 1748.

[25] R. Feynman. Simulating physics with computers. *International Journal of Theoretical Physics*, 21:6/7:467–468, 1982.

[26] E. Fredkin and T. Toffoli. Conservative Logic. *International Journal of Theoretical Physics*, 21:219–253, 1982.

[27] S. Gay. Quantum Programming Languages: Survey and Bibliography. *Mathematical Structures in Computer Science*, 16(4), 2006.

[28] S. Gay and R. Nagarajan. Communicating quantum processes. In *Proceedings of the 32nd ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*. ACM Press, 2005. Preliminary version in [68]; also arXiv:quant-ph/0409052.

[29] W. Gerlach and O. Stern. The directional quantisation in the magnetic field. *Annalen der physik*, 16(74), 1924.

[30] J. Grattage. Haskell code for a QML compiler into typed quantum circuits, http://sneezy.cs.nott.ac.uk/qml/compiler.

[31] A. Green and T. Altenkirch. From reversible to irreversible computations. In Peter Selinger, editor, *Proceedings of the 4th International Workshop on Quantum Programming Languages*, Electronic Notes in Theoretical Computer Science. Elsevier Science, 2006.

[32] L. Grover. Quantum Mechanics helps in searching for a needle in a haystack. *Physics Review Letters*, 79(2):325–328, 1997.

[33] J. Gruska. *Quantum Computing*. McGraw-Hill, Maidenhead, 1999.

[34] T. Hey and P. Walters. *The New Quantum Universe*. Cambridge University Press, 1st edition, 2003.

[35] P. Hines and P. Scott. The unitary trace on Hilbert spaces, and conditional quantum iteration. *To appear in Mathematical Structures in Computer Science*, 2007.

[36] M. Hirvensalo. *Quantum Computing*. Springer-Verlag New York, Inc., 2001.

[37] J. Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37:67–111, May 2000. http://www.cs.chalmers.se/∼rjmh/Papers/arrows.ps.

[38] J. A. Jones and M. Mosca. Implementation of a quantum algorithm on a nuclear magnetic resonance quantum computer. *Journal of Chemical Physics*, 109(5):1648–1653, 1998.

[39] R. Jozsa. Quantum algorithms and the Fourier Transform. In *Proceedings of the Santa Barbara Conference on Quantum Coherence and Decoherence*, 1997.

[40] J. Karczmarczuk. Structure and interpretation of quantum mechanics: a functional framework. In *Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 50–61. ACM Press, 2003.

[41] A. Kitaev. Quantum computations: algorithms and error correction. *Russian Mathematical Surveys*, 52(6):1191–1249, 1997.

[42] M. J. Klein. Maxwell, his demon, and second law of thermodynamics. *American Scienist*, 58(1):84–87, 1970.

[43] E. Knill. Conventions for quantum pseudocode, 1996. Technical Report LAUR-96-2724.

[44] R. Laflamme, D. Cory, C. Negrevergne, and L. Viola. NMR quantum information processing and entanglement. *Quantum Information and Computation*, 2(2):166–176, 2002.

[45] R. Laflamme, E. Knill, W.H. Zurek, P. Catasti, and S.V.S. Mariappan. NMR Greenberger - Horne - Zeilinger, 2006. http://arxiv.org/abs/quant-ph/9709025.

[46] R. Landauer. Irreversibility and heat generation in the computing process. *IBM Journal of Research and Development*, 5(3):183–191, 1961.

[47] S. Mac Lane. *Categories for the Working Mathematician*. Graduate Texts in Mathematics S. Springer-Verlag New York Inc., 1998.

[48] K. Mattle, H. Weinfurter, P. G. Kwiat, and A. Zeilinger. Dense coding in experimental quantum communication. *Physical Review Letters*, 76(25):4565–4659, 1996.

[49] C. McBride and J. McKinna. The view from the left. *Journal of Functional Programming*, 14(1), 2004.

[50] C. Monroe. Quantum information processing with atoms and photons. *Nature*, 416(6877):238–246, 2002.

[51] C. Monroe, D. M. Meekhof, B. E. King, W. M. Itano, and D. J. Wineland. Demonstration of a Fundamental Quantum Logic Gate. *Physical Review Letters*, 75(25):4714–4717, 1995.

[52] S-C. Mu and R. S. Bird. Quantum functional programming. In *2nd Asian Workshop on Programming Languages and Systems*, 2001.

[53] M. Nielsen and I. Chuang. *Quantum Computation and Quantum Information*. Cambridge University Press, Cambridge, 2000.

[54] B. Ömer. Procedural Quantum Programming. In *Proceedings of the AIP Conference on Computing Anticipatory Systems (AIP Conference Proceedings 627)*, pages 276–285. American Institute of Physics, 2001.

[55] S. Peyton-Jones and J. Hughes (editors). Haskell 98: A non-strict, purely functional language. Technical report, University of Glasgow, February 1999.

[56] B. C. Pierce. *Basic Category Theory for Computer Scientists*. Foundations of Computing. The MIT Press, 1991.

[57] A. Pittenger. *An Introduction to Quantum Computing Algorithms*, volume 19 of *Progress in Computer Science and Applied Logic (PCS)*. Berkhauser, 2000.

[58] J. Preskill. Course Notes - Computer Science 219: Quantum Computation, 1998. http://www.theory.caltech.edu/people/preskill/ph229/.

[59] R. Raussendorf and H. J. Briegel. A one-way quantum computer. *Phys. Rev. Lett.*, 86(22):5188–5191, May 2001.

[60] R. Raussendorf, D. E. Browne, and H. J. Briegel. The one-way quantum computer - a non-network model of quantum computation. *Journal of Modern Optics*, 49:1299, 2002.

[61] R. Raussendorf, D. E. Browne, and H. J. Briegel. Measurement-based quantum computation with cluster states. *Physical Review A*, 68:022312, 2003.

[62] K. F. Riley, M. P. Hobson, and S. J. Bence. *Mathematical Methods for Physics and Engineering*. Cambridge University Press, 1st edition, 1998.

[63] R. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2), 1978.

[64] A. Sabry. Modeling quantum computing in Haskell. In *Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 39–49. ACM Press, 2003.

[65] J. W. Sanders and P. Zuliani. Quantum Programming. In *Mathematics of Program Construction*, volume 1837 of *Lecture Notes in Computer Science*. Springer, 2000.

[66] T. Schenkel, A. Persaud, S. J. Park, J. Nilsson, J. Bokor, J. A. Liddle, R. Keller, D. H. Schneider, D. W. Cheng, and D. E. Humphries. Solid state quantum computer development in silicon with single ion implantation. *Journal of Applied Physics*, 94(11):7017–7024, 2003.

[67] E. Schrödinger. Die gegenwärtige Situation in der Quantenmechanik (The present situation in quantum mechanics). *Naturwissenschaften*, 23(48–50):807–812; 823–828; 844–849, 1935.

[68] P. Selinger, editor. *Proceedings of the 2nd International Workshop on Quantum Programming Languages*, number 33 in TUCS General Publications. Turku Centre for Computer Science, 2004.

[69] P. Selinger. Towards a Quantum Programming Language. *Mathematical Structures in Computer Science*, 14(4):527–586, 2004.

[70] P. Selinger. Towards a semantics for higher-order quantum computation. *Proceedings of the International Workshop on Quantum Programming Languages*, pages 127–143, 2004.

[71] P. Selinger, editor. *Proceedings of the 3rd International Workshop on Quantum Programming Languages*, Electronic Notes in Theoretical Computer Science. Elsevier Science, 2005.

[72] P. Selinger and B. Valiron. A lambda calculus for quantum computation with classical control. In *Proceedings of the 7th International Conference on Typed Lambda Calculi and Applications (TLCA)*,

volume 3461 of *Lecture Notes in Computer Science*. Springer, 2005. http://www.mathstat.dal.ca/∼selinger/papers/qlambda-full.pdf.gz.

[73] P Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings, 35th Annual Symposium on Foundations of Computer Science*. CA: IEEE Press, 1994.

[74] J. Silverman. *A friendly introduction to number theory*. Prentice Hall, 2006.

[75] A. M. Steane. Error correcting codes in quantum theory. *Physical Review Letters*, 77(5):793–797, 1996.

[76] A. M. Steane. The Ion Trap Quantum Information Processor. *Applied Physics B*, 64(6):623–642, 1997.

[77] W. Tittel, J. Brendel, Z. Binden, and N. Gisin. Violation of Bell inequalities by photons more than 10 km apart. *Physical Review Letters*, 81(17):3563–3566, 1998.

[78] T. Toffoli. Reversible Computing. Technical report, MIT/LCS/TM-151, February 1980.

[79] Q. A. Turchette, C. J. Hood, W. Lange, H. Mabuchi, and H. J. Kimble. Measurement of conditional phase shifts for quantum logic. *Physical Review Letters*, 75(25):4710–4713, 1995.

[80] A. van Tonder. Quantum computation, categorical semantics and linear logic, 2003. http://arxiv.org/abs/quant-ph/0312174.

[81] A. van Tonder. A lambda calculus for quantum computation. *SIAM Journal of Computing*, 33:1109–1135, 2004. http://arxiv.org/abs/quant-ph/0307150.

[82] J. Vizzotto, A. Rocha Costa, and A. Sabry. Quantum arrows in haskell. In Peter Selinger, editor, *Proceedings of the 4th International Workshop on Quantum Programming Languages*, Electronic Notes in Theoretical Computer Science. Elsevier Science, 2006.

[83] J. K. Vizzotto. *Structuring General and Complete Quantum Computations in Haskell: The Arrows Approach.* PhD thesis, Universidade Federal do Rio Grande do Sul, Brazil, 2006.

[84] J. K. Vizzotto, T.Altenkirch, and A. Sabry. Structuring Quantum Effects: Superoperators as Arrows, 2005. http://arxiv.org/abs/quant-ph/0501151.

[85] P. Walther, K. J. Resch, T. Rudolph, E. Schenck, H. Weinfurter, V. Vedral, M. Aspelmeyer, and A. Zeilinger. Experimental one-way quantum computing. *Nature*, 434:169–176, 2005.

[86] H. Xi. *Dependent Types in Practical Programming.* PhD thesis, Carnegie Mellon University, 1998.

Appendix A

Shor's algorithm and the Quantum Fourier Transform

This appendix will give an overview of one of the main theoretical applications of quantum computation: factoring large numbers efficiently. Shor's Algorithm will be presented, along with a mathematical description of how it works. This will include a discussion of the period-finding algorithm and the Quantum Fourier Transform. There is also a proof that the algorithm can find the prime factors of number using polynomially bounded resources with certainty.

An implementation of the classical portion of Shor's algorithm in Haskell is presented, and a QML implementation of a quantum Fourier transform is discussed.

A.1 Efficient factoring: Shor's algorithm

The publication of what is now know as Shor's algorithm [73] was responsible for a huge increase in interest in quantum computing, especially from cryptanalysts. This is because Shor's algorithm gives a method for finding the prime factors of a large composite integer, the classical intractability of which is the key to public-key cryptosystems such as RSA [63]. Although it is often presented as such, Shor's algorithm isn't actually a factoring algorithm; it is a quantum computer algorithm that computes the period of a real valued function. As such, it is also useful in other contexts.

A periodic function is one that repeats itself; *i.e.*

$$f(x) = f(x + mp) \tag{A.1}$$

where p is the period of the function, and mp is an integer multiple of the period, for all values of x in the domain of f. The reason that it is often presented as a factoring algorithm is that there is a well-known reduction of period finding to factoring, presented later in this section, and this use of the algorithm has generated the most interest (and funding).

Finding the period of any given periodic function, *i.e.* calculating p, is *hard* classically. The best known classical algorithms for finding the period of a periodic function takes a time that grows faster than any power of the number of bits representing p. However, Shor's algorithm can find the period in a time proportional to only a polynomial function of n, with an arbitrarily high probability of success.

A.2 From period finding to factoring

This is a presentation of a randomised reduction of the factoring problem to the period finding problem. This argument assumes there is an efficient algorithm for finding the period of a function. Also it makes use of the Euclidean algorithm, which efficiently computes the greatest common divisor (gcd) of two numbers, and the Chinese Remainder Theorem. The Euclidean algorithm is a well known example of a problem in the P (polynomial) complexity class, whose time complexity is bounded by a quadratic function of the length of the input values. The Euclidean algorithm can therefore be efficiently computed classically, and the algorithm can be found in any elementary number theory text, such as [74].

The goal of the reduction of period finding to factoring is to find the non-trivial factors of a number N, which can be represented by n bits. It will be assumed that N is the product of two distinct primes, p and q; this will later be shown to be the worst case. The reduction also assumes there is available an efficient way of computing the period of a function, namely Shor's algorithm. Throughout the reduction the

complexity (efficiency) of each step will be noted, since for the reduction to be useful it must be more efficient than classical factoring techniques.

The first step is to pseudo-randomly select a number $a < N$, assuming N is odd; otherwise 2 is a non-trivial factor. The greatest common divisor of a and N is then calculated, $\gcd(a, N)$. This calculation, as noted previously, can be computed efficiently by the Euclidean algorithm, with complexity $O(n^3)$.

If $\gcd(a, N) > 1$, then a is itself a non-trivial factor of N, and can be returned as the result. However, if a is *coprime* to N (shares no common factor, and therefore the $\gcd(a, N) = 1$), then further calculation is required.

It is important to note here that the numbers $a < N$ coprime to N, denoted $\mathbb{Z}_N^*$, form a finite group, with the group operation being multiplication $\mod N$. This follows as each element of $\mathbb{Z}_N^*$ has an inverse, as each element of $\mathbb{Z}_N^*$ is *distinct*, which itself follows from the fact that $N|x(a-b) \Rightarrow N|(a-b)$. Note that '$a|b$' reads '$a$ *divides* b'; *i.e.* $\exists c \in \mathbb{Z}.b = ac$

Each element x of this group must, by the definition of a group, have an *order* r, which is the least positive integer such that

$$x^r \equiv 1 \mod N, \tag{A.2}$$

It is shown below that the order of $a \mod N$ is also the period of the function

$$f(x) = a^x \mod N. \tag{A.3}$$

Recalling the definition of a periodic function (A.1), it can be shown that this is true by substituting p with r, and using function (A.3) above. This can then be rearranged to give (A.2).

An initial assumption was that there is an efficient way of finding the period of a function. Applying this method to (A.3) therefore also gives the order r of a, as the order of a is the period of this function. However, it is required that the function $f(x) = a^x \mod N$ can be computed efficiently. This is not at all obvious as x could be very large, requiring $O(x)$ expensive multiplication operations. Fortunately, the well known computer science technique of *repeated squaring* (modulo N) allows the

function to be calculated with a complexity of $O(\log x)$, or $O(m)$ where m is the number of bits required to represent x in binary form. This is much better than the $O(2^m)$ complexity of the näive method for large values of x.

Using the above and the assumption that the period of a function can be efficiently calculated, the order r of $a \mod N$ can be calculated. If r happens to be an odd number, then this method cannot find a non-trivial factor, and the algorithm must be restarted, this time selecting a different value of a from $\mathbb{Z}_N^*$. Suppose then that r is indeed even. In this case we since we know $a^r \equiv 1 \mod N$, then it is known that:

$$a^r - 1 \equiv 0 \mod N \tag{A.4}$$

and therefore that:

$$N|(a^r - 1). \tag{A.5}$$

As (by assumption) r is even, $a^r - 1$ can be factored to give

$$N|(a^{r/2} - 1)(a^{r/2} + 1) \tag{A.6}$$

It can immediately be seen that $N \nmid (a^{r/2} - 1)$, as this would imply that the order of a would be less than r. If it is also the case that $N \nmid (a^{r/2} + 1)$; *i.e.* , that

$$a^{r/2} \neq -1 \mod N$$

then N must share a non-trivial common factor with both $(a^{r/2} - 1)$ or $(a^{r/2} + 1)$. Therefore one of $\gcd(a^{r/2} \pm 1, N)$ is a non-trivial factor of N. As has already been shown, the greatest common divisor can be calculated efficiently using $O(n^3)$ operations.

Therefore, once r is calculated, N can be factored efficiently, assuming that r is even and that $a^{r/2} \neq -1 \mod N$. It now remains to calculate the probability of success and show that it is acceptable, and then to present a quantum algorithm for efficiently calculating the period of a function.

A.3 Probability Analysis

There are two cases in which the reduction given above will fail. The first is if the *order*, r, of the randomly selected number $a < N$ is odd. The second is if the value r is even, but that $a^{r/2} = -1 \mod N$. It will be shown in this section that the probability of either of these two cases occurring is *at most* $\frac{1}{2}$. Recall that $N = pq$, which was earlier asserted to be the worst case.

From the Chinese Remainder Theorem, explained in figure A.1, it is known that choosing a value a from $\mathbb{Z}_N$ is equivalent to choosing two values a_p and a_q, such that

$$a = a_p \mod p$$

$$a = a_q \mod q \tag{A.7}$$

Also, this pair is unique, and has the same statistical distribution as choosing two numbers randomly from $\mathbb{Z}_p$ and $\mathbb{Z}_q$. Each of these two new values, a_p and a_q also have an associated *order*, r_p and r_q, in the groups $\mathbb{Z}_p$ and $\mathbb{Z}_q$, respectively.

It will now be shown that for each possible choice of a, from $\mathbb{Z}_N$, the order r of a is the least common multiple (lcm) of the orders r_p and r_q of a_p and a_q. This is because, from the definitions of order and N,

$$a^r \equiv 1 \mod N \equiv 1 \mod pq \tag{A.8}$$

which implies that $a^r \equiv 1 \mod p$, and $a^r \equiv 1 \mod q$. Therefore r must be an integer multiple of r_p and r_q. It can further be shown that $a^x \equiv 1 \mod pq$ where x is any integer multiple of r_p and r_q. Therefore, because r is (by definition) the *least* integer with the property that $a^r \equiv 1 \mod pq$, and since r must be common (integer) multiple of r_p and r_q, then r must be the least common multiple of r_p and r_q. This result tells us that r can only be odd if both r_p and r_q are odd, and even if either one of r_p or r_q is even. Note that r_p and r_q cannot both be even, as this would imply that $a^{r/2} \equiv 1 \mod pq$, which contradicts the definition of r as the order of a.

If r is even then the reduction will only fail if $a^{r/2} \equiv -1 \mod pq$. By application of the CRT it can be seen that this can only be the case if both $a^{r/2} \equiv -1 \mod p$ and

The Chinese Remainder Theorem

The Chinese Remainder Theorem (CRT) is used frequently in this discussion, so it is explained here with a concise proof. The CRT essentially states that working in modulo N, where $N = pq$, is the same as working modulo p and modulo q simultaneously, where p and q are distinct primes.

Conjecture

Let $p \neq q$ be two primes, and let $N = pq$. For every $a \in \mathbb{Z}_p$, $b \in \mathbb{Z}_q$, there is a unique c such that $0 \leqslant c \leqslant N$, and $c \equiv a \mod p$ and $c \equiv b \mod q$.

Proof

Let $u = p^{-1} \mod q$ and $v = q^{-1} \mod p$, and let $c' = upb + vqa$. Then

$$c' \equiv upb + vqa \equiv u \cdot 0 \cdot b + 1 \cdot a \equiv a \mod p$$

and equivalently

$$c' \equiv upb + vqa \equiv 1 \cdot b + v \cdot 0 \cdot a \equiv b \mod q$$

Let $c = c' \mod N$. Then $N|(c-c')$, which means $p|(c-c')$ so $c \equiv c' \mod p$. Similarly, $c \equiv c' \mod q$. Therefore $c \equiv c' \equiv a \mod p$ and $c \equiv c' \equiv b \mod q$.
Hence c satisfies all the conditions of the conjecture; $0 \leqslant c \leqslant N$, and $c \equiv a \mod p$, and $c \equiv b \mod q$.
This shows that for every pair a and b, as defined, there exists a exactly one value of c; c is *unique*.

□

FIGURE A.1: Definition and proof of the Chinese Remainder Theorem

$a^{r/2} \equiv -1 \mod q$. Now, suppose 2^{c_p} and 2^{c_q} are the largest powers of 2 that divide r_p and r_q, respectively, and also that 2^c is the largest power of 2 that divides r. It will now be shown that to have either r as odd valued, or to have r even *and* $x^{r/2} \equiv -1 \mod pq$, it is necessary that $c_p = c_q$. It will be then shown that the probability $c_p = c_q$ for a randomly selected value of a is $\frac{1}{2}$.

As has already been stated, for r to be odd, r_p and r_q must both be odd. This follows from the dual facts that $r_p|r$, and $r_q|r$. Therefore, if r is odd then the values c_p and c_q must both be 0, and are therefore equal.

In the case where either one of r_p or r_q is even, then it must follow that $a^{r/2} \neq -1 \mod pq$, unless r_p and r_q are both odd multiples of the same power of 2; $c_p = c_q$. This is because, if $c_p > c_q$ then, as r is an integer multiple of r_p and r_q, then r must equal $2 \cdot r_q \cdot x$, where x is some integer. It must then follow that $a^{r/2} \equiv 1 \mod q$, which means the algorithm must succeed as r is even and, as was stated previously, $a^{r/2}$ cannot in this case be equivalent to $-1 \mod pq$ (by the CRT). By a similar chain of reasoning it can be seen that $c_p < c_q$ implies $a^{r/2} \equiv 1 \mod p$; which also rules out $a^{r/2} \equiv -1 \mod pq$ for the same reason.

The analysis in the preceding paragraphs shows that the reduction will only fail if $c_p = c_q$. It now remains to calculate the upper bound of the probability of this happening for a random value a. This result follows from the fact that for any given prime p, a randomly selected element from the group $\mathbb{Z}_p$ has a probability of exactly $\frac{1}{2}$ that it will be an odd multiple of any power of 2. The proof of this is presented below.

First, note that $\varphi(p) = p - 1$ is even, as p is odd, and therefore $c_p \geqslant 1$ (recall $2^{c_p} = r_p$). $\varphi(p)$ is the Euler Phi Function, or Totient function, defined as the number of positive integers $\leqslant p$ that are coprime to p. An elementary result from number theory states that the group $\mathbb{Z}_p^*$ is cyclic, and therefore there exists a generator g for $\mathbb{Z}_p^*$, with an order equal to $\varphi(p) = (p-1)$. This means that any element from $\mathbb{Z}_p^*$ may be written in the form $g^k \mod p$, for some k such that $1 \leqslant k \leqslant (p-1)$. Now let r_g be the order of $g^k \mod p$ and consider the two following cases:

Firstly, if k is odd then from the fact that $g^{kr_g} \equiv 1 \mod p$ it can be deduced that kr_g is a multiple of $\varphi(p)$. Therefore $2^{c_p}|r_g$, since k is odd, and r_g has the same

number of powers of 2 as does the order of g, $\varphi(p)$.

Secondly, if k is even, then

$$g^{k\varphi(p)/2} = (g^{\varphi(p)})^{k/2} = 1^{k/2} = 1 \mod p \tag{A.9}$$

thus r_g must divide $\varphi(p)/2$, and therefore r_g must contain at least one less power of 2 than $\varphi(p)$ does.

To summarise, the group $\mathbb{Z}_p^*$ can be split into two sets of equal size: those that may be written g^k with k odd, for which $2^{c_p} | r_g$; and those written g^k with k even, for which $2^{c_p} \nmid r_g$. Therefore, with a probability of exactly $\frac{1}{2}$, the integer 2^{c_p} divides the order r_g of any random element from $\mathbb{Z}_p^*$. Given this, the probability of both a_p and a_q having orders r_p and r_q, that are an odd multiple of the same power of 2, *i.e.* $c_p = c_q$, is *at most* $\frac{1}{2}$. Note these are independent events.

Recall the earlier assertions that N being the product of two distinct primes was the worst case. If N is the product of n primes, then

$$P(success) \geqslant 1 - \frac{1}{2^{n-1}} \tag{A.10}$$

The probability of failure decreases by a factor of $1/2$ for every extra prime making up N, thus the probability of success in factoring N increases. This can seen by generalising the discussion above to n primes (given fully in [53]), and finding that in order for the algorithm to fail now, it must be the case that $c_{p_1} = c_{p_2} = \ldots = c_{p_n}$, where $2^{c_{p_x}}$ is the largest power of 2 which divides the order r_{p_x} of a_{p_x}. Thus, following the same reasoning as for just two primes, the probability of this occurring is $\frac{1}{2} \cdot \frac{1}{2} \cdot \ldots \cdot \frac{1}{2}$, a total of $n-1$ times;

$$P(fail) \leqslant \frac{1}{2^{n-1}} \tag{A.11}$$

In conclusion, the probability of the reduction succeeding, in the worst case, is *at least* half; $P(\text{success}) \geqslant \frac{1}{2}$. This is a constant value, not relying on the size of N, and this means that the algorithm has to be repeated only a small number of times to get a high probability of success. Therefore, the algorithm is efficient.

A.4 Haskell implementation of Shor's algorithm

Presented in this section are some Haskell functions that perform the classical parts of the reduction algorithm given in section A.1 in order to factor a number N. Haskell is a pure functional programming language that is well suited to tasks of this kind. Before implementing a function to perform the reduction, it helps to define auxiliary functions that may be used by other functions. Firstly, a function is required that, when given two integers, a and n, returns the order of $a \mod n$. This function will be inefficient as it will be computed classically; however it will be shown in section A.5 that this computation can be done efficiently by a quantum computer.

$$
\begin{aligned}
&\mathit{order} \quad \in \mathit{Int} \to \mathit{Int} \to \mathit{Int}\\
&\mathit{order}\ a\ n = \mathit{find}\ a\\
&\quad \textbf{where}\ \mathit{find}\ b = \textbf{if}\ \mathit{mod}\ b\ n \equiv 1\\
&\qquad\qquad\qquad\qquad \textbf{then}\ 1\\
&\qquad\qquad\qquad\qquad \textbf{else}\ 1 + (\mathit{find}\ (\mathit{mod}\ (a \times b)\ n))
\end{aligned}
$$

The algorithm used by the *order* function is a very inefficient implementation, but serves as an understandable definition. More efficient classical algorithms, though less efficient than that presented for quantum computers, can be used, such as the Baby step – Giant step technique in number theory.

An efficient way of calculating the function $f(x) = a^x \mod N$ is also required. This is implemented using the method of repeated squaring modulo N, and is based on the built-in Haskell library function $a \uparrow x = a^x$. This function returns $a^x \mod n$.

$$
\begin{aligned}
&\mathit{pow}\ a\ 0\ n \qquad\qquad = 1\\
&\mathit{pow}\ a\ x\ n \mid x > 0 = f\ a\ (x-1)\ a\\
&\quad \textbf{where}\ f\ \_\ 0\ y = y\\
&\qquad\qquad f\ a\ x\ y = g\ a\ x\ \textbf{where}\\
&\qquad\qquad g\ a\ x \mid \mathit{even}\ x \quad = g\ (\mathit{mod}\ (a \times a)\ n)\ (x\ `\mathit{quot}`\ 2)\\
&\qquad\qquad\qquad\quad \mid \mathit{otherwise} = f\ x\ (x-1)\ (\mathit{mod}\ (a \times y)\ n)\\
&\mathit{pow}\ \_\ \_\ \_ \mid \mathit{error}\ \texttt{"pow: negative exponent"}
\end{aligned}
$$

A function to efficiently perform factorisation can now be defined; excepting that calculating $\mathit{order}\ a\ n$ is inefficient in this instance. This function makes use of the

built-in Haskell function for calculating the greatest common divisor, which is known to be efficient. The first step in implementing the function to perform the reduction algorithm is to define a new data type which covers all possible outputs:

$$\textbf{data}\ \mathit{Shor} = \mathit{FailOdd} \mid \mathit{FailEvenMod} \mid \mathit{Trivial}\ \mathit{Int} \mid \mathit{Factor}\ \mathit{Int}$$

A data type is a set of values from which a function may take its value. This line of Haskell code defines a new data type called *Shor*, which can take one of four possible values. Each is separated by a vertical bar, which can be read as *or*.

Using this new data type a function can now be defined that returns a value of type Shor, which is the result of the reduction algorithm.

$$\begin{aligned}
&\mathit{shor} \in \mathit{Int} \to \mathit{Int} \to \mathit{Shor} \\
&\mathit{shor}\ a\ n \mid \mathit{even}\ n \qquad\quad = \mathit{Trivial}\ 2 \\
&\qquad\quad \mid \mathit{triv} \not\equiv 1 \qquad = \mathit{Trivial}\ \mathit{triv} \\
&\qquad\quad \mid \mathit{odd}\ r \qquad\quad = \mathit{FailOdd} \\
&\qquad\quad \mid \mathit{ar} \equiv (n-1) = \mathit{FailEvenMod} \\
&\qquad\quad \mid \mathit{otherwise} \qquad = \mathit{Factor}\ (\mathit{gcd}\ n\ (\mathit{ar}+1)) \\
&\quad \textbf{where}\ \mathit{triv} = \mathit{gcd}\ a\ n \\
&\qquad\quad\ \ r \quad = \mathit{order}\ a\ n \\
&\qquad\quad\ \ \mathit{ar} \quad = \mathit{pow}\ a\ (r\ `\mathit{div}`\ 2)\ n
\end{aligned}$$

The first line of this function gives its name and *type signature*. This function takes two integers, and returns one of the *Shor* data types. The second line is the start of the actual algorithm, and the inputs to this function, a and n, represent the values a and N in the discussion above – n is the number to be factored, and $a < n$. This function treats the reduction algorithm as an analysis of five cases.

The first case simply checks whether the number n to be factored is even. This can be computed efficiently by reading the least significant bit of the numbers binary expansion. If the number is even then the *Shor* data type *Trivial* is returned with 2 as a non-trivial factor of n. If n is odd then the next case is examined, which asks if $\mathit{triv} \neq 1$, where *triv* is an auxiliary function that is defined as the greatest common divisor (gcd) of a and n. If this is the case then the function returns *Trivial* along with the value of *triv*, the gcd, as a non-trivial factor of n. The value returned in this second case is called *Trivial* not because the factor is trivial (it is by definition

non-trivial), but because the non-trivial factor can be computed without recourse to the period finding algorithm; what is meant is that it is computationally trivial.

The next case tests a value r, which is defined as the order of a with respect to n, to determine if it is odd. If it is odd, then a factor cannot be found using this method. The *Shor* value *FailOdd* is therefore returned with no factor. It is important to note that the calculation of r is the only part of this algorithm that cannot be done efficiently on a classical computer; however, the reduction given previously explains how this can be done efficiently using the quantum computer order-finding algorithm, presented in section A.5.

The fourth case also results in a failure if the Boolean expression evaluates to true. This case examines the value ar, which is defined as $a^{r/2} \mod n$, and returns *FailEvenMod* if $ar = n - 1$. Recall that the reduction will only work if $a^{r/2} \neq n - 1 \mod n$. This calculation is done efficiently using the *pow* function.

The final case, if reached, will return the greatest common divisor of a and $a^{r/2} - 1$, which was shown in the reduction to be a non-trivial factor of a and n. This value is then returned with the *Shor* data type *Factor*. This uses only the Euclidean algorithm again, and ar which has already been calculated, and therefore this step can be calculated efficiently.

Note that the value $a < n$ input to this function should be randomly selected; however, a simple additional function, *shor′*, will calculate the output of the *shor* function for all possible values of a. This function makes use of the Haskell list comprehension syntax, and returns a list of pairs. The first element in each pair is the value of a used, and the second is the returned *Shor* value.

$$shor' \quad \in Int \to [(Int, Shor)]$$
$$shor'\ n = [(x, shor\ x\ n) \mid x \leftarrow [2\,..\,(n-1)]]$$

The Haskell function *shor* presented here will succeed to factor N given a with a probability of at least one half, as shown in section A.3. The function is efficient, except for the order-finding auxiliary function *order*. The next section, A.5, explains how a quantum computer can perform the order-finding function efficiently.

A.5 Order-finding by phase estimation

There is no known algorithm, at the current time, that can solve the order-finding problem using resources polynomial to the number of bits required to specify it. It is believed to be computationally hard, using classical methods. The quantum order-finding algorithm can, however, find the order of an element in a group using polynomially bounded resources. Recall that the goal of order-finding is to find the least value r such that

$$a^r \equiv 1 \mod N$$

for $a \in \mathbb{Z}_N^*$, *i.e.* $a \leqslant N$ with a coprime to N, as previously. The quantum order-finding algorithm can complete this task with a complexity of only $O(n^3)$, where n is the number of bits in the binary expansion of N.

The algorithm used to perform order-finding efficiently on a quantum computer is actually a specific example of a more general quantum algorithm, called *Phase Estimation*. In turn, the phase estimation technique makes use of the quantum equivalent of the *Fourier transform*. Both of these techniques, and how they are used to solve the order finding problem will be described here.

A.5.1 The phase estimation algorithm

Given a unitary operator U, with some eigenvector $|u\rangle$ and associated eigenvalue $\lambda = e^{2\pi i\varphi}$, the phase estimation algorithm, due to Kitaev [41], returns an estimate of φ, and therefore also estimates the eigenvalue λ of U. In the case of the order-finding algorithm the unitary operator is

$$U_a|x\rangle = |ax \mod N\rangle \tag{A.12}$$

where $x \in \{0,1\}^n$, *i.e.* takes values in the range 0 to $N-1$. Note that if $x \geqslant N$ then U_a returns x; the operator only acts non-trivially when $x \leqslant N$. This operation is unitary because a is coprime to N, and therefore multiplication by $a \mod N$ is invertible.

Following from the fact that r is the order of $a \mod N$, multiplying the operator

U_a by itself r times will result in the identity matrix:

$$U_a^r \equiv \mathbf{1} \tag{A.13}$$

From this it follows that the eigenvalues of U_a are r^{th} roots of unity:

$$\lambda_s = e^{2\pi i(s/r)} \tag{A.14}$$

where $s \in \{0, 1, \cdots, r-1\}$. This is the same as the formula for λ given above, except with the phase, φ, substituted by s/r. The corresponding eigenvectors are:

$$|u_s\rangle = \frac{1}{\sqrt{r}} \sum_{k=0}^{r-1} e^{2\pi ik(s/r)} |a^k \mod N\rangle \tag{A.15}$$

as:

$$U_a|u_s\rangle = \frac{1}{\sqrt{r}} \sum_{k=0}^{r-1} e^{2\pi ik(s/r)} |a^{k+1} \mod N\rangle = e^{2\pi i(s/r)}|u_s\rangle$$

Applying the phase estimation procedure to U_a should give an estimate of the phase, $\varphi = s/r$. From this it is then possible to obtain the order r itself, completing the algorithm.

The phase estimation procedure makes use of two registers of quantum bits. The first registers qubits are all initially in the state $|0\rangle$, and the second register initially set to be in one of the eigenvectors of U, $|u_s\rangle$. The second register need only contain enough qubits to represent $|u_s\rangle$, but the number of qubits in the first register is dependant on two factors: the accuracy of the estimate of s/r required, and the probability of the algorithm being successful. These factors will be discussed later in this section.

Before discussing the phase estimation procedure proper, it may be instructional to look at a simpler algorithm. Suppose the registers are prepared as above, with the second register containing $|u_s\rangle$, and with the first containing only one qubit in the state $|0\rangle$. Suppose this is the input into the circuit given in Figure A.2. The quantum circuit metaphor is fully explained in section 3.3, but a detailed understanding is not required for this analysis. The action of each operation in this circuit on the input will now be explained. Only the first register will be considered, as from this it is easy to see that the second register remains unchanged throughout the computation.

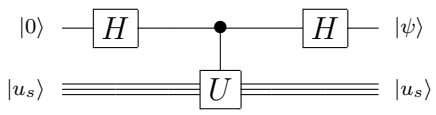


FIGURE A.2: Example phase estimation circuit

Firstly, the application of the initial Hadamard transform has the following action:

$$|\psi_1\rangle = \frac{|0\rangle + |1\rangle}{\sqrt{2}} \tag{A.16}$$

placing the control qubit into a superposition. Next, the application of the controlled-U gate produces the state:

$$|\psi_2\rangle = \frac{|0\rangle + U_a|1\rangle}{\sqrt{2}} = \frac{|0\rangle + \lambda_s|1\rangle}{\sqrt{2}} \tag{A.17}$$

as $U_a|u_s\rangle = \lambda_s|u_s\rangle$. The action of the final Hadamard transform then causes the state to become:

$$|\psi_3\rangle = \frac{|0\rangle + |1\rangle}{2} + \frac{\lambda_s|0\rangle - \lambda_s|1\rangle}{2} = \frac{(1+\lambda_s)|0\rangle}{2} + \frac{(1-\lambda_s)|1\rangle}{2} \tag{A.18}$$

If the first register is now measured the outcome would have the following probability distribution:

$$P(0) = |\frac{1}{2}(1+\lambda_s)|^2$$

$$P(1) = |\frac{1}{2}(1-\lambda_s)|^2$$

Preskill [58] notes that this distribution can be rewritten as:

$$P(0) = \cos^2(\pi\varphi)$$

$$P(1) = \sin^2(\pi\varphi)$$

because $\lambda_s = e^{2\pi i\varphi}$, which allows two eigenvalues to be distinguished with absolute certainty: if $\varphi = 0$ then $\lambda_s = 1$, and if $\varphi = 1/2$ then $\lambda_s = -1$. Unfortunately, other values of λ_s cannot be extracted with such a high probability. In order to obtain

a sufficiently accurate estimate of the phase an exponential number of controlled-U operations, such as that above, would be required.

However, if computing large powers of U, such as U^{2^x}, could be achieved efficiently, then applying the circuit given in figure A.2 would allow

$$e^{2\pi i 2^x \varphi} \tag{A.19}$$

to be determined. This circuit therefore allows the x^{th} bit of λ_s to be calculated.

Extending this circuit for a larger number of qubits in the first register, discarding the final Hadamard transform, gives us the circuit shown in figure A.3. This circuit

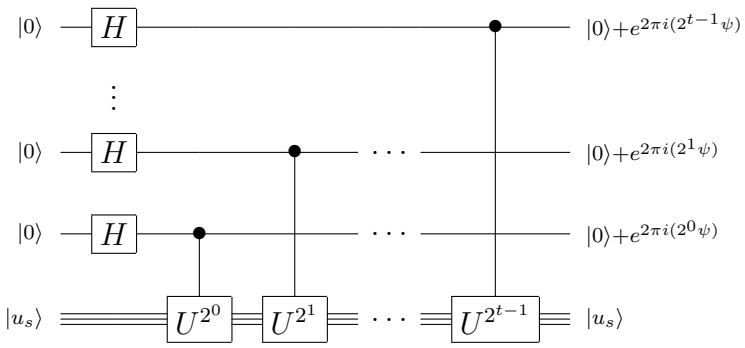


FIGURE A.3: First stage of phase estimation, where $|u_s\rangle$ denotes the second register. Normalisation factors are missing from output states.

gives the first stage of the phase estimation algorithm. It starts by applying the Hadamard transform to the first register, and then applying controlled-U_a operations to the second register; each raised to a successive power of 2, and controlled by a different qubit from the first register. The state of the second register remains unchanged throughout, while the state of the first register becomes:

$$\begin{aligned} &\frac{1}{\sqrt{2^t}}\left(|0\rangle + e^{2\pi i 2^{t-1}\varphi}|1\rangle\right) \otimes \left(|0\rangle + e^{2\pi i 2^{t-2}\varphi}|1\rangle\right) \\ &\qquad \otimes \cdots \otimes \left(|0\rangle + e^{2\pi i 2^0 \varphi}|1\rangle\right) \\ &= \frac{1}{\sqrt{2^t}} \sum_{k=0}^{2^t-1} e^{2\pi i \varphi k}|k\rangle \end{aligned} \tag{A.20}$$

where t is the number of qubits in the first register.

Suppose that the phase, φ, in the formula above can be represented by the following binary fraction *exactly*:

$$\varphi = .\varphi_0\varphi_1\cdots\varphi_{t-1} = \frac{\varphi_0}{2^1} + \frac{\varphi_1}{2^2} + \cdots + \frac{\varphi_{t-1}}{2^t}$$

where t is the number of bits in the first register, and $\varphi_x \in \{0, 1\}$. This notation can be used to rewrite equation A.5.1 above, the output from this first stage of the phase estimation algorithm, as:

$$\frac{1}{\sqrt{2^t}}\left(|0\rangle + e^{2\pi i.\varphi_{t-1}}|1\rangle\right) \otimes \left(|0\rangle + e^{2\pi i.\varphi_{t-2}\varphi_{t-1}}|1\rangle\right)$$

$$\otimes\cdots\otimes \left(|0\rangle + e^{2\pi i.\varphi_0\cdots\varphi_{t-2}\varphi_{t-1}}|1\rangle\right) \tag{A.21}$$

The second and final stage of the phase estimation algorithm is now to apply the *inverse* quantum Fourier transform (QFT) to the first register. After performing this operation measuring the first register will result in the state $|.\varphi_0\cdots\varphi_{t-2}\varphi_{t-1}\rangle$ – from which $\varphi = s/r$ can now be calculated. It is now a simple task to extract the order, r. How the QFT performs this operation will be explained in section A.5.2.

In order to execute the phase estimation procedure two prerequisites must be met. Firstly, it must be possible to efficiently compute the controlled-U^{2^x} operations, for any integer value x. Secondly, the phase estimation procedure requires that an eigenvector of U be placed in the second register, and it cannot be prepared using the formula for $|u_s\rangle$ given previously, as this requires knowledge of the order, r.

The first prerequisite can be met using a method called *modular exponentiation*, which is similar to the *repeated squaring* algorithm for calculating integer powers. A full explanation is given is given on page 228 of [53]. The procedure works as follows: First, calculate $U^2|x\rangle = |x^2 \mod N\rangle$ by applying U twice. $U^4|x\rangle$ can then calculated by applying U^2 twice, and then continuing in this way until the required power of U is reached. This results in a complexity of $O(n^3)$, where n is the number of bits used to represent N.

Unfortunately, preparing an eigenvector of U without knowledge of r is more difficult, if not impossible. However, a simple observation allows this problem to be

avoided. Recall the definition of the eigenvector (equation A.15):

$$|u_s\rangle = \frac{1}{\sqrt{r}} \sum_{k=0}^{r-1} e^{2\pi i k(s/r)} |a^k \mod N\rangle$$

From this it can be shown that

$$\frac{1}{\sqrt{r}} \sum_{s=0}^{r-1} |u_s\rangle = |1\rangle \tag{A.22}$$

which means $|1\rangle$ is an equally weighted superposition of the r eigenvectors of U_a.

So in order to perform the phase estimation algorithm the second register needs only to be initialised to the state $|1\rangle$, which is trivial to construct. If $r < 2^n$ then setting the size of the first register, t, to $2n+1$ allows an estimate of s/r to be obtained that is accurate to $2n+1$ bits.

The final step in calculating the order is to now extract a good estimate for r from this information, and from the fact that s and r are integers. From this information it follows that φ is itself a rational number; the ratio of two bounded integers. It is now possible to use the *continued fractions* [74] method in order to calculate the nearest such fraction to φ; in which r will be the denominator. To paraphrase, the continued fractions algorithm can produce a pair s' and r' efficiently, that have no common factors such that $s'/r' = s/r$. All that remains is to confirm that r' is indeed the order by calculating $a^{r'} \mod N$, obtaining the result 1.

There are now two possible scenarios to consider which cause the order-finding algorithm to fail. The first is if the phase estimation section results in a bad estimate of the phase, and thus s/r. The chances of this occurring can be reduced greatly by negligible increases in the size of the first register, as shown in [53], and is already very small if $t = 2n + 1$. Reducing the number of qubits only increases the chance of failure logarithmically, above $2n + 1$.

The second scenario is that if the pair s and r share a common factor, then the value of r' returned by the continued fraction algorithm may be a factor of r, and not r itself. It should be noted that the probability of s and r not sharing any common factors is significant as when s is selected randomly from $\mathbb{Z}_{r-1}$; the probability of s being coprime to r is in fact $\phi(r)/r$. However, only repeating the order-finding

algorithm a constant number of times allows us to extract r with certainty. To calculate r with certainty first requires performing the order-finding procedure twice, giving two estimates of s and r; s'_1,r'_1 and s'_2,r'_2 respectively. If s'_1 and s'_2 are coprime then r can be calculated by finding the least common multiple of r'_1 and r'_2. Otherwise, perform the order-finding algorithm until two coprime values of s' are obtained. The probability that s'_1 is coprime to s'_2 is given by

$$1 - \sum_p P(p|s'_1)P(p|s'_2) \tag{A.23}$$

where $P(p|s)$ is the probability that p divides s, and the sum is over all prime numbers less than r. Note that all possible values of s,s' are bounded by r. Now, if $p|s'_1$ then it is also the case that p divides the actual value s_1 $(= s)$, so finding an upper bound for $P(p|s_1)$ suffices. The probability of $p|s_1$ for a random $s_1 < r$ is easily seen to be at most p^{-1}, and therefore $P(p|s'_1) \leqslant p^{-1}$. The same argument can be applied to show $P(p|s'_2) \leqslant p^{-1}$, and the probability that s'_1 and s'_2 are coprime can now be written as

$$1 - \sum_p P(p|s'_1)P(p|s'_2) \leqslant 1 - \sum_p p^{-2} \tag{A.24}$$

The right-hand side of equation A.24 can be bounded by noting that

$$\sum_p p^{-2} \leqslant \frac{3}{2}\int_2^\infty y^{-2}dy = \frac{3}{4} \tag{A.25}$$

thus giving:

$$1 - \sum_p P(p|s'_1)P(p|s'_2) \leqslant \frac{1}{4} \tag{A.26}$$

Therefore, the probability of success is at least 1/4.

Summarising, the order-finding algorithm can calculate the order r with a small constant number of repetitions. It makes use of the phase estimation technique with the quantum Fourier transform to estimate s/r, and then uses the continued fractions algorithm to extract r. The phase estimation algorithm uses $O(n)$ Hadamard transforms, followed by $O(n^3)$ transforms to calculate the controlled unitary transforms, giving a total cost of $O(n^3)$. The inverse quantum Fourier transform requires a further $O(n^2)$ gates, while the continued fraction algorithm consumes $O(n^3)$ operations

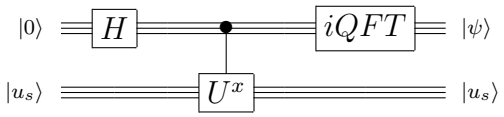


FIGURE A.4: Schematic representation of order-finding algorithm. The top two wires represent the first register of t qubits, and $iQFT$ is the *inverse* quantum Fourier transform

to give an overall complexity $O(n^3)$. Lastly, to ensure the output from the continued fractions algorithm is the order the order-finding algorithm must be repeated a constant number of times, for a total final cost of $O(n^3)$. A schematic representation of the order-finding algorithm, using phase estimation, is shown in figure A.4.

A.5.2 The Quantum Fourier Transform

The Fourier transform is a very useful tool that is used a lot in maths, physics and computer science. It provides a representation of functions defined over some interval, with possibly no particular periodicity, in terms of a superposition of sinusoidal functions. It can be viewed as a generalisation of the Fourier series representation of periodic functions.

The classical *discreet Fourier transform* (DFT) take as its input a vector of complex numbers $|x\rangle = x_0, x_1, \cdots, x_{N-1}$, where N is the size of the vector, and returns the transformed vector $|y\rangle = y_0, y_1, \cdots, y_{N-1}$. The action of the DFT can be given as:

$$\sum_x f(x)|x\rangle \rightarrow \sum_y \left(\tfrac{1}{\sqrt{N}} e^{2\pi i xy/N} f(x)\right)|y\rangle \tag{A.27}$$

This can be considered to be a definition of an $N \times N$ unitary matrix, where each element m indexed by x and y is given by:

$$m_{xy} = \left(e^{2\pi i/N}\right)^{xy}$$

A naive implementation of this transform would require $O(2^{n^2})$ operations; however this can be simplified to $O(n2^n)$ without recourse to a quantum computer, using the *Fast Fourier Transform* (FFT).

The *Quantum Fourier Transform* (QFT) has exactly the same action, except that the input is a quantum state, and it makes use of quantum parallelism to give an even more efficient computation than the FFT. The QFT is a linear operation that has the following action on the computational basis state:

$$QFT : |x\rangle \rightarrow \frac{1}{\sqrt{N}} \sum_{y} e^{2\pi i x y/N} |y\rangle \tag{A.28}$$

Assuming that $N = 2^n$, $|x\rangle$ and $|y\rangle$ can be expressed as the following binary expansions:

$$|y\rangle = y_0 2^{n-1} + y_1 2^{n-2} + \cdots + y_{n-1} 2^0$$

$$|x\rangle = x_0 2^{n-1} + x_1 2^{n-2} + \cdots + x_{n-1} 2^0 \tag{A.29}$$

Using this binary expansion notation a more useful version of the definition of the QFT (equation A.28) can be derived, as shown in [53], called the product representation:

$$\begin{aligned}
|x\rangle \;\rightarrow\; & \frac{1}{\sqrt{2^n}} \sum_{y=0}^{2^{n-1}} e^{2\pi i x y/2^n} |y\rangle && \text{(A.30)}\\
= \; & \frac{1}{\sqrt{2^n}} \sum_{y_0=0}^{1} \cdots \sum_{y_{n-1}=0}^{1} e^{2\pi i x (\sum_{j=1}^{n} y_j 2^{-j})} |y_0 \cdots y_{n-1}\rangle && \text{(A.31)}\\
= \; & \frac{1}{\sqrt{2^n}} \sum_{y_0=0}^{1} \cdots \sum_{y_{n-1}=0}^{1} \bigotimes_{j=0}^{n-1} e^{2\pi i x y_j 2^{-j}} |y_j\rangle && \text{(A.32)}\\
= \; & \frac{1}{\sqrt{2^n}} \bigotimes_{j=0}^{n-1} \left(\textstyle\sum_{y_j=0}^{1} e^{2\pi i x y_j 2^{-j}} |y_j\rangle \right) && \text{(A.33)}\\
= \; & \frac{1}{\sqrt{2^n}} \bigotimes_{j=0}^{n-1} \left(|0\rangle + 2^{2\pi i x y 2^{-j}} |1\rangle \right) && \text{(A.34)}\\
= \; & \frac{1}{\sqrt{2^n}} \left(|0\rangle + e^{2\pi i (.x_{n-1})} |1\rangle \right) \otimes \left(|0\rangle + e^{2\pi i (.x_{n-2} x_{n-1})} |1\rangle \right) \otimes \\
& \cdots \otimes \left(|0\rangle + e^{2\pi i (.x_0 \cdots x_{n-2} x_{n-1})} |1\rangle \right) && \text{(A.35)}
\end{aligned}$$

This product representation has two useful properties: firstly, it makes it straightforward to derive an efficient circuit to perform the procedure; and secondly, it shows how the second stage of the phase estimation algorithm works. By comparing the product representation given in equation A.35 with the output from the initial phase of the order-finding algorithm, given in equation A.5.1, shows that performing the inverse of the quantum Fourier transform on this state would produce the product state $|.\varphi_0\varphi_1\cdots\varphi_{t-1}\rangle$, thus completing the phase estimation part of the algorithm. If there is a circuit that performs the QFT, then the inverse QFT can simply be performed by reversing the order of the operations. Figure A.5 shows a circuit that performs the QFT efficiently, given a rotation R_d defined as:

$$R_d = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\pi/2^d} \end{pmatrix} \tag{A.36}$$

where d is the 'distance' between the R gate and its control qubit; *i.e.* d is the number of wires separating R from the corresponding control wire.

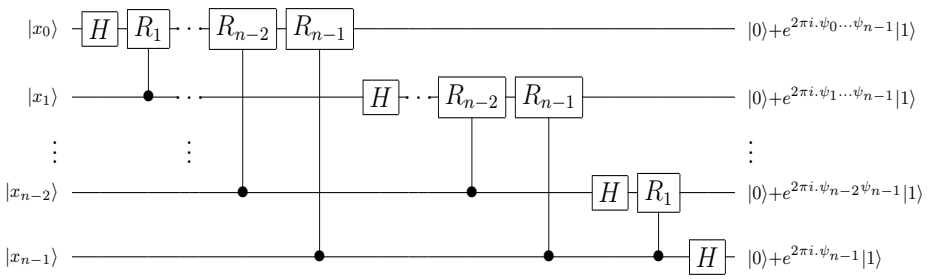


FIGURE A.5: An efficient circuit for the quantum Fourier transform. Normalisation factors of $1/\sqrt{2}$ are missing from output. Note the order of qubits has been reversed in the output

From the circuit given for the QFT, the complexity can be calculated to be $O(n^2)$, and a full analysis of this is provided on page 219 of [53]. This initially seems to be much better than the complexity of the DFT or FFT, and naïvely one would assume that using the QFT instead of these would incur large efficiency gains. Unfortunately, the output from the QFT is a quantum superposition, and measuring it would collapse the superposition – erasing all values of the Fourier transform except for one,

determined at random. Additionally, there is no general algorithm for efficiently preparing the state to be input into the QFT. This makes finding uses for the QFT much harder, but the phase estimation algorithm, and specifically the order-finding algorithm, is an excellent example of an algorithm that outperforms classical computations by making use of the quantum Fourier transform. In addition, Jozsa [39], has shown how all the main quantum algorithms rely to some extent on the quantum Fourier transform in order to work efficiently.

A.6 Implementing the three qubit Quantum Fourier Transform

The three qubit quantum Fourier transform can be implemented as the circuit shown in figure A.6, using the quantum circuits defined in section 3.3. The rotations S and

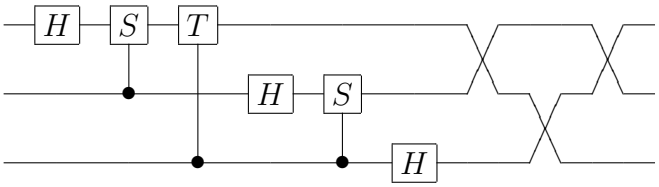


FIGURE A.6: A circuit implementing the 3-qubit quantum Fourier Transform, where S is the phase-gate and T is the $\pi/8$-gate

T used in this definition are defined as:

$$S = \begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix}, \qquad T = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{pmatrix}$$

S is know as the phase-gate, while T is sometimes called the $\pi/8$-gate (for historical reasons). H is the standard Hadamard transform defined in section 3.1.3.

This circuit can be translated into a small QML program that mimics the action of the circuit, and this is shown in figure A.7. The language QML is defined in chapter 6. In the QML implementation, cS is the controlled-S transformation, defined as:

$cS : \mathcal{Q}_2 \multimap \mathcal{Q}_2 \multimap \mathcal{Q}_2 \otimes \mathcal{Q}_2$

$cS\ c\ x = \mathbf{if}^\circ\ c\ \mathbf{then}\ (\mathbf{qtrue}, i \times \mathbf{qtrue})$

$$
\begin{aligned}
&\mathit{qft3} : \mathcal{Q}_{\mathbf{2}} \multimap \mathcal{Q}_{\mathbf{2}} \multimap \mathcal{Q}_{\mathbf{2}} \multimap \mathcal{Q}_{\mathbf{2}} \otimes \mathcal{Q}_{\mathbf{2}} \otimes \mathcal{Q}_{\mathbf{2}} \\
&\mathit{qft3}\ x\ y\ z = \mathbf{let}\ (b, a, c) \quad = (cS\ y\ (\mathit{had}\ x), z) \\
&\qquad\qquad\qquad\quad (c', a', b') \quad = (cT\ c\ a, b) \\
&\qquad\qquad\qquad\quad (c'', b'', a'') = (cS\ c'\ (\mathit{had}\ b'), a') \\
&\qquad\qquad\ \mathbf{in}\ \ (\mathit{had}\ c'', b'', a'')
\end{aligned}
$$

FIGURE A.7: The 3-qubit QFT circuit implemented in the quantum programming language QML

$$\qquad\qquad\qquad\qquad \mathbf{else}\ (\mathbf{qfalse}, \mathbf{qfalse})$$

and cT is the controlled-T transformation, defined as:

$$
\begin{aligned}
&cT : \mathcal{Q}_{\mathbf{2}} \multimap \mathcal{Q}_{\mathbf{2}} \multimap \mathcal{Q}_{\mathbf{2}} \otimes \mathcal{Q}_{\mathbf{2}} \\
&cT\ c\ x = \mathbf{if}^{\circ}\ c\ \mathbf{then}\ (\mathbf{qtrue}, (e^{i\pi/4}) \times \mathbf{qtrue}) \\
&\qquad\qquad\qquad\quad \mathbf{else}\ (\mathbf{qfalse}, \mathbf{qfalse})
\end{aligned}
$$

Finally, the Hadamard transformation can be defined in QML as

$$
\begin{aligned}
&\mathit{had} : \mathcal{Q}_{\mathbf{2}} \multimap \mathcal{Q}_{\mathbf{2}} \\
&\mathit{had}\ x = \mathbf{if}^{\circ}\ x\ \mathbf{then}\ (-1) \times \mathbf{qtrue} + \mathbf{qfalse} \\
&\qquad\qquad\qquad\quad \mathbf{else}\ \mathbf{qtrue} + \mathbf{qfalse}
\end{aligned}
$$

In QML, simple normalisation factors are inferred by the compiler and can therefore be omitted. The program *qft3* is a direct translation of the circuit implementation of the 3-qubit quantum Fourier Transform into QML, and as such has a similar structure to the circuit. It is left for future work to develop a natural QML implementation of the QFT, and then Shor's algorithms. Other example QML programs are given in chapter 6.

A.7 Summary

This appendix has presented a well known problem that can be solved more efficiently using quantum computation than is known possible classically. This problem is of both great theoretical and practical interest due to the reliance of some cryptographic protocols on the computational difficulty of factorisation.

This appendix shows how Shor's algorithm actually solves the problem of period

finding, which can then be reduced to factoring. The useful Phase–estimation and Quantum Fourier Transform algorithms are also presented. A detailed analysis of efficiency of the algorithms is also presented, which prove that Shor's algorithm can solve the factoring problem using only polynomially bounded resources with certainty. In addition, an implementation of the classical parts of Shor's algorithm is presented in Haskell, and an implementation of a 3 qubit quantum Fourier transform is given in both the circuit model and QML.