Towards a formally verified functional quantum programming language

Alexander S. Green, BSc.

A Thesis submitted for the degree of Doctor of Philosophy

School of Computer Science

University of Nottingham

July 2010

Abstract

This thesis looks at the development of a framework for a functional quantum programming language. The framework is first developed in Haskell, looking at how a monadic structure can be used to explicitly deal with the side-effects inherent in the measurement of quantum systems, and goes on to look at how a dependently-typed reimplementation in Agda gives us the basis for a formally verified quantum programming language. The two implementations are not in themselves fully developed quantum programming languages, as they are embedded in their respective parent languages, but are a major step towards the development of a full formally verified, functional quantum programming language. Dubbed the "Quantum IO Monad", this framework is designed following a structural approach as given by a categorical model of quantum computation.

Acknowledgements

I would firstly like to thank my supervisor, Thorsten Altenkirch, for all his insights that have proved to be invaluable for keeping my research on track, and headed in the right direction. I would also like to thank the rest of the FP lab at the University of Nottingham who, over the last four years, have been my first point of call for any research related advice. A special mention should go to Jonathan Grattage, whose advice and feedback was always greatly received.

I have been privileged enough to receive funding from both the EPSRC Network on Semantics of Quantum Computation, and the Foundational Structures in Quantum Information and Computation STREP grant, for trips to conferences where I have met many interesting people working in related subject areas.

The use of Agda in this thesis has been helped greatly by Ulf Norell's continuing work on the language, and I make extensive use of Ralf Hinze's and Andres Löh's implementation of *lhs2TeX* to typeset the Haskell and Agda code. I would like to thank them for their ongoing work, especially the recent addition of Agda support in *lhs2TeX*.

I would also like to thank my examiners, Ian Mackie and Henrik Nilsson, for the extensive discussion and feedback that made my viva such a pleasurable experience.

As always, I would like to thank my family, especially my Mother, whose support and patience over the years has given me the taste for knowledge necessary for researching this thesis.

Finally, I would like to thank Karen, whose love and support is a blessing.

Contents

List of Figures

Chapter 1

Introduction

1.1 Introduction

In this thesis I present my research into the field of quantum programming languages, from the perspective of a functional programmer. Quantum programming languages are implemented to give us control over machines that use the quantum mechanical aspects of superposition and entanglement in a computational manner. It has been shown that such *quantum computers* are able to perform certain tasks faster than their classical counterparts. For example, the most famous quantum algorithm is Shor's algorithm ([Sho94]) that can be used to find the factors of large numbers, a task that is infeasible on todays classical computers because of the complexity (exponential in the size of the number to be factored) of the best known classical solutions. Shor's algorithm, if run on a suitably sized quantum computer, only has a complexity that is polynomial in the size of the number to be factored (We cover Shor's algorithm in more detail in section 6.5).

Although there are many other algorithms for quantum computers (E.g. see section 2.6), their design has mainly arisen from a calculational approach, that is, using the underlying mathematical structure of quantum mechanics to derive these algorithms in quite a low-level manner. In computer science, it is common place to use higher-level structures to simplify the design of algorithms, and abstract away from all the low-level details. For example, computers run at the lowest level on bits, but computer programmers use high-level languages to abstract away from simple logic gates acting on these individual bits. As a computer scientist, it is therefore natural to think of a quantum computer in terms of how to model the low-level quantum mechanical system in terms of a higher-level language, or in terms of higher level constructs within a quantum programming language.

When looking at quantum computation, there is one key aspect that is different than in the classical paradigm. Namely, quantum computation relies upon measurements, or observations in the underlying quantum mechanical system, that are modelled by a wave-function collapse. This gives rise to the fact that measurements can cause side-effects in the entire quantum state of a system, as when part of a quantum state is measured, the overall state collapses into a state where this measurement outcome is now the only possible measurement outcome for that given part of the overall quantum state. We shall look in more detail at quantum states in Chapter 2, and go into more detail as to how measurements can have side-effects in the rest of the quantum system (2.3.3). Many of the current languages designed for quantum computation have no explicit way of dealing with these side-effects, they merely happen implicitly whenever a measurement operation occurs and lead to a programming paradigm in which reasoning about the behaviour of the programs becomes disjoint from the actual programs themselves. That is, programs written for a quantum computer are by their nature impure, and these languages in which side-effects occur implicitly have no defined structure that explains what side-effects are occurring.

In functional programming, we strive to make our programs pure. We are able to use the categorical notion of a monad to give a pure model of effectful programs. This gives rise to a programming paradigm in which side-effects must be dealt with explicitly, within the language and the programs themselves. Haskell, as the functional programming language of choice for much of this thesis, uses monads to model any form of effectful computation, and indeed even uses a monadic structure, known as the *IO Monad*, to model any form of I/O in a pure manner. Recent work ([Swi08, SA07]) has shown that a monadic approach can be used to give a sound basis for reasoning about the behaviour of effectful programs, as the monadic structure can be used to describe exactly the side-effects that are able to occur. It is from this background that the idea for designing a monadic interface to quantum computation first occurred. In this thesis I present my work on such a monadic interface to quantum computation, which has been dubbed the *Quantum IO Monad* in honour of the *IO Monad*. It is the monadic structure of the *Quantum IO Monad*, and how this leads towards a language in which the side-effects of measurement must occur explicitly that makes this work different from any of the other quantum programming languages that are being developed. The following section looks in more detail into the background of quantum programming languages, and section 1.2.1 looks at work related to the work in this thesis. Much of the work on the development of the *Quantum IO Monad* has been published as joint work with my supervisor Thorsten Altenkirch ([AG10]), and section 1.4 goes over this and other joint work presented in this thesis.

The design of a quantum programming language also depends on the constructs that are to be available within the language. The operations available in the *Quantum IO Monad* follow in no small part from a categorical model of quantum circuits that is also presented in this thesis (see chapter 3). Category theory has given rise to many ideas in functional programming (such as monads as noted previously) as it provides a sound mathematical basis for the structures that can be defined. In fact, there is even a category of Haskell programs, where Haskell's types form the objects of the category, functions definable in Haskell form the morphisms, and functional composition denotes the composition of arrows. The related work section (1.2.1) looks at more work that has been done using category theory to model the structures available in quantum computation.

In computer science, it is often very important to prove that programs have exactly the behaviour given in their specification. With the non-deterministic behaviour of quantum computation, this will become especially important for programs written for quantum machines. Currently, in many main stream languages, these proofs must be written separately from the very programs which they are *verifying*. Indeed, although there are many tools to help with such verification of computer code, it is often thought to be a hard job to derive these proofs, indeed a lot of code is only verified if it is to be used in areas where security and/or safety become important. Recent work on dependent types gives rise to languages in which the programs themselves can be thought of as proofs of their own specification. This *programs as proofs* paradigm gives rise to a redevelopment of the *Quantum IO Monad* in the dependently-typed language Agda. The approach is still able to use a monadic structure to deal with side-effects, but now the constructs available are also able to contain proofs that the unitary operations we define are unitary by their definition. The work given in this thesis on a dependently-typed implementation of the *Quantum IO Monad* gives a proof of concept, that such an approach may lead to a formally verified quantum programming language. The verification process now becomes part of the coding process, and the verifications are checked at compile time leading to compiled code that would be certifiably correct with respect to its original specifications.

As it stands, there are currently no physical realisations of a scalable quantum computer. However, research into the area is ongoing, and the number of qubits that can be realised is growing slowly. For example, a recent experimental realisation of a quantum computer was able to run Shor's algorithm over 7 qubits, to calculate the factors of 15 [VSB+01]. One field that has shown big advances is that of quantum cryptography, with implementations able to distribute provably secure keys over fibre optic networks up to 150km in length [HRP+06].

1.2 Background

The idea of quantum programming languages has been around for almost as long as the idea of having a quantum computational system. Much work focuses on how we can interface with such a device from a classical starting point. E.g. having the quantum device as an auxiliary device controllable from some classical hardware. This has lead to many languages being based on extensions to current pre-existing languages, specifically common languages such as C++. Recently, work has started to focus more on the behaviour of such a device, or more importantly the meaning of quantum programs. Keeping with the proper terminology, recent work has started to think more about the semantics of quantum computation, and how this can best be modelled in quantum programming languages.

The work presented in this thesis follows this idea, that quantum programming languages should be used to model the semantics of quantum computation, and not just as an interface to a quantum device. As such, only work related to this area is covered, and no time is spent looking at languages that don't follow such an approach.

1.2.1 Related Work

There has been much related work to that which is presented in this thesis, so this section shall just cover some of the most recent related work in some of the more specific subject areas. Firstly, there has been previous work on modelling quantum computation in a functional setting. [MB01] proposes the idea of using a "monadic style" for modelling quantum programming in a functional setting, however, their implementation doesn't quite give rise to a truly monadic structure in the Haskell sense. Haskell is also the language of choice in [Kar03], which shows how to model the underlying mathematical structures of quantum mechanics, developing a framework that focuses more on this mathematical interpretation than a programming language based approach. [Sab03] goes on to give a model of quantum computing in Haskell that is designed to give programmers an intuitive approach to quantum computation. Although this is a similar approach as taken in the design of the *Quantum IO Monad*, the model proposed isn't monadic.

More recent work, [VAS06], has looked at how quantum effects can be modelled in Haskell using the idea of *arrows*. Arrows are a generalisation of monads, and this approach allows the density matrix formulation of quantum state, along with the idea of super-operators (operations mapping density matrices to density matrices) to be modelled in a pure manner. This approach works very well, as measurement can be thought of as a specific type of super-operator, allowing computations to be defined that mix measurements with unitary operators. However, the paper shows that in general, super-operators cannot be modelled simply as monads. The approach taken in designing the *Quantum IO Monad* also follows this idea that to model quantum computation, the languages must somehow allow measurements to be part of the computations, and although we don't model super-operators explicitly, we are able to model measurements using a monadic structure. The paper finishes off by discussing that the Haskell implementation is limited, in that it is possible to define arrows that would not be physically realisable. The suggestion to overcome this problem is by using a carrier language that explicitly deals with weakening and decoherence, which can be compiled into super-operators that are realisable in the Haskell implementation.

One such language, that explicitly deals with weakening and decoherence, is QML [AG05, Gra06]. QML is presented as a language that doesn't just allow the classical control of quantum data, but introduces a form of quantum control, whereby an arbitrary quantum state can be used in a control structure. This comes in the form of a quantum "if" structure, whereby both branches of the if statement can contribute to the overall computation depending on the state of the control qubit. The work on this language gave a large input into the design of the *Quantum IO Monad*, as the same approach is followed in that we have quantum data, as well as quantum control available in our implementations. QML was presented along with a compiler written in Haskell, allowing QML programs to be compiled into quantum circuits. It has also be shown that QML programs can be directly interpreted in terms of super-operators, and as such give the language a constructive denotational semantics in terms of the arrows presented in [VAS06].

Another language of note is presented in [Sel04]. QPL introduces some of the major ideas used in designing quantum programming languages that give a more structural approach to quantum programming. QPL is a functional quantum programming language, with features that include high-level programming structures that include recursive procedures, and quantum data-types. The underlying semantics is given in the form of a categorical model of superoperators. Work on this categorical model has inspired much work on other structural models of quantum computation, and we shall look at a few of these later on in this section. The work on QPL and quantum lambda calculi has been followed up extensively in [Val08, SV09].

There are many other quantum programming languages that can be related to the work in this thesis, and the two language surveys [Gay06, Rud07] are both good starting points for looking into the what is currently available.

Quantum programming languages rely heavily on their underlying semantics, with lots of languages using quantum circuits as their target semantics. However, using such a low-level structure as the basis for a language, means a large amount of abstraction is necessary during language design. Much work has been devoted to modelling the structures in quantum computation in a categorical manner, and as such, the categorical notions arising from this work can lead to a nicer semantics from which to design quantum programming languages. Work on a categorical model of quantum circuits is presented in this thesis, which is also used to influence the design of the constructs available in the *Quantum IO Monad*. In [AC03, AC04] a categorical model is developed that can be used as a semantics for certain quantum protocols. This categorical model is also used in [Coe05] to give a diagrammatic model of quantum computation, whereby computations can be defined in terms of certain diagrams, and the underlying categorical structure gives rise to rules that can be used to simplify the given diagrams. A nice example is how a diagram that represents the teleportation protocol can be simplified to an identity "channel" between the sending and receiving parties. A different account is given in [Sel07], that extracts a *dagger* structure from the strongly compact closed categories presented in [AC04]. The paper shows how such categories with a dagger structure can be given another diagrammatic interpretation, but also sketches a proof that shows how such categories can be used in equational reasoning. Indeed, it has been remarked that the category **FQC** presented later in this thesis could be alternatively given as an instance of a dagger-complete category.

In designing the *Quantum IO Monad*, we have taken the approach that the side-effects inherent in the measurement operations of quantum computation can be modelled using a monadic structure. This work is heavily based on [SA07, Swi08], which introduce ideas on how to reason about classical effects in functional programming languages. They go on to extend those ideas using a dependently typed setting to give "meaning" to the effects that can occur. Knowing the behaviour of effects gives rise to a sound framework for reasoning about effectful programs. As such, following a similar approach for quantum programming, should lead to a sound framework for reasoning about quantum programs.

1.3 Overview of thesis

This thesis presents in chapter 2 an introductory overview of quantum computation. Section 2.2 begins by introducing reversible computation. Reversible computation is an excellent starting point for looking at quantum computation, as the definition of reversible computations is directly related to the definition of unitary operators in quantum computation. Indeed, we use the classical analogue of reversible computation in many examples throughout the thesis, such as the classical subset of QIO, and a toy language of reversible circuits implemented in

Haskell and Agda. The chapter goes on to give an introduction and overview of quantum computation. This includes superposition, measurement,and entanglement in section 2.3, and goes on to look at the quantum gate model (2.5), and an overview of some of the most famous quantum algorithms (2.6).

Chapter 3 introduces a categorical model of reversible circuits, with specific instances of the category that model both classical reversible circuits, and quantum circuits. The chapter goes on to describe a category of irreversible computations, that can be embedded into the given category of reversible circuits by introducing a notion of *Heap* inputs, and *Garbage* outputs. Much of the work in this chapter inspires the choice of operators used later in the definition of the *Quantum IO Monad*.

Chapter 4 gives an introduction to functional programming, and the functional language Haskell. The chapter focuses on the aspects of Haskell that are used extensively in the definition of the *Quantum IO Monad*, such as effects, type-classes, monads and Haskell's *do* notation. The chapter also introduces a toy language of reversible circuits, with an evaluator function used for *running* the circuits.

The following chapters look at the design of the *Quantum IO Monad*, and its implementation in Haskell. An implementation of a few of the most famous quantum algorithms written in *QIO* are also given. Chapter 7 finishes off with an explanation of some of the pitfalls of the Haskell implementation of the *Quantum IO Monad* and describes how a dependently-typed implementation could be used to iron out these pitfalls.

Chapter 8 gives an introduction to dependent types, and more specifically the dependently-typed language Agda. It looks in depth at how programs can be thought of as proofs of their own specification, and gives a reimplementation of the toy language of reversible circuits, in which the evaluator function doesn't just enable the circuits to be run, but also gives a formally verified proof that the circuits are indeed reversible.

Having introduced Agda, chapter 9 goes on to look at the redevelopment of the *Quantum IO Monad* in Agda, looking in detail at an implementation of the classical subset of *QIO* whereby proofs of reversibility are encoded in the semantics of the programs. The following section looks at how this is extended to the full quantum implementation of *QIO*, whereby the unitarity of our monoidal operators is encoded in the semantics of the computation. This section also introduces a library of *postulated* complex number operations, with proofs of the field properties of the complex numbers. Compiled *QIO* programs can be run using the underlying floating-point representation of numbers as an estimation to the complex numbers, giving code that can actually be run despite their being no current implementation of the complex numbers in Agda.

It is this Agda implementation of the *Quantum IO Monad* that gives rise to the choice of title for this thesis. Having *QIO* embedded within Agda means that it is not yet a stand-alone quantum programming language, but more of a library for quantum computation within Agda. However, using Agda as a parent language does have the added benefit that we can think of *QIO* as being a formally verified language in two different senses:

1. The semantics of the "unitary operators" available in *QIO* ensure that only truly unitary operations can be defined. That is, every member of the *USem* data-type contains a formal proof of its own unitarity.

2. The dependent-type system of Agda means that computations written using *QIO* can be formally verified, in the sense that their types can ensure certain properties of their specification. This second type of formal verification in *QIO* follows directly from the use of a parent language with dependent types.

The final chapter (chapter 12) gives a comparison of the two implementations of the *Quantum IO Monad*, and goes on to give some final conclusions and remarks on the work presented in this thesis.

1.4 Joint Work

Some of the work in this thesis has been published previously as joint work with my supervisor Thorsten Altenkirch. Firstly, much of the work in chapter 3 was published in the proceedings of QPL 2006, [GA08]. Secondly, much of the work in chapters 5, 6 and 7 is published as a chapter in the book "Semantic Techniques in Quantum Computation" [AG10].

This thesis gives an introduction to many of the individual subject areas that have been used heavily in my research. The introductory topics are all well known within their respective communities, but are included herein to complement the presentation of my own work. The following list is to clarify which parts of this thesis present my own contributions to the subject area:

- Chapter 3 introduces a categorical model of circuits. This work was originally published as joint work with Thorsten Altenkirch in the proceedings of QPL 2006 [GA08].

- Section 4.6 introduces a toy language of reversible circuits as an example of reversible computation in Haskell. This toy language was implemented by me for use in this thesis.

- Chapter 5 introduces the Quantum IO Monad in Haskell. The original work on *QIO* in Haskell was joint work with Thorsten Altenkirch, which has matured into the current implementation as described in this thesis. Although a lot of the original design decisions were joint work, the latest implementation, and corresponding code, has all been put together by me. Some of the later developments include the unitary let operation, the instances of quantum data for integers, and allowing any arbitrary single qubit rotation, instead of just providing a universal set. Much of this work has also been published in [AG10].

- Chapter 6 introduces a few of the more famous quantum algorithms implemented as computations in *QIO*. These algorithms are all well known in the field of quantum computation, but their implementations in *QIO* were done by me, and have been published as part of the chapter in [AG10].

- Chapter 7 goes over the implementation of the Quantum IO Monad in Haskell, mainly covering the implementation of the simulation functions. The original implementation followed directly from the design of the operations in *QIO* which was joint work with Thorsten Altenkirch. Again, the latest implementation, and corresponding code, has been put together by me, including the use of restricted monads to model the equality requirements. This work has also been published as part of [AG10].

- Section 8.4 re-implements the toy language of reversible circuits, from section 4.6, in Agda. This new implementation adds to the semantics of the circuits a formal proof of their reversibility. This implementation of the toy language was also defined by me for use in this thesis.

- Chapter 9 introduces all my work on an extension of the classical subset of *QIO* to Agda.

- Chapter 10 introduces all my work on extending the full set of quantum operations in *QIO* to Agda, and includes work on a library of Complex numbers for Agda that contains sufficient proofs on the field properties of complex numbers for adding some ability to reason about programs that use the complex numbers. The actual type of complex numbers introduced is a postulated type that compiles down to an underlying floating point representation.

The main scientific contributions of this thesis are the use of a monadic structure to explicitly model the effectful parts of quantum computation; namely measurements, and the move into a dependent type system to give a formal verification of the *unitary* structures.

Chapter 2

Quantum Computation

2.1 The history of quantum computation

Quantum computation is a relatively new research area, even within the realms of computer science. It's roots lie in the ways in which we can exploit the strange aspects of Quantum Mechanics in a computational manner. The idea of Quantum Computation itself didn't really appear until the 1970's, and the first description of how such a quantum computer could be modelled didn't surface until Richard Feynman's presentation in 1981 [Fey82].

There have been many advances in the area, most importantly in the types of quantum algorithm that could be designed to take advantage of this quantumness. The most famous of all is Shor's algorithm, which he first presented in 1994 [Sho94]. It offers an exponential speed up over the best known classical solution to the factorisation problem (see section 2.6.6). Other algorithms have also surfaced that are actually provably faster than their classical counterparts. We shall look at some of these algorithms in more detail later in this chapter (section 2.6).

Currently much research by physicists revolves around actually being able to implement a scalable quantum computer, which could be used to run these algorithms over more than a handful of qubits. However, in computer science, we have given ourselves the task of coming up with new languages for these hypothetical large scale quantum computers that will enable a more natural approach to the design of quantum software that can take advantage of the quantumness available. This certainly involves coming up with languages that enable quantum algorithms to be designed within them. This means that we want high-level constructs that help in the design of algorithms, such as proof structures that are able to verify the correctness of the programs being developed.

Before jumping into an introduction of quantum computation, it is good to look at the area of classical reversible computation. Classical reversible computation is closely related to quantum computation as the reversible nature corresponds very well to the unitary nature of quantum computations.

2.2 Reversible Computation

2.2.1 The history of reversible computation

Reversible computation is any form of computation that can be reversed. This means that for any change of state within the computation, enough information is kept so that the state can be returned to its previous state without any extra input. However, even at a very low level we come across functions that are irreversible. An example of an irreversible function is the logical *and* function acting on two boolean values (or equivalently bits). The *and* function returns true if and only if both the input values are true, and returns false otherwise. This operation is irreversible because, although we can deduce what the inputs were if the output is true, we have no way of knowing which of the three other input states were used if the output is false (false and false, false and true, or true and false). An example of a reversible operation on boolean values would be the *negation* operation which returns the logical negation of the input value. In fact, the *negation* operation is it's own inverse; as if we negate a boolean value twice we are back with the value we started with.

Reversible computations are also an interesting prospect when we look at Lan-

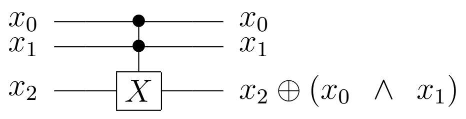


Figure 2.1: The Toffoli gate

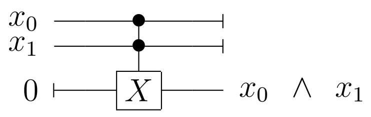


Figure 2.2: The *and* function ($\wedge$) embedded into the Toffoli gate

dauer's principle [Lan00] which states that *"any logically irreversible manipulation of information, such as the erasure of a bit or the merging of two computation paths, must be accompanied by a corresponding entropy increase in non-information bearing degrees of freedom of the information processing apparatus or its environment"*. In practical terms, this means that reversible computation provides us with a way of improving the energy efficiency of computers beyond the von Neumann-Landauer limit. This limit, given by the formula $kTln2$, where k is the Boltzmann constant and T is the temperature of the system, corresponds to the amount of energy that must be released per bit of lost information.

Any function can be thought of as reversible if it is of a one-to-one nature, but it is also possible to embed irreversible functions into a (larger) reversible function. So that it isn't the job of the programmer to ensure that the computations they produce are reversible, it is useful to look at a way of building up reversible computations from a small (universal) set of reversible functions, along with ways of combining these to build up arbitrary functions, which in themselves are reversible, but may contain or define a function that is in itself irreversible. In the next section I shall introduce the notion of reversible circuits, and how they can be used to build up arbitrary reversible computations.

2.2.2 An introduction to reversible computation

Reversible computations in the sense that we are looking at here, can be thought of in terms of reversible circuits. The Toffoli gate (Figure 2.1) is an example of a reversible circuit that is also universal. This means that any boolean function can be constructed purely from Toffoli gates, along with a notion of static inputs (input values set to a constant of either 0 or 1), and garbage (outputs that don't form part of the required output of the boolean function, but that are required so that the computation is reversible). As an example, we can construct a reversible *and* operation using only a single Toffoli gate, setting one of the inputs as a constant 0, and marking 2 of the outputs as garbage (Figure 2.2).

We shall talk more about constructing reversible (and embedding irreversible) computations in Chapter 3, and shall also introduce a category of reversible circuits.

2.2.3 The relation between reversible and quantum computation

Reversible computation is a good starting point to looking at Quantum computation as all Quantum computations are inherently unitary, and therefore of a reversible nature. In fact, as we'll see later in this thesis, the operations available in reversible computation form a subset of the operations available in quantum computation. A result of this relationship is that we are also able to use the classical analogue throughout this thesis to help introduce ideas that can then be extended into the quantum realm.

2.3 An overview of quantum computation

There are many excellent introductory texts on quantum computation, such as [NC00], and other on-line resources such as [Pre04]. In this section, I shall give an overview of the subject. I aim to introduce the concepts in such a way as to complement the rest of this thesis, introducing the topics of qubits, superposition, entanglement, unitary operators and measurement in a similar way as they are used in the Quantum IO monad later (in Chapter 5).

2.3.1 Qubits

Qubits form the basis of quantum computation, and are the quantum counterpart to bits in classical computation. In fact as computer scientists when we look at qubits we often refer to them as having the base states $|0\rangle$ and $|1\rangle$ corresponding to the two states a classical bit can be in (0 or 1). We'll see later that it is possible to use any orthogonal states as the basis for qubits, but the computational basis of $|0\rangle$ and $|1\rangle$ seems the most natural from a computational perspective. The *ket* notation ($|\ \rangle$) was first devised by Dirac [Dir82], and is now extensively used in quantum mechanics, along with its dual (the *bra*, $\langle\ |$), for describing quantum states. I shall explain a little more about the use of Dirac notation as we go on.

The main difference between qubits and classical bits is their ability to be in more than one of the base states at the same time, forming what is known as a superposition of states. The next section shall give more details about these superpositions, and go on to define what it is for a superposition to exist of more than a single qubit. Although we shall be describing qubits in terms of the states they can represent, we can also think of qubits in terms of a physical object that has the corresponding quantum mechanical properties, and as such, even qubits in an entangled state can be thought of as physically separated objects.

2.3.2 Superposition

When a qubit is in a superposition of states, it can be thought of as being in both states at the same time. However, there are some restrictions on what superpositions a qubit can be in, which are inherent in the quantum mechanical aspects of their definition. Firstly, each base state that the super-position consists of has a corresponding complex amplitude (in $\mathbb{C}$), and we are restricted by the fact that the sum of the squares of the absolute values of these amplitudes must be equal to one. So a qubit in an arbitrary super-position ($|\psi\rangle$) can be thought of as being in the state $|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle$ with the restriction that $|\alpha|^2 + |\beta|^2 = 1$ (and $\alpha, \beta \in \mathbb{C}$).

This restriction allows us to think of the possible states of a single qubit as

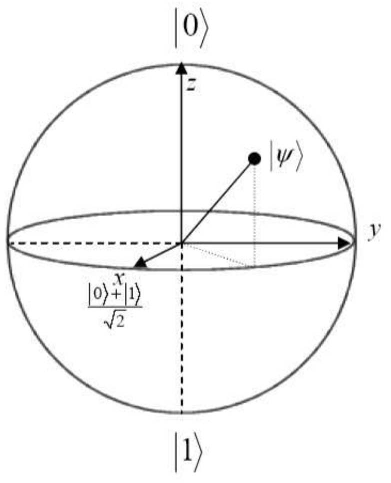


Figure 2.3: The Bloch sphere

lying on the surface of a unit sphere, commonly referred to as the Bloch sphere. Such a representation is able to occur because global phase doesn't affect physical state, and any single qubit state can be rewritten such that the amplitude of the $|0\rangle$ base state is real and non-negative. With a single qubit state reformulated in this way, we are able to write it in terms of two *angles* that represent a point on the unit sphere. That is, we have $|\psi\rangle = cos\frac{\theta}{2}\,|0\rangle + e^{i\phi}sin\frac{\theta}{2}\,|1\rangle$, with $0 \leqslant \theta \leqslant \pi$, and $0 \leqslant \phi < 2\pi$. Figure 2.3 shows a representation of the Bloch sphere showing the single qubit states $|0\rangle$, $|1\rangle$, $\frac{|0\rangle+|1\rangle}{\sqrt{2}}$ and an arbitrary single qubit state $|\psi\rangle$. As an aside, it is interesting to note that any two points that are opposite each other on the Bloch sphere are by definition orthogonal states, and as we mentioned earlier can be used as the basis for the state of a qubit.

The second, and main restriction that arises from quantum mechanics is the fact that it is impossible to know which current state an arbitrary qubit is in. For us to gain any knowledge of the state we must observe or measure the qubit. In doing so, the quantum mechanical aspects of the qubit are lost, and it will have collapsed into one of the original base states ($|0\rangle$ or $|1\rangle$). All is not lost however, and the next section shall go over the details of these measurements, and what information we can actually gain from them.

2.3.3 Measurement

As we have just seen, measurements on qubits will collapse those qubits into one of their original base states. It is possible to measure a qubit in any orthogonal basis, but we shall once again stick to the computational basis in our examples. We are however able to gain some information about the original super-position of the measured qubit as the probabilities of measuring either base state are determined exactly by their corresponding amplitudes in the original super-position. In fact, the probability of measuring a certain base state is the absolute value squared of the corresponding complex amplitude. If we look at our examples from the Bloch sphere, we can see how this might work in practice. The states $|0\rangle$, and $|1\rangle$ are quite boring as they always measure to their own value, but if we now look at the state $\frac{|0\rangle+|1\rangle}{\sqrt{2}}$, which can also be written as $\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$, we can see that both states have the same amplitude (of $\frac{1}{\sqrt{2}}$), which gives a corresponding probability of $\frac{1}{2}$ for measuring either of the computational base states. This state is said to be in an equal super-position of both the base states. More generally, for the arbitrary state $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$ we have that the probability of measuring $|0\rangle$ as $|\alpha|^2$, and the probability of measuring $|1\rangle$ as $|\beta|^2$.

As an aside, another interesting view of measurement was presented in [Eve57]. From his *Relative State* view of quantum mechanics, measurement can be thought of as a process that splits the universe into *parallel* universes, with each measurement outcome occurring in one of the parallel universes. As such, the reason we only see one outcome from a measurement is because we, as the observer, only continue to exist in the universe in which the corresponding measured state exists. This is slightly beyond the scope of this thesis, but is mentioned as it is one of the views that first interested me about the field of quantum computation.

So far, we have only been looking at the case of a single qubit, the next section shall go on to explain how a quantum system containing multiple qubits can behave, and shall go into detail as to how and what it means for qubits to become entangled.

2.3.4 Entanglement

Entanglement is a special form of superposition over a multiple qubit state. As such, it is useful to first look at arbitrary multiple qubit states. If we look at what happens in our classical analogue when we have multiple bits, we'll notice that a system containing n bits can be in any of 2^n base states, namely any of the bit strings of length n. In fact, if we move back into the quantum realm, a system of n qubits can be in an arbitrary super-position over any of these bit strings of length n. That is, the state of an n qubit system can be defined by the sum over all the complex amplitudes of each of the 2^n bit strings. We have a similar restriction as for the single qubit case that the sum of the squares of the absolute values of all the complex amplitudes must be equal to 1. For example, the state of an arbitrary three-qubit system can be defined exactly by the sum $|\psi\rangle = \alpha\,|000\rangle + \beta\,|001\rangle + \gamma\,|010\rangle + \delta\,|011\rangle + \epsilon\,|100\rangle + \zeta\,|101\rangle + \eta\,|110\rangle + \theta\,|111\rangle$, with the restriction that $|\alpha|^2+|\beta|^2+|\gamma|^2+|\delta|^2+|\epsilon|^2+|\zeta|^2+|\eta|^2+|\theta|^2 = 1$. (again with $\alpha, \beta, \gamma, \delta, \epsilon, \zeta, \eta, \theta \in \mathbb{C}$)

This exponential growth in the size of the available state space starts to hint at what gives quantum computers their increase in *power* over classical computers, but we have to keep in mind how these quantum states behave upon measurement. As we have previously seen, a single qubit will collapse into one of its base states upon measurement, and this behaviour generalises to multiple-qubit states. Upon measurement of all the qubits in a multiple qubit state, the whole system will be in one of the bit string base states, again with each base state having the probability of being measured equal to the square of the absolute value of its corresponding complex amplitude. However, it is also possible to only measure a subset of all the qubits in a multiple qubit state. Each time a qubit is measured, it will be in one of it's base states ($|0\rangle$ or $|1\rangle$), and in doing the measurement, every base state in the multiple-qubit super-position in which the qubit was in the opposite state is removed from the overall state of the system. That is, the measurement of a single qubit has a side-effect that can affect future measurements of other qubits in the system. This dependency between qubits is known as entanglement. As an example we'll look at some two qubit super-positions.

The state $\frac{1}{2}(|00\rangle + |01\rangle + |10\rangle + |11\rangle)$ is in an equal super-position of two qubits, that is, the probability of measuring any of the two-qubit base states is equal, namely $\frac{1}{4}$. This is a valid two-qubit state, but is not an entangled state as the measurement of either of the qubits doesn't effect the state of the other qubit. The probability of measuring the first qubit as $|0\rangle$ is calculated by adding the probabilities of all the two-qubit base states in which the first qubit is a 0, namely $|00\rangle$ and $|01\rangle$. Similarly, we can calculate the probability of measuring the first qubit as $|1\rangle$ by summing the probabilities of all the two-qubit states in which the first qubit is a 1, namely $|10\rangle$ and $|11\rangle$. In each case here we get the probability of $\frac{1}{2}$. Upon measurement, all the base-states which contain the first qubit in the state it wasn't measured in are removed from the overall state, and the amplitudes are re-normalised to reflect this. So, in this example, measuring a $|0\rangle$ or a $|1\rangle$ leaves the rest of the system in the same state, namely $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$, meaning that our original state wasn't an entangled state. It is also possible to see that our original state wasn't an entangled state as we could have re-written it straight away as the tensor product $(\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)) \otimes (\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle))$. The tensor product $\otimes$ is a useful operator to define the combination of quantum systems. In fact, in using $|\ \rangle$ notation it is used implicitly a lot of the time, with for example $|0\rangle \otimes |0\rangle$ being given as simply $|00\rangle$.

If we now look at the two-qubit state $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$, which is a bell state([Bel64]) and an example of a maximally entangled two-qubit state, we can see how entanglement can effect measurements. This is a valid two-qubit state, as it is customary to leave out base states whose amplitudes are 0 when describing super-positions. In this example, if we were to measure the first qubit, we would as before work out the probabilities for measuring $|0\rangle$ or $|1\rangle$ by summing the probabilities for the corresponding two-qubit base states whose first qubit is in either state. Doing this, we again see a probability of $\frac{1}{2}$ for measuring either of the two single qubit base states. However, after measurement, when we look to see what state the rest of the system will be in, we'll notice that there is only one state the second qubit can be in, namely exactly the same state as in which we have measured the first qubit. In other words, measuring the first qubit has had the side-effect of collapsing the second qubit into one of its base states.

It is this type of entanglement that lead to the EPR paradox [EPR35], which is often cited today as showing that quantum mechanics violates classical intuition. For example, classical intuition cannot explain the non-locality of measurements that is used when defining quantum teleportation. We shall look at quantum teleportation in some detail in section 2.6.7, and more information on the EPR paradox can be found in [Bel64] and [Mer85].

We have talked about quantum states, which can involve super-positions, and entanglement, but not really mentioned how these quantum states can be used for computation. We shall now introduce the concept of unitary transformations, and how they can be used to define quantum computations.

2.4 Unitary transformations

Unitary transformations describe the changes in state of the qubits involved in a computation. We have seen that we can define a quantum state of n qubits in terms of the complex amplitudes of the corresponding 2^n bit strings of length n. In fact, it is useful to be able to think of the state space of a quantum system in terms of the state spaces of the smaller systems it is comprised of. In quantum mechanical terminology, the state space of a single qubit is the two-dimensional Hilbert space, which we denote $\mathcal{H}_2$. A Hilbert space is defined as a complex vector space with an inner product, that is complete with respect to the inner product. Members of such a Hilbert space are defined by complex valued vectors of the same dimension as the Hilbert space, so the members of $\mathcal{H}_2$, are complex valued vectors of dimension two. In fact, it is this representation of quantum states that gives us Dirac's bra-ket notation ([Dir82]) that we have already been using. That is, that the inner product of a *bra* ($\langle\psi|$) and a *ket* ($|\phi\rangle$) can be given by the *bra-ket* construction $\langle\psi \mid \phi\rangle$.

$$|0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \text{ and } |1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

More generally, from an arbitrary one qubit state $|\psi\rangle$, we can derive

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

We can now generalise this quantum mechanical view to quantum states of more than a single qubit. The state space of an n qubit system is the tensor product of the corresponding n two-dimensional Hilbert spaces. In other words, the state space of an n qubit system is the Hilbert space of dimension 2^n, denoted by $\mathcal{H}_{2^n}$, and as such, members of this state space can be denoted by a complex valued vector of dimension 2^n whereby each element of the vector represents the complex amplitude of the corresponding base state. For example, the two-qubit bell state we introduced in the measurement section can be given by

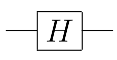
$$\tfrac{1}{\sqrt{2}}(|00\rangle + |11\rangle) = \tfrac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

If we were to use this terminology, unitary transforms for an n qubit system would relate to a unitary complex valued matrix of dimension $2^n \times 2^n$. That is, a matrix that when multiplied by its conjugate transpose, gives the identity matrix. For a small number of qubits, this representation of unitary operators is quite straight forward, but the exponential growth in the dimensions of the matrix needed to represent a unitary means that for more than a few qubits this approach isn't such a useful one. The next section looks at how we can look at unitary operators in terms of quantum gates, and circuits. These operators acting on a small number of qubits, are often given in terms of their matrix representation, and the rules governing how circuits can be constructed from these gates are also defined in terms of the mathematical operations in the corresponding matrix representation. Chapter 3 goes on to look at a categorical model of these quantum gates and circuits.

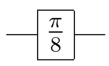
2.5 The quantum gate model

The quantum gate model gives us a way of modelling unitary transformations over more than a few qubits in a simple way. Quantum circuits are defined by a universal set of quantum gates, and rules that govern how these gates can be combined. The gates themselves are usually given in terms of their corresponding matrix representation, and the combinators are given in terms of the underlying functions on matrices that they represent. There is more than one set of universal quantum gates, but the combinators usually boil down to parallel composition being defined in terms of the tensor product of matrices, and sequential composition relating to matrix multiplication. In chapter 3, we present a categorical model of circuits, and our choice of quantum gates in this model leads on to the choice of unitary operators in our implementation of *QIO* (chapter 5). The quantum gate model also relates very nicely to the classical reversible circuits we used in section 2.2.

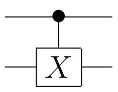
We now look at one specific example of a universal set of gates, although other universal sets of quantum gate do exist. We introduce here the universal family of gates introduced in [NC00], which also includes proofs of their universality (pages 188 to 198), in the sense that any unitary operator can be approximated up to an arbitrary accuracy. The family consists of the *Hadamard* gate, the $\frac{\pi}{8}$ gate, and the *CNOT* gate. Figure 2.4 gives the diagrams that we shall use to represent these gates, along with the corresponding matrix representations of the unitaries (We omit the *phase* gate here as it can be defined in terms of two $\frac{\pi}{8}$ gates).



$$\equiv \frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$



$$\equiv \begin{bmatrix} 1 & 0 \\ 0 & e^{i\frac{\pi}{4}} \end{bmatrix}$$

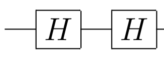


$$\equiv \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

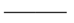
Figure 2.4: A universal set of quantum gates, consisting of the *Hadamard* gate, the $\frac{\pi}{8}$ gate, and the *CNOT* gate.

Arbitrary unitary operations can now be defined as quantum circuits using these gates along with the operations of sequential composition and parallel composition. Sequential composition joins circuits of the same arity (number of qubits) in sequence, and corresponds to matrix pre-multiplication of the first unitary by the second unitary. Parallel composition joins circuits so that they run in parallel over different qubits (giving rise to a circuit whose arity is the sum of the underlying arities). Parallel composition corresponds to the tensor product of the matrix representations.

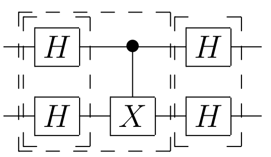
As an example, we can use sequential composition to show that the Hadamard gate is its own inverse,



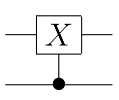
$$\equiv \frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \cdot \frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \equiv \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \equiv$$



Another example that uses both parallel and sequential composition is to show that following circuits are equivalent.

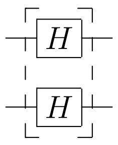


$$\equiv$$



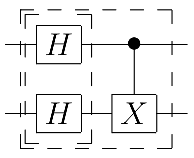
$$\left(\equiv \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \right)$$

The groupings in the diagram represent the order in which we shall perform the steps of the matrix calculations (Although in practise the order in which we do the two matrix multiplications doesn't matter as the operation is associative). The first step is to define the matrix representation of the unitary that performs two *Hadamard* gates in parallel.



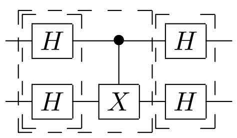
$$\equiv \frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix} \otimes \frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix} \equiv \frac{1}{2}\begin{bmatrix}1 & 1 & 1 & 1\\ 1 & -1 & 1 & -1\\ 1 & 1 & -1 & -1\\ 1 & -1 & -1 & 1\end{bmatrix}$$

Next, we calculate the unitary that is obtained by running this in sequence with the *CNOT* gate.



$$\equiv \begin{bmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 0 & 1\\ 0 & 0 & 1 & 0\end{bmatrix} \cdot \frac{1}{2}\begin{bmatrix}1 & 1 & 1 & 1\\ 1 & -1 & 1 & -1\\ 1 & 1 & -1 & -1\\ 1 & -1 & -1 & 1\end{bmatrix} \equiv \frac{1}{2}\begin{bmatrix}1 & 1 & 1 & 1\\ 1 & -1 & 1 & -1\\ 1 & -1 & -1 & 1\\ 1 & 1 & -1 & -1\end{bmatrix}$$

Finally, we calculate the unitary that we obtain from running the previous unitary in sequence again with the two parallel *Hadamard* gates.



$$\equiv \frac{1}{2}\begin{bmatrix}1 & 1 & 1 & 1\\ 1 & -1 & 1 & -1\\ 1 & 1 & -1 & -1\\ 1 & -1 & -1 & 1\end{bmatrix} \cdot \frac{1}{2}\begin{bmatrix}1 & 1 & 1 & 1\\ 1 & -1 & 1 & -1\\ 1 & -1 & -1 & 1\\ 1 & 1 & -1 & -1\end{bmatrix} \equiv \begin{bmatrix}1 & 0 & 0 & 0\\ 0 & 0 & 0 & 1\\ 0 & 0 & 1 & 0\\ 0 & 1 & 0 & 0\end{bmatrix}$$

Although we have given this universal family of circuits, in the rest of this section we shall use gates that represent arbitrary single qubit rotations, which are given along with their corresponding matrix representation. We also use controlled versions of larger unitaries, for ease of notation, although these too could be derived from this universal family of gates we have given above.

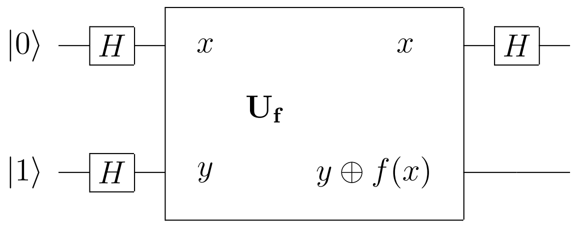
Having introduced the concept of quantum computation, we shall now go on to look at some famous quantum algorithms and how they use quantum states, along with entanglement to achieve classically infeasible tasks. In many cases, we also include the corresponding circuit diagram that represents the algorithm as a quantum computation.

2.6 Quantum Algorithms

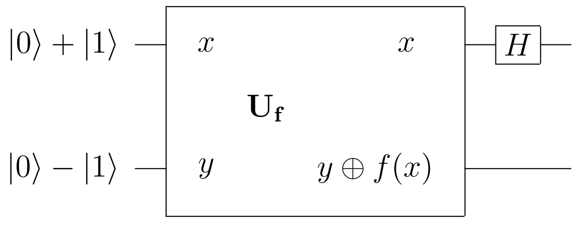
2.6.1 Deutsch's Algorithm

Deutsch's algorithm [Deu85] is often used as an introduction to quantum algorithms as it is in essence the most simple algorithm to have been found that can be proved to give a result *faster* than the classical solution to the same problem. It involves being given an unknown function that takes a single boolean value as input, and returns a single boolean value as its output. You are then asked to find out whether the function you have been given is balanced or constant whilst only applying the function as few times as possible. Classically, it is easy to show that you must apply the function twice to get a certain result (once for each possible input value), but using Deutsch's algorithm, it is possible to apply the function only once (albeit over a quantum state), and gain enough information from the result to determine the solution to the original problem. It is a simple algorithm to introduce first as it only uses two qubits, and as such is small enough so that we can go through all the mathematical detail without filling too much space.

- We start with the top qubit in the state $|0\rangle$ and the bottom qubit in state $|1\rangle$.



- After the First Hadamard gates we are left with the top qubit in the $|+\rangle$ state ($|0\rangle + |1\rangle$), and the bottom qubit in the state $|-\rangle$ ($|0\rangle - |1\rangle$).



- Depending on the function f we have one of four outcomes

  - $f(x) = 0$, has no overall effect on either qubit so we are left in the state $(|0\rangle + |1\rangle)(|0\rangle - |1\rangle)$ as before.
  - $f(x) = 1$, adds 1 onto the value of the bottom qubit for both parts of the value of the top qubit, so we are left with the state $(|0\rangle + |1\rangle)(|1\rangle - |0\rangle)$, which is equivalent to the state $-(|0\rangle + |1\rangle)(|0\rangle - |1\rangle)$.
  - $f(x) = x$, adds 1 onto the value of the bottom qubit for the $|1\rangle$ part of the top qubit, so we are left with the state $|0\rangle\,(|0\rangle - |1\rangle) + |1\rangle\,(|1\rangle - |0\rangle)$, which is equivalent to the state $|0\rangle\,(|0\rangle - |1\rangle) - |1\rangle\,(|0\rangle - |1\rangle)$ and can be simplified to $(|0\rangle - |1\rangle)(|0\rangle - |1\rangle)$.
  - $f(x) = \neg x$, adds 1 onto the value of the bottom qubit for the $|0\rangle$ part of the top qubit, so we are left with the state $|0\rangle\,(|1\rangle - |0\rangle) + |1\rangle\,(|0\rangle - |1\rangle)$, which is equivalent to the state $|0\rangle\,(|1\rangle - |0\rangle) - |1\rangle\,(|1\rangle - |0\rangle)$ and can be simplified to $(|0\rangle - |1\rangle)(|1\rangle - |0\rangle)$ or $-(|0\rangle - |1\rangle)(|0\rangle - |1\rangle)$.

- We can see that for the constant cases ($f(x) = 0$ and $f(x) = 1$) we have the states $\pm(|0\rangle + |1\rangle)(|0\rangle - |1\rangle)$, and for the balanced cases ($f(x) = x$ and $f(x) = \neg x$) we have the states $\pm(|0\rangle - |1\rangle)(|0\rangle - |1\rangle)$. We can now apply the final Hadamard in both cases.

$$\begin{array}{rcl} \pm|0\rangle + |1\rangle & -\boxed{H}- & \pm|0\rangle \\ |0\rangle - |1\rangle & \text{———} & |0\rangle - |1\rangle \end{array} \qquad \begin{array}{rcl} \pm|0\rangle - |1\rangle & -\boxed{H}- & \pm|1\rangle \\ |0\rangle - |1\rangle & \text{———} & |0\rangle - |1\rangle \end{array}$$

- As global phase (the $\pm$) is not observable under measurement, we can now measure the top qubit and ascertain exactly whether the original function was one of the balanced or one of the constant functions, measuring a $|1\rangle$ or a $|0\rangle$ respectively.

2.6.2 Deutsch-Jozsa Algorithm

The Deutsch-Jozsa algorithm [DJ92] is an extension to, and generalisation of, Deutsch's algorithm to determine whether a boolean function of an arbitrary number of inputs, but still only one output, is balanced or constant (and it is guaranteed to be one of these). For a given function which takes n input booleans, there

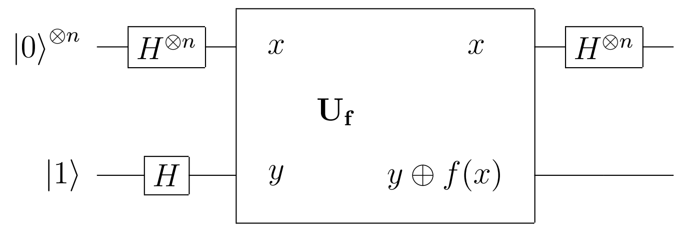


Figure 2.5: The Deutsch-Jozsa Algorithm

is now an input domain of size 2^n, and classically it can be seen that in the worst case the function will have to be applied to one more than half of these possible inputs ($2^{n-1} + 1$) to see if the function is balanced or constant. However, just as before for Deutsch's algorithm, we can use the Deutsch-Josza algorithm to attain a solution having only had to apply the given function once (again, over a quantum state). The algorithm works in a very similar way to Deutsch's algorithm, requiring one more qubit than the number of inputs to the boolean function. Figure 2.5 shows us the circuit for the Deutsch-Jozsa algorithm, where a measurement of the top n qubits after the application of the circuit will reveal whether the input function f was constant or balanced.

2.6.3 Simon's Algorithm

Simon's algorithm [Sim94] is another quantum algorithm that was shown to give a solution faster than the best possible classical solution. It gives the solution to a very specific (somewhat contrived) problem, in a time exponentially faster than the best known classical solution. We mention it briefly here as it is often cited as the inspiration behind Shor's factorisation algorithm which we look at below.

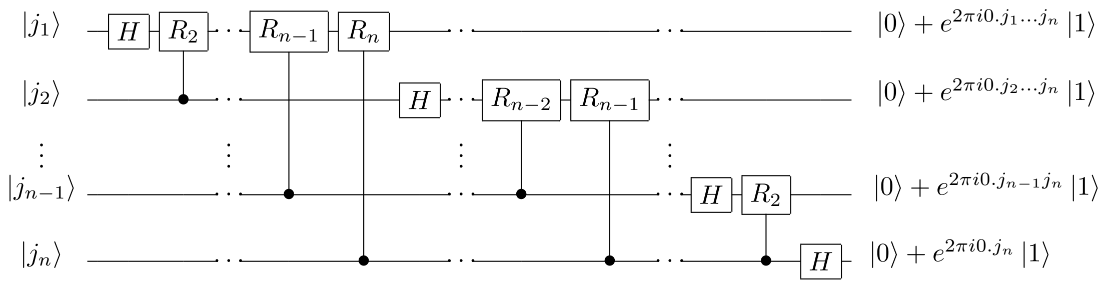
The problem involves being given a function $f : 0,1^n \rightarrow 0,1^m$ with $m \geqslant n$. The function is guaranteed to be either one-to one, or there exists a bit string of length n (s), such that for any pair of different inputs to the function (x,y), $f(x) = f(y)$ if and only if $x \oplus s = y$ (and $y = x \oplus s$). Simon's algorithm is then able to determine which of the cases the function falls into, and in the second case is able to return the value of s. Simon described this algorithm as being a solution to the problem *Is a function invariant under some xor-mask?*.

2.6.4 Grover's Algorithm

Grover's algorithm [Gro96], otherwise known as Grover's database search algorithm, was discovered by Lov Grover in 1996. It is a quantum algorithm for searching an unsorted database. It gives a quadratic speed-up over the fastest classical solution, which is a linear search. For a database of size N, we must encode a quantum state over N distinct base-states, and define a unitary that is able to add a negative phase only to the base state that represents the element we are searching for. Grover showed that such a unitary is able to be defined, and used in a probabilistic quantum algorithm to return the state being searched for.

2.6.5 Quantum Fourier Transform

Although not necessarily thought of as a quantum algorithm in its own right, the quantum Fourier transform is used in many other quantum algorithms to extract certain information we want from a quantum super-position of states. More precisely, the quantum Fourier transform is used to increase the amplitudes of certain states in a super-position that represent the states that are "useful" for gaining the result of certain quantum algorithms. This is a very over-simplified view of what the quantum Fourier transform does, but this view of it is the basis for how Shor defined his factorisation algorithm. His algorithm uses the quantum Fourier transform to extract the period of a modular exponentiation function, from a super-position of the results of the modular exponentiation function applied to an equal superposition of its possible input states. The quantum Fourier transform can basically be thought of as the fast discrete Fourier transform applied to a quantum register. The discrete Fourier transform can be thought of as mapping functions in the time domain into functions in the frequency domain. In other words, decomposing a function into a series of sinusoidal functions of different frequencies. An excellent derivation of the quantum Fourier transform is given in [NC00] on pages 216 to 221. The circuit they derive is given here in figure 2.6 for reference.



$|j_1\rangle$
$|j_2\rangle$
$|j_{n-1}\rangle$
$|j_n\rangle$
H
R_2
R_{n-1}
R_n
R_{n-2}
$|0\rangle + e^{2\pi i 0.j_1 \ldots j_n} |1\rangle$
$|0\rangle + e^{2\pi i 0.j_2 \ldots j_n} |1\rangle$
$|0\rangle + e^{2\pi i 0.j_{n-1} j_n} |1\rangle$
$|0\rangle + e^{2\pi i 0.j_n} |1\rangle$

Figure 2.6: A circuit for the Quantum Fourier transform, where R_k is given by the unitary $\begin{bmatrix} 1 & 0 \\ 0 & e^{\frac{2\pi i}{2^k}} \end{bmatrix}$

2.6.6 Shor's Algorithm

Shor's algorithm is possibly the most famous of all the quantum algorithms as it provides an exponential speed up over the fastest known classical solution to what is thought to be a classically infeasible operation (requiring a process that runs for an exponential amount of time compared to the input). Shor showed [Sho94] that finding the prime factors of a large integer could be restated as the problem of finding the period of a specifically constructed modular exponentiation function, and he went on to give a quantum algorithm that could solve this task in polynomial time. This polynomial time solution does however require a suitably sized quantum computer with enough qubits to encode both the input and output integers, which is many more than the handful available in current implementations (such as [VSB$^+$01]).

Shor's algorithm is sometimes referred to as the "Killer Application" for quantum computers. This nomenclature came about because factorising large numbers was thought to be so computationally infeasible that it forms the basis of the RSA encryption protocol ([RSA77]). The RSA encryption protocol is a very widely used public-key protocol for sending secure information over public channels such as the Internet. It works on the principle that multiplying 2 large prime numbers (p and q) is computationally easy, and a public and private key can be computed from the result ($n = p * q$). However, if it is possible for an eavesdropper to compute p and q from n, then it is also possible for them to work out the private key. If we

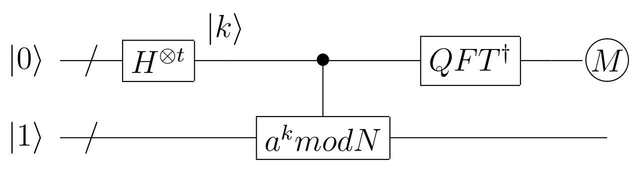


Figure 2.7: Shor's algorithm

can now factor large numbers in polynomial time, this encryption scheme has effectively been broken allowing anyone with a sufficiently large quantum computer to intercept and decode encrypted data.

Peter Shor exploited the fact that the factors of a number can be computed from the period of a given modular exponentiation function. In fact, the exponentiation function required was shown to be of the form $f(x) = a^x modN$ where N is the number we wish to factorise, and a is known to be co-prime to N. That is, that the greatest common divisor of a and N is 1. Calculating the greatest common divisor of two numbers can be done efficiently classically, using the Euclidean algorithm, and hence finding values for a can also be done classically. Depending on the value of a, it is possible that the period of this function can be used to find factors of the input. Shor discovered that finding the period of this function (and other periodic functions) can be done efficiently on a quantum computer. The efficiency of this period-finding algorithm comes directly from the use of quantum parallelism.

Two quantum registers (of sufficient number of qubits to represent N) are initialised into the state $|\vec{0}\rangle$. The first of these two quantum registers is placed into an equal super-position of all its possible base states (using Hadamard rotations). This register ($|k\rangle$) is then used as x in our exponential function $f(x) = a^x modN$ such that the result of the function is stored in the second quantum register. At this point in the computation our quantum registers will be in the state ($|k, a^k modN\rangle$) in such a way that each value of k in the first register is entangled with its corresponding $a^k modN$ in the second register. Shor goes on to show that the application of a discrete Fourier transform to the first register will (with high probability) yield the period of the given input function.

Figure 2.7 shows a slightly idealised quantum circuit representation of the part of Shor's algorithm that calculates the period of the given modular exponentiation function. The period obtained as a result of running this quantum circuit (p) can be used to find the factors of the original input by noting that as long as p is even, and $a^{\frac{p}{2}} \neq -1 mod N$, then the greatest common divisors of $a^{\frac{p}{2}} + 1$ and $a^{\frac{p}{2}} - 1$ are factors of N.

2.6.7 Quantum Teleportation

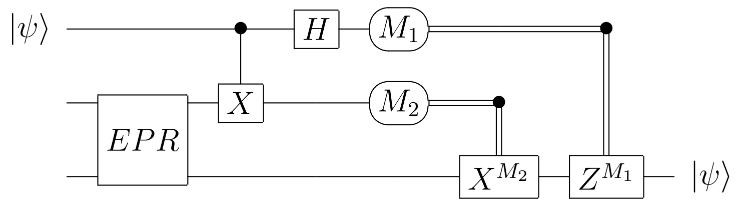
Quantum teleportation can be thought of as the transfer of an arbitrary quantum state from one qubit to another, and is achieved without knowing the original (input) state. The basic process involves having an entangled pair of qubits, which may be physically separated. The unknown (input) qubit is entangled with one of the qubits from the original entangled pair, and these are both then measured. The outcome of these measurements must then be transmitted (classically) to the site of the second qubit from the entangled pair, and depending on these measurements one of 4 adjustments is made to the second qubit. This second qubit is then in the state of the original (input) qubit. The operation doesn't break the no-cloning theorem as the state of the original qubit is collapsed upon measurement, and the operation doesn't allow faster than the speed of light communication as the classical information can only be transmitted at this limiting speed. An example text-book description of quantum teleportation as taken from ([NC00]) is as follows:

> Alice and Bob met long ago but now live far apart. While together they generated an EPR pair, each taking one qubit of the EPR pair when they separated. Many years later, Bob is in hiding, and Alice's mission, should she choose to accept it, is to deliver a qubit $|\psi\rangle$ to Bob. She does not know the state of the qubit, and moreover can only send *classical* information to Bob. Should Alice accept the mission?
>
> Intuitively, things look pretty bad for Alice. She doesn't know the

state $|\psi\rangle$ of the qubit she has to send to Bob, and the laws of quantum mechanics prevent from determining the state when she only has a single copy of $|\psi\rangle$ in her possession. What's worse, even if she did know the state $|\psi\rangle$, describing precisely takes an infinite amount of classical information since $|\psi\rangle$ takes values in a *continuous* space. So even if she did know $|\psi\rangle$, it would take forever for Alice to describe the state to Bob. It's not looking good for Alice. Fortunately for Alice, quantum teleportation is a way of utilising the entangled EPR pair in order to send $|\psi\rangle$ to Bob, with only a small overhead of classical communication.

In outline, the steps of the solution are as follows: Alice interacts the qubit $|\psi\rangle$ with her half of the EPR pair, and then measures the two qubits in her possession, obtaining one of four classical results, 00, 01, 10, and 11. She sends this information to Bob. Depending on Alice's classical message, Bob performs one of four operations on his half of the EPR pair. Amazingly by doing this he can recover the original state $|\psi\rangle$!

The following quantum circuit gives a more precise description of quantum teleportation.



The top two lines represent Alice's qubits, that is the input qubit (which is in the state $|\psi\rangle$), and her qubit from the EPR pair. The bottom line represents Bob's qubit. Alice entangles the input qubit with her EPR pair qubit using a controlled Not operation. Then she performs a Hadamard rotation on the input qubit before measurement (which is equivalent to a measurement in the Hadamard basis). The double lines coming from the measurements represent the classical data that she sends to Bob, who correspondingly has to perform (conditionally depending on the classical bits) an X and a Z rotation on his qubit, which will then be in the state $|\psi\rangle$. It is also clear to see that the state of the original input qubit has been lost as it will be in one of the base states, $|0\rangle$ or $|1\rangle$, depending upon the measurement outcome.

Chapter 3

A categorical model of circuits

In this chapter I introduce work on a categorical model of both reversible and quantum circuits [GA08]. Our model, dubbed $\mathbf{FxC}^{\simeq}$, works on the premise that irreversible computations are a derived notion, and the underlying reversibility comes from the physical nature of the universe. In the previous chapters we have seen how both reversible and quantum computations can be thought of in terms of circuits, or more precisely a universal group of circuits along with notions of how these circuits can be composed (both in serial and in parallel). The categorical model introduced in this chapter introduces a family of such universal circuits and compositions as morphisms in the defined category. Our diagrammatic approach means that defining circuits in the category is as easy as piecing together smaller circuits, while the underlying categorical structure ensures the correctness of all the mathematical properties.

3.1 Generalised reversible circuits

3.1.1 Objects of $\mathbf{FxC}^{\simeq}$

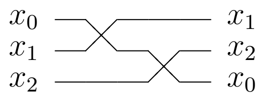
In a categorical setting, it is useful to look at groupoids when defining reversible computations. Groupoids are a form of category in which every morphism is in fact an isomorphism, that is, that every morphism in the category has an inverse morphism in the category such that each morphism and its inverse form an isomorphism. The category $\mathbf{FxC}^{\simeq}$ is such a groupoid, and shall be defined as a strict groupoid in which we can assume any isomorphic objects are in fact equal. So, we have the groupoid $\mathbf{FxC}^{\simeq}$, with every morphism $\psi \in \mathbf{FxC}^{\simeq}(a, b)$ having an inverse $\psi^{-1} \in \mathbf{FxC}^{\simeq}(b, a)$. The strictness of the groupoid means that $\mathbf{FxC}^{\simeq}(a, b)$ is empty if $a \neq b$, so we can define morphisms as members of homsets $\mathbf{FxC}^{\simeq}(a, a)$, and simplify our presentation by writing $\mathbf{FxC}^{\simeq}\, a$ in place of $\mathbf{FxC}^{\simeq}(a, a)$.

To allow our morphisms to be composed in a parallel manner, we define that our category (or groupoid) $\mathbf{FxC}^{\simeq}$ has a strict monoidal structure, with identity object I, and the operation $\otimes$ which corresponds directly to the parallel composition operation. There is also a special object corresponding to the Booleans, which can be denoted by $\mathbb{N}_2$. All other objects in the category are generated directly from these three objects, and we can again simplify our presentation by using the natural numbers ($\mathbb{N}$) to denote the objects. More specifically, the natural number $a \in \mathbb{N}$ shall be used to denote the object 2^a, and our three generator objects can be denoted such that $I = 0$, $\mathbb{N}_2 = 1$, and for objects a and b, $a \otimes b$ can be thought of as addition of the natural number representation of a and b $(a + b)$.

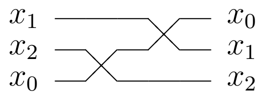
3.1.2 Morphisms of $\mathbf{FxC}^{\simeq}$

Now we have defined our category, and its objects, we can start to define the morphisms of the category. It is the morphisms of the category that directly relate to reversible circuits, and as such we shall present them diagrammatically, and in an inductive manner. As $\mathbf{FxC}^{\simeq}$ is a groupoid, we shall also present the inverses of the given morphisms. These inverses, as would be expected, also define the inverses of the circuits that the morphisms represent. A morphism $\psi \in \mathbf{FxC}^{\simeq}\, a$, can simply be thought of as a reversible circuit with both a inputs, and a outputs, and is hence why the strictness of the groupoid can be thought of as the preservation of information in our circuits.

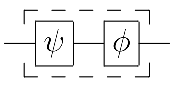
The first morphism we shall define corresponds to **rewirings** in the circuit model. A set of (neighbouring) wires can be thought of as the initial segment of the natural number representation of the objects. E.g. each wire (or $\mathbb{N}_2$ object) is denoted by its corresponding natural number, and the set of n wires is denoted by the set of natural numbers less than n. We shall write $[a] = \{i \in \mathbb{N} \mid i < a\}$ as such an initial segment of $\mathbb{N}$. Rewirings can then be thought of as bijections acting on these initial segments. For $\phi : [a] \simeq [a]$, we have the morphism $\text{wires}\,\phi \in \mathbf{FxC}^{\simeq}\,a$ as the associated rewiring. As an example, we can give the following rewiring diagram:



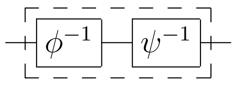
which would be defined by the bijection $\phi(0) = 2$, $\phi(1) = 0$, and $\phi(2) = 1$, and hence the morphism $\text{wires}\,\phi \in \mathbf{FxC}^{\simeq}, 3$. The existence of wires follows from the strict monoidal structure of $\mathbf{FxC}^{\simeq}$, and the identity morphism (id_a) is just a special case of wires (where the bijection $\phi : [a] \simeq [a]$, is just the identity). The inverse of a rewiring morphism is defined exactly by the inverse of the embedded bijection. For example, for the rewiring given above, we would have the inverse bijection $\phi^{-1} : [a] \simeq [a]$, such that $\phi^{-1}(0) = 1$, $\phi^{-1}(1) = 2$, and $\phi^{-1}(2) = 0$. Giving rise to the inverse morphism $\text{wires}\,\phi^{-1} \in \mathbf{FxC}^{\simeq}, 3$ and the diagram (keeping the same labels as before):



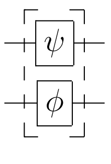
The second morphism we can define corresponds to the **sequential composition** of circuits. Thinking of the circuit model, it is quite straightforward to realise that only circuits of the same arity can be joined in sequence, e.g the number of output wires from the first circuit must correspond exactly to the number of input wires to the second circuit. In $\mathbf{FxC}^{\simeq}$ we define sequential composition by the operation $\circ$, such that given $\psi, \phi \in \mathbf{FxC}^{\simeq} a$ we can construct $\phi \circ \psi \in \mathbf{FxC}^{\simeq}\,a$. Diagrammatically, sequential composition is achieved by simply joining the output wires of the first circuit (ψ), with the input wires of the second circuit (ϕ).



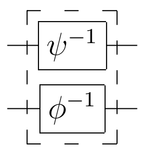
The inverse of a sequential composition corresponds to a composition of the inverses of the original sub-circuits. This composition is also a sequential composition, but the inverses are joined in the reverse order. So, in $\mathbf{FxC}^{\simeq}\,a$, the inverse of the above sequential composition is achieved using ϕ^{-1} and ψ^{-1} to give $\psi^{-1} \circ \phi^{-1}$. Or, diagrammatically this is represented by:



The third morphism we can define corresponds to the **parallel composition** of circuits. In a similar manner to sequential composition, it takes two other morphisms as arguments, and places them such that they are in parallel with one-another. Unlike for sequential composition, the two sub-circuits need not be of the same arity, and it is this parallel composition that can be thought of as the tensor product ($\otimes$) of the underlying monoidal structure. The arity of the new circuit that has been constructed is simply the sum of the arities of the underlying sub-circuits. So, we can define parallel composition as, given $\psi \in \mathbf{FxC}^{\simeq}a$ and $\phi \in \mathbf{FxC}^{\simeq}b$ we can construct $\psi{\otimes}\phi \in \mathbf{FxC}^{\simeq}(a{\otimes}b)$. The diagrammatic representation is as follows:



The inverse is again constructed from the inverses of the underlying sub-circuits, ψ^{-1} and ϕ^{-1}, and is given by $\psi^{-1} \otimes \phi^{-1}$. As a diagram this would be:



The underlying identity object of the monoidal structure relates to the empty circuit, that is, a circuit with zero inputs, and zero outputs. Parallel composition as described here, along with this identity object obey all the monoid laws as expected. (We talk more about monoids later, in section 4.3.1).

Our fourth set of morphisms correspond to circuits of arity 1, or more generally as operations that act on a single “bit”. It is the operations we define here that can restrict our categorical model to classical reversible circuits, or allow us to model fully quantum circuits. These **rotations** are therefore any elements of $\mathbf{FxC}^{\simeq}1$, and are given diagrammatically as single “bit” gates that are labelled by their underlying single “bit” operation. We’ll see later that in the classical case the only rotation is the Not rotation, which corresponds to logical negation, which is its own inverse. (E.g. $\neg \in \mathbf{FxC}^{\simeq}1$, and $\neg^{-1} = \neg$). Using this Not operation as an example we would have the diagram

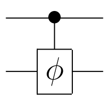
$$-\boxed{Not}-$$

In the quantum case, we would have members of $\mathbf{FxC}^{\simeq}1$ that represent any single qubit rotation (i.e. a unitary operation in $U(2)$). Usually we would represent a single qubit rotation as a unitary two by two complex valued matrix, and the inverse rotation would be the conjugate transpose of that matrix. For example, we could define the corresponding quantum Not rotation by

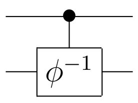
$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

and use the same diagram as for the classical case. It is also the case here that the conjugate transpose (or inverse) of the Not rotation is itself, but this is not necessarily the case in general.

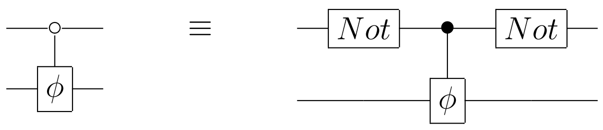
Our fifth and final morphism corresponds to a control structure in a circuit. Or more generally, a **conditional** gate that uses a control wire to decide whether the given argument computation should be performed. In $\mathbf{FxC}^{\simeq}$, this can be defined as, given $\phi \in \mathbf{FxC}^{\simeq}a$ we can construct $\phi \mid id_a \in \mathbf{FxC}^{\simeq}(\mathbb{N}_2 \otimes a)$, or using our natural number representation of objects $\phi \mid id_a \in \mathbf{FxC}^{\simeq}(1 + a)$. That is, given a circuit of arity a, we construct a conditional circuit of arity $1 + a$, whereby the extra wire is the control wire of the conditional. A typical circuit diagram for a conditional would be:



The conditional can be thought of such that the circuit $\phi \in \mathbf{FxC}^{\simeq}a$ is only applied if the value in the control wire is true (with the identity circuit id_a being applied if the control wire is false). The inverse of the conditional morphism is once again given by the inverse of the underlying sub-circuit. Thus given ϕ^{-1} we can construct the conditional $\phi^{-1} \mid id_a$, which is the inverse of the original conditional. Again, we can give this diagrammatically:

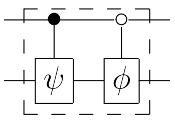


The structure of the conditional morphism gives rise to a naturally symmetric operation, namely the conditional which applies its argument circuit when the control wire is in the false state (as compared to the true state for the conditional defined above). For ease of notation, we shall also introduce this symmetric conditional, which is defined in terms of the conditional given above, along with a pre- and post-application of the Not rotation on the control wire. The definition is given diagrammatically as:

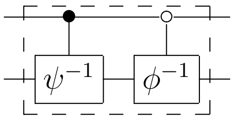


In $\mathbf{FxC}^{\simeq}$ we shall denote such a negated conditional acting on the sub-circuit $\phi \in \mathbf{FxC}^{\simeq}a$ as $id_a \mid \phi \in \mathbf{FxC}^{\simeq}(1 + a)$.

This naturally leads us to a choice operator, such that given two computations of the same size the value of the control wire is used to govern which sub-morphism is applied. That is, given $\psi, \phi \in \mathbf{FxC}^{\simeq}a$ we can construct the choice operator $\psi \mid \phi \in \mathbf{FxC}^{\simeq}(\mathbb{N}_2 \otimes a)$ as the following circuit:



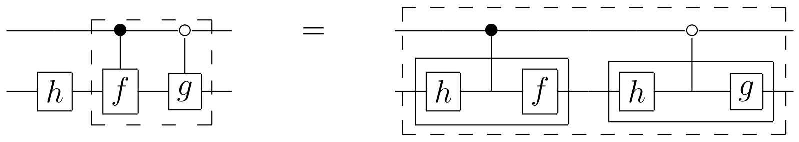
the inverse is once again given by ψ^{-1} and ϕ^{-1}, and constructed as $\psi^{-1} \mid \phi^{-1}$:



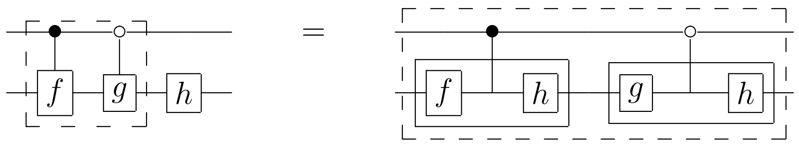
3.1.3 Equalities and laws in $\mathbf{FxC}^{\simeq}$

The underlying categorical infrastructure of $\mathbf{FxC}^{\simeq}$ gives us our standard laws for wires, sequential composition and parallel composition. However, we would like to also introduce some equalities that hold for our conditional morphisms, or more specifically how the conditional operation can be distributed over the sequential and parallel composition operations. We shall give the definitions of these equalities in terms of our diagrammatic model.

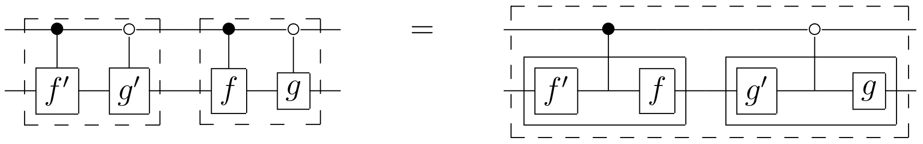
Firstly, we have the equality that $\circ$ distributes over $\mid$ or more precisely, that for $f, g, h \in \mathbf{FxC}^{\simeq}a$ we have $(f \mid g) \circ (id_1 \otimes h) = f \circ h \mid g \circ h$. Diagrammatically this can be shown as:



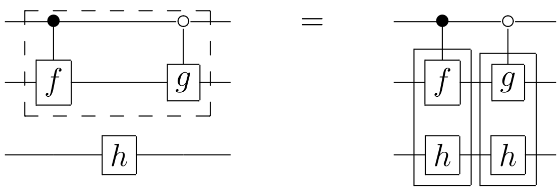
Secondly, we have the opposite distribution equality for $\circ$ and $\mid$. Such that given $f, g, h \in \mathbf{FxC}^{\simeq}a$ we have $(id_1 \otimes h) \circ (f \mid g) = h \circ f \mid h \circ g$ and the corresponding diagram:



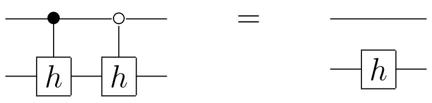
Our third equality defines how $|$ can distribute over $\circ$, giving for $f, f', g, g' \in$ $\mathbf{FxC}^{\simeq}a$ that $(f \mid g) \circ (f' \mid g') = (f \circ f') \mid (g \circ g')$.



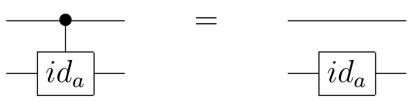
We can also give an equality for distributivity over $\otimes$ and $|$, such that given $f, g \in \mathbf{FxC}^{\simeq}a$ and $h \in \mathbf{FxC}^{\simeq}b$ we have that $(f \mid g) \otimes h = (f \otimes h) \mid (g \otimes h)$.



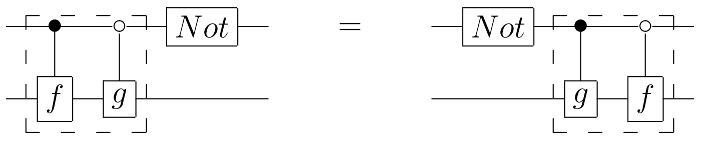
This last equality allows us to combine the first two such that we have $(h \mid h) = (id_1 \otimes h)$



and we can simplify this last axiom for the specific case that h is in fact the identity id_a. More precisely, we have that $id_a \mid id_a = id_{1+a}$, and we can give the diagram (in it's most simple form) as:



Our last equality axiom gives a symmetry relation for $|$, whereby moving a Not gate along the control wire over a choice operator can be accommodated by flipping the arguments to the choice operator. More precisely, this is defined that for $f, g \in \mathbf{FxC}^{\simeq}a$ we have $(\neg \otimes id_a) \circ (f \mid g) = (g \mid f) \circ (\neg \otimes id_a)$, and the corresponding diagram:



3.1.4 Examples of $\mathbf{FxC}^{\simeq}$ categories

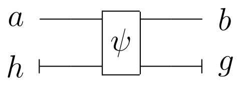
We mentioned briefly how there is more than one instance of an $\mathbf{FxC}^{\simeq}$ category, in the sense that the category depends upon a notion of the available 1 "bit" operations. The two main instances of $\mathbf{FxC}^{\simeq}$ that we have thought about are the instances that relate to classical reversible computation and quantum computation. The classical instance of $\mathbf{FxC}^{\simeq}$ we have dubbed $\mathbf{FCC}^{\simeq}$, and is the category of finite classical reversible circuits. The quantum instance of $\mathbf{FxC}^{\simeq}$, we have dubbed $\mathbf{FQC}^{\simeq}$, and is the category of finite quantum circuits.

In $\mathbf{FCC}^{\simeq}$ we only have two 1 bit operators (here we use bit in its normal meaning of binary digit), namely the logical negation operation and the identity operation. It's useful to note that the identity operation on 1 bit is not defined as a rotation as it can be derived directly as id_1, or just a single wire. Extensional equality in $\mathbf{FCC}^{\simeq}$ can be attained by looking at circuits as permutations on the initial segment $[a]$, or more simply by looking at the truth tables corresponding to the circuits. In $\mathbf{FQC}^{\simeq}$ we have an infinite number of rotations, which correspond to all the possible single qubit unitary operators. Extensional equality in $\mathbf{FQC}^{\simeq}$ is achieved by looking at the circuits as unitary operators on an a-dimensional Hilbert space. These can be a lot harder to calculate and compare than in the classical case, so it's nice to note that that $\mathbf{FCC}^{\simeq} \hookrightarrow \mathbf{FQC}^{\simeq}$ and this embedding preserves the extensional equality. This can easily be checked by looking at the unitary operators that can be obtained only from the rotations that represent the available classical 1 "bit" operations, and noting that they only contain 0s and 1s in their matrix representation. Such matrices in effect only define permutations, in the same way as we're interpreting extensional equality in $\mathbf{FCC}^{\simeq}$.

3.2 Generalised irreversible computation

Having defined a category of reversible circuits, I shall now go on to look at how we can derive the notion of irreversible computations from our notion of reversible circuits. In fact, we shall define another category **FxC** of (possibly) irreversible computations. Every morphism of the category **FxC** will indeed represent an irreversible computation, although it will actually take the form of a triple, $\psi' = (h, g, \psi)$, whereby h is a set of heap inputs, g is a set of garbage outputs, and ψ is the underlying reversible computation. So, a morphism in $\mathbf{FxC}(a, b)$ can be given as a morphism in $\mathbf{FxC}^{\simeq}((a \otimes h), (b \otimes g))$ with the requirement that $(a \otimes h) = (b \otimes g)$. Indeed, this model of irreversible computation is used in defining QML [AG05].

If we have an irreversible computation, or more precisely, a morphism $(h, g, \psi) \in \mathbf{FxC}$, we can extend our diagrammatic model of the underlying reversible computation $\psi \in \mathbf{FxC}^{\simeq}$, to a diagrammatic model representing the irreversible computation by explicitly marking the heap inputs, and the garbage outputs as such.

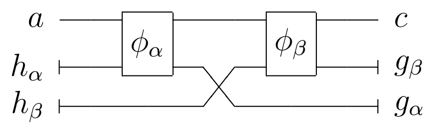


We can also see that any circuit in $\mathbf{FxC}^{\simeq}$ has an equivalent circuit in **FxC**, such that the set of heap inputs is empty, and the set of garbage outputs is also empty. More formally, we shall denote this as a lifting by having, for any $\psi \in \mathbf{FxC}^{\simeq} a$ a lifted version $\widehat{\psi} \in \mathbf{FxC}(a, a)$, with $\widehat{\psi} = (\emptyset, \emptyset, \psi)$. This could also be given as the following predicate:

$$\frac{\psi \in \mathbf{FxC}^{\simeq} a}{\widehat{\psi} \in \mathbf{FxC}(a, a)}$$

3.2.1 Morphisms in FxC

We have seen that the irreversible computations in the category $\mathbf{FxC}$ are defined as circuits in the $\mathbf{FxC}^{\simeq}$ category, along with a set of heap inputs, and a set of garbage outputs. The underlying circuits of these irreversible computations are constructed as morphisms in the $\mathbf{FxC}^{\simeq}$ category, and as such can use all the morphism described for $\mathbf{FxC}^{\simeq}$. However, we are also able to define some morphisms that exist specifically for the computations in $\mathbf{FxC}$. Firstly, we can define sequential composition of irreversible computations. If we are given irreversible computations $\alpha = (h_\alpha, g_\alpha, \phi_\alpha) \in \mathbf{FxC}(a, b)$, and $\beta = (h_\beta, g_\beta, \phi_\beta) \in \mathbf{FxC}(b, c)$, then we can define $\beta \circ \alpha \in \mathbf{FxC}(a, c)$ as:

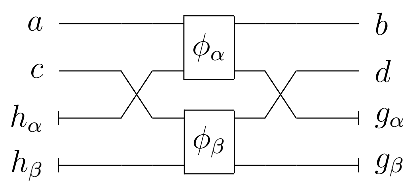


More precisely, two irreversible computations can be composed in sequence if the number of outputs from the first computation is equal to the number of inputs to the second computation. The computations are combined by *wiring up* said outputs to said inputs. The heap inputs to the second computation must also be thread through from the start of the computation, and the garbage outputs from the first computation must be thread through to the end of the computation.

The special case of wires that represents the identities in $\mathbf{FxC}^{\simeq}$ can be directly lifted to represent the equivalent identities in $\mathbf{FxC}$, or more specifically $id_a^{\mathbf{FxC}} = \widehat{id_a^{\mathbf{FxC}^{\simeq}}}$.

We can also lift the monoidal structure of the underlying $\mathbf{FxC}^{\simeq}$ circuits to give $\mathbf{FxC}$ an equivalent monoidal structure. This lifting allows us to define a form of parallel composition for irreversible computations in $\mathbf{FxC}$. First, we can simply lift the neutral element of the tensor, the empty circuit, by using our lifting operation previously defined $I^{\mathbf{FxC}} = \widehat{I^{\mathbf{FxC}^{\simeq}}}$. The definition of $\otimes$ is also inherited from the underlying $\mathbf{FxC}^{\simeq}$ category, and can be defined such that given $\alpha = (h_\alpha, g_\alpha, \phi_\alpha) \in \mathbf{FxC}(a, b)$ and $\beta = (h_\beta, g_\beta, \phi_\beta) \in \mathbf{FxC}(c, d)$, we can have

$\alpha \otimes \beta \in \mathbf{FxC}(a \otimes c, b \otimes d)$ as:



Again, rewirings are used to thread the heap inputs, and garbage outputs so that they are grouped together at the beginning, and end, of the computation respectively.

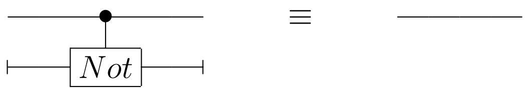
3.2.2 Examples of FxC categories

As for the $\mathbf{FxC}^{\simeq}$ category of reversible circuits, the two examples of **FxC** categories correspond to classical and quantum computation respectively. In fact, we are just able to extend our two examples, $\mathbf{FCC}^{\simeq}$ and $\mathbf{FQC}^{\simeq}$, from above. In the classical case we are able to extend upon the category $\mathbf{FCC}^{\simeq}$ of reversible circuits to give us the category **FCC** of finite classical computations. In the quantum case, we extend on the category $\mathbf{FQC}^{\simeq}$ of quantum circuits to give us the category **FQC** of finite quantum computations. Extending upon our definitions for extensional equality for the $\mathbf{FxC}^{\simeq}$ categories, we can easily derive a notion of extensional equality for the classical case. Finite classical computations (or the morphisms) in $\mathbf{FCC}^{\simeq}$ can be thought of as functions acting upon finite sets. For the computation $(h, g, \phi) \in \mathbf{FCC}(a, b)$, we can define $(0^h, -) \in [a] \to [a \otimes h]$ as an initialisation function that initialises the necessary number of heap inputs, and we can define $\pi_g \in [b \otimes g] \to b$ as a projection function that projects out the garbage, returning only the b results as defined by the computation. The whole computation can then be interpreted as $\pi_g \circ [\![\phi]\!] \circ (0^h, -) \in [a] \to [b]$. So, in other words, the equality of the underlying circuit ϕ, with respect to only the non-heap inputs, and the non-garbage outputs, can be used.

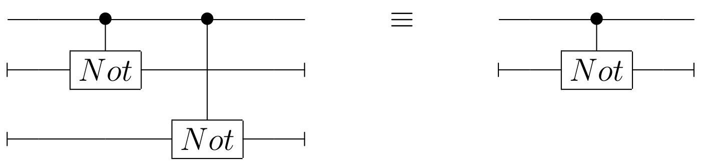
Extensional equality in the quantum case, **FQC**, raises a few more questions. We are able to interpret our finite quantum computations as superoperators (see

[Sel04, VAS06], or [Gra06] for an implementation in Haskell). Superoperators are morphisms on density operators, which are positive operators on the a-dimensional Hilbert space. A superoperator $f \in \mathbf{Super}(a, b)$ is a linear function mapping density operators on a to density operators on b, which preserve the trace and are stable under $\otimes$. Analogously to the classical case, we interpret $(h, g, \phi) \in \mathbf{FQC}(a, b)$ as $\mathrm{tr}_g \circ [\![\phi]\!] \circ 0^h \otimes - \in \mathbf{Super}(a, b)$, where $[\![\phi]\!] \in \mathbf{Super}(h \otimes a, g \otimes b)$ is the superoperator associated to the unitary operator given by interpreting the reversible circuit ϕ. $0^h \otimes - \in \mathbf{Super}(a, a \otimes h)$ initialises the heap and $\mathrm{tr}_g \in \mathbf{Super}(g \otimes b, b)$ is a partial trace which traces out the garbage.

The equality we have used in the classical case doesn't just lift over this definition of quantum computations. There are cases in which the garbage may have become entangled with the output qubits, and the tracing out of the garbage is to all intents and purposes the same as measuring it (see [NC00] p187). For example, in the classical case the following two circuits would be equivalent:



However, this equivalence does not hold when we move into the category of finite quantum computations **FQC**. This is because in quantum computation the control wire (or qubit) can become entangled with the target wire (qubit). For example, think of the case when the input is in the state $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ (the second input is in the state $|0\rangle$ by the definition of a heap). The controlled Not operation leaves the overall state as $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$, and measurement of the garbage leaves the output qubit in one of the states $|0\rangle$ or $|1\rangle$, each with probability $\frac{1}{2}$. Neither of these states being the same as the input state. As it stands, this doesn't cause a problem in our model, as we have no rules that would be able to prove this equivalence. However, there are other forms of equivalence that do hold in both **FCC** and **FQC**, and we would like to look at how we can start to define these forms of equivalence. The approach we take is to define some equivalence rules that do hold in **FQC** (and hence **FCC**), and show how these can be used to prove certain *larger* equivalences that we know do hold for quantum computations. The example we give here is an instance of von Neumann's measurement postulate.



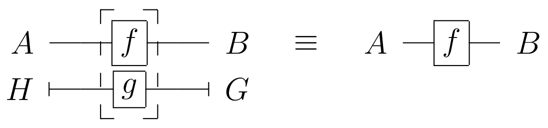
The next section goes on to look at three laws we have developed that can be used to prove such an equivalence, along with a description of just such a proof.

3.3 Three equivalence laws for FxC categories

We start this section by introducing the three laws we have developed that are used to try and define the equivalences that hold between computations in both the **FCC** and **FQC** categories. The first law is dubbed "The law of garbage collection", the second law is dubbed "The uselessness of garbage processing", and the third law is dubbed "The uselessness of heap pre-processing". There is somewhat of a symmetry between the last two laws, but we'll go on to see how a side condition of the third law takes away a lot of the elegance of this symmetric relationship. The next three subsections present the laws, along with their diagrammatic representations.

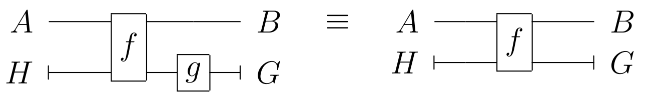
3.3.1 The law of garbage collection

If a circuit can be reduced into two smaller circuits such that one part of the circuit only acts on heap inputs and on garbage outputs, then that part of the circuit can be removed.

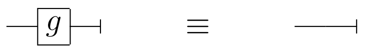


3.3.2 The uselessness of garbage processing

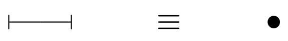
If a circuit can be reduced into two smaller circuits such that one part of the circuit only has an effect on garbage outputs, then that part of the circuit can be removed.



This can be alternately stated as saying that if the only outputs of (part of) a circuit are garbage outputs, then this is equivalent to just having garbage.

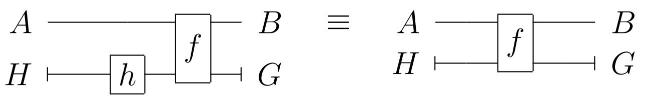


This second law can be used to simplify the first law so that a wire that only connects the heap to the garbage is equivalent to having nothing (or an empty computation).

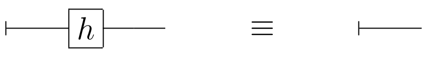


3.3.3 The uselessness of heap pre-processing

If a circuit can be reduced into two smaller circuits such that one part of the circuit only has effect on heap inputs, and the effect on the zero vector is the identity, then that part can be removed.
$if \quad h\vec{0} = \vec{0} \quad then$



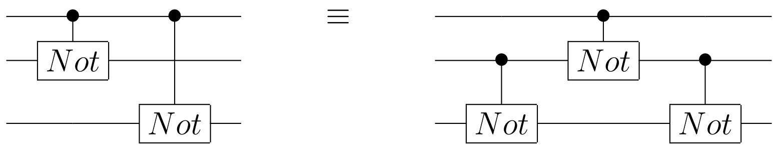
An alternate notation for this would again be to state that if (part of) a circuit only has heap inputs, and its effect on the zero vector is the identity, then this is equivalent to just having a heap.
$if \quad h\vec{0} = \vec{0} \quad then$



3.4 Using the three laws: A proof of the measurement postulate

As has been previously mentioned, we can use our three laws to prove the measurement postulate as given at the end of section 3.2.2. This proof can be given in a diagrammatic fashion as follows.

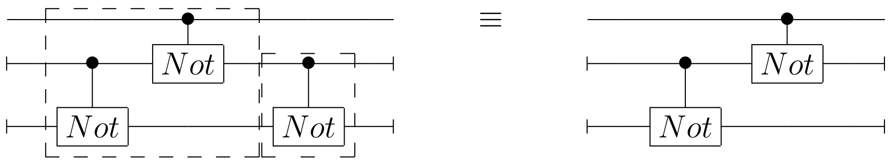
As the embedding of **FQC** in **FQC**$^{\simeq}$ is full and faithful, We can use the underlying equality from the **FQC**$^{\simeq}$ category to substitute in the following circuit (as the underlying circuit of the computation):



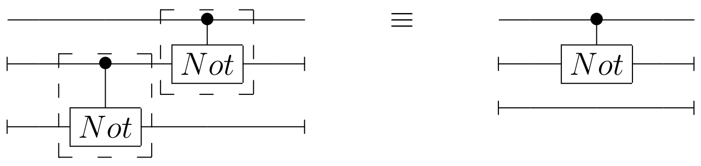
We can use the underlying equivalences, as the lack of heap and garbage shows that these are just reversible circuits from **FQC**$^{\simeq}$. The classical nature of all the morphisms in the circuit mean that we can inherit the equivalences for these circuits from the simpler category **FCC**$^{\simeq}$, and a quick check of the corresponding truth tables suffices as the equivalence proof. Namely, both these circuits have the same truth table as shown below (labelling $q0, q1, q2$ as the top, middle, and bottom qubits respectively).

$q0_{in}$	$q1_{in}$	$q2_{in}$	$q0_{out}$	$q1_{out}$	$q2_{out}$
0	0	0	0	0	0
0	0	1	0	0	1
0	1	0	0	1	0
0	1	1	0	1	1
1	0	0	1	1	1
1	0	1	1	1	0
1	1	0	1	0	1
1	1	1	1	0	0

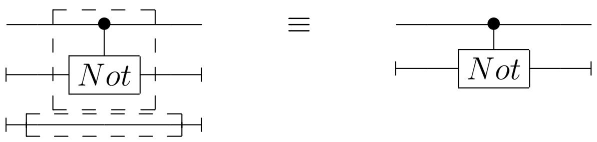
Moving back into the **FQC** category, we simply keep the set of heaps and garbages the same as they were. The next step of the proof is to remove the third controlled *Not* operation using the second of our laws.



The next step of our proof uses the fact that the controlled *Not* preserves the zero vector, that is, if we apply the controlled *Not* to the two-qubit state $|00\rangle$, we are left with the same $|00\rangle$ state. This allows us to remove the first controlled *Not* by use of our third law.



The last step of the proof is to simply remove the bottom wire, which simply connects a heap input to a garbage output, by use of our first law.



3.5 Remarks on the three laws

It has been shown in the previous section how our three laws governing equivalences in the **FQC** category of finite quantum computations can be used to give a proof of the measurement postulate. However, it has not be shown whether all equivalent computations in **FQC** can be shown to be equivalent by means of the three laws alone. This is indeed still an open question. The work shown here has been extended on in [YY09] using functoriality of the choice and parallel composition morphisms to deduce equivalence properties instead of the inductive approach used above. It has also been remarked that the categorical structures we have defined here can be reformulated in terms of Selinger's dagger-complete categories [Sel07], which could lead to a more complete framework of equivalence relations.

Chapter 4

Functional Programming

4.1 An introduction to functional programming

Purely functional programming languages are languages that treat computations as the evaluation of pure mathematical functions. Functions are said to be pure if they always return the same result when given the same inputs, and if the application of the function on any of its inputs will not have caused a side-effect within any other part of the system. The functional language used in this thesis is Haskell, and the next few sections shall give a brief overview of the language and some of its features. One of the nice aspects of using a purely functional language is how it deals with side-effects. Section 4.3 will go into more details, but suffice to say here that with the current definition of pure functions, it could be argued that any program that cannot have any side-effects within the system, equates to a program that cannot have any input or output as these relate to changes in an overall system state. Haskell makes use of the categorical notion of monads to explicitly deal with these side-effects, and to give a more precise definition of what side-effects are able to occur. Section 4.3 will also introduce the notion of Monads in Haskell, and section 4.5 will go over the specific example of the IO Monad that is used to deal with I/O in Haskell programs. Another useful aspect of many functional languages, including Haskell, is how they make use of lazy evaluation, whereby function evaluation only takes place as and when required. This is useful in many ways as it allows the use of infinite data structures, whereby only a finite amount of it is required, and can also save time by not evaluating sub-expressions if these results are not required to evaluate the overall expression.

It is the way that Haskell utilises monads to explicitly deal with side-effects that has been the main motivating factor for using it to develop an interface to quantum computation. We'll see in Chapter 5 that we are able to define a monad in Haskell that is used to explicitly deal with the side-effects present in quantum computation. In other (imperative) languages, where side-effects are implicit at even the lowest level (e.g. updating stored values), this design would not be possible. Having the side-effects of measurement explicitly dealt with by a monadic structure also means we are able to define our unitary transformations separately from this monadic structure, in essence allowing us to design our reversible computations away from the complications of measurement.

4.2 Haskell - A purely functional programming language

Haskell, is a purely functional programming language. It has been around and developed since 1990, and has a large and growing user base. It relies heavily on pattern matching and currying, allowing for a more natural, mathematically based, way of defining functions. There are many compilers and interpreters available for Haskell, although GHC (The Glasgow Haskell Compiler) is fast becoming the most commonly used because of it's large library of extensions. The compilers, along with the Haskell language definition [Jon03], can be found on the Haskell homepage: `www.haskell.org`. The Haskell homepage also contains descriptions of all the standard libraries, along with tutorials and much more information on the language itself. There are also a number of introductory textbooks available, e.g. [Hut07].

In this section we shall give a brief introduction to Haskell, starting with an introductory example that is often given as a first programming exercise in functional programming languages. Namely, defining a function that calculates the factorial of the given input. Firstly, it is useful (although not always necessary) to give the type of the function you wish to define. In this case, the factorial function is a function from an integer to another integer (*Int* being a 32 bit integer in Haskell).

$$\mathit{factorial} :: \mathit{Int} \rightarrow \mathit{Int}$$

We can now make use of pattern matching to split the function definition into its constituent parts, firstly the base case for when the input is zero (the output is one), and secondly the recursive case for when the input (n) is greater than zero (the output is n multiplied by the factorial of n-1). In Haskell we can define this as:

$$\mathit{factorial}\ 0 = 1$$
$$\mathit{factorial}\ n = n * \mathit{factorial}\ (n - 1)$$

The factorial function is a nice introductory example as it shows off some important techniques that are extensively used in Haskell. The use of pattern matching allows function definitions to be split depending on the input. When evaluating a function that is defined using pattern matching, Haskell will look sequentially through each definition until the input matches the given pattern, in the case of the *factorial* function the second line will be evaluated unless the input is 0. The other major technique employed by the definition of the *factorial* function above is its use of recursion. That is the function is able to call itself. Mostly, the recursive call will need to be over a *smaller* input than the original call, although it is the job of the programmer to ensure this is the case (to prevent non-terminating programs). In this example it is quite clear that the recursive call is over a *smaller* argument, as $n - 1$ is smaller than n. However, the definition of smaller isn't necessarily as clear cut as in this case, and indeed the function definition as it stands would be non-terminating for a negative input.

Another data-type that is extensively used in Haskell is the list datatype. A list in Haskell is a possibly infinite list of elements with the same type. For example a list of integers would have the type $[\mathit{Int}]$, and we can now write functions acting on these lists. For example we could write a function that reverses a list of integers, starting again by defining the type of the function, and then using pattern matching over the two list patterns, $[\,]$ which matches with the empty list, and $(x : xs)$ which matches with a non-empty list with head element x and a (possibly empty) tail xs.

$\mathit{reverseList} :: [\mathit{Int}] \rightarrow [\mathit{Int}]$

$\mathit{reverseList}\ [\,] = [\,]$

$\mathit{reverseList}\ (x : xs) = \mathit{reverseList}\ xs +\!\!+ [x]$

From this we can see that the reverse of an empty list is an empty list, and the reverse of a non-empty list is the reverse of the tail of the list concatenated onto the front of the singleton list containing the head element. If we look more closely at the definition of the *reverseList* function above, we'll also notice that we haven't needed to look at the values of the individual elements of the list. Indeed, this is often the case with list operations, and Haskell provides us with a more generic way of defining operations on lists so that we don't have to rewrite the function for every possible type of list. We can define a more generic list reverse function whereby the type a is any *arbitrary* type.

$\mathit{reverseList}' :: [a] \rightarrow [a]$

$\mathit{reverseList}'\ [\,] = [\,]$

$\mathit{reverseList}'\ (x : xs) = \mathit{reverseList}'\ xs +\!\!+ [x]$

This $\mathit{reverseList}'$ function is able to reverse a list containing elements of any type, and the type checker is able to infer at runtime which specific instance for a is being used. However, it is important to remember that if we need to look at the value of any of the elements of this list this generic definition may no longer be correct. For example, if we were writing a function to sort a list we would need to know that the elements had some sort of ordering. We shall look at how Haskell uses type-classes to deal with this separate kind of generic programming in section 4.4, but for now we shall look at the specific example of sorting a list of integers.

As lists are used quite extensively in Haskell, we are provided with a form of list comprehension. This simple sort function for a list of integers can be thought of as creating a list of elements less than the head of the original list, and a list of elements greater than the head of the original list, which are both then sorted themselves before being concatenated back onto either side of the head of the original list. We make use of the **where** keyword to give variable names to the two parts of the list we are sorting (although they are both still immutable data).

$$\begin{aligned}
&\mathit{sortList} :: [\mathit{Int}] \to [\mathit{Int}] \\
&\mathit{sortList}\ [\,] = [\,] \\
&\mathit{sortList}\ (x : \mathit{xs}) = \mathit{ltx} +\!\!+ [x] +\!\!+ \mathit{gtx} \\
&\qquad\qquad\qquad\qquad \textbf{where}\ \mathit{ltx} = \mathit{sortList}\ [y \mid y \leftarrow \mathit{xs}, y \leqslant x] \\
&\qquad\qquad\qquad\qquad\qquad\quad \mathit{gtx} = \mathit{sortList}\ [y \mid y \leftarrow \mathit{xs}, y > x]
\end{aligned}$$

The code $[y \mid y \leftarrow xs, y \leqslant x]$ can be read as creating a list of all elements y such that y comes from the list xs and y is less than or equal to the element x. Similarly for *gtx* ($[y \mid y \leftarrow xs, y > x]$) we have the list of all elements y such that y comes from the list xs, and y is greater than the element x. The recursive calls to the *sortList* function will sort these two sublists before they are recombined with the original head element x. List comprehension can be a very useful tool when creating functions over lists in Haskell. The *sortList* function is in fact an implementation of the Quicksort algorithm. As previously mentioned we shall look at how a more generic list sorting algorithm can be defined using type-classes in section 4.4.

All the functions defined so far have been pretty simple introductory examples to try and give the reader a taste for how programs are written in Haskell. The next few sections shall introduce more programming constructs which are used later in this thesis for the implementation of the Quantum IO Monad, with section 4.6 having some more relevant examples to compare with the quantum computations written using the QIO Monad in chapter 6. One last example I shall give here is to introduce the idea of an accumulator function. Accumulator functions can often be used to implement more efficient functions acting on lists, and the simplest example of an accumulator function is to re-write our list reversing algorithm using one.

$$
\begin{aligned}
&\mathit{reverseListAccumulator} :: [\,a\,] \to [\,a\,] \to [\,a\,]\\
&\mathit{reverseListAccumulator}\ \mathit{xs}\ [\,] = \mathit{xs}\\
&\mathit{reverseListAccumulator}\ \mathit{xs}\ (y : \mathit{ys}) = \mathit{reverseListAccumulator}\ (y : \mathit{xs})\ \mathit{ys}\\
&\mathit{reverseList}'' :: [\,a\,] \to [\,a\,]\\
&\mathit{reverseList}'' = \mathit{reverseListAccumulator}\ [\,]
\end{aligned}
$$

Accumulators are a form of higher order function that use an extra data-structure to enable the result to be accumulated rather than constructed explicitly. As we can see, the accumulator function itself takes two lists as arguments, and in essence shifts the head of one list onto the head of the other list creating a stack like structure whereby every time the head is popped off the top of one list, it is then pushed onto the top of the other. It is this process that reverses the list when the function is called with the empty list as its first argument. The reason this function is more efficient than the previous implementation of the $\mathit{reverseList}'$ function comes from the way that the concatenation operation on lists is defined. In the accumulator example above, the original input list is only traversed once, whereas in the original implementation the list is traversed for every call of the concatenation function ($+\!\!+$), which is called on each recursive call of the overall function.

4.3 Effects in a purely functional setting

As we have mentioned previously, pure functions are defined as functions that do not produce any side effects. However, in purely functional languages we are still able to define effectful computations by using datatypes that describe the side-effects that can take place. These datatypes come in the form of Monads which encapsulate all the effectful parts of the computation so that the overall program is still pure. In Haskell, monads are used extensively to create any sort of computations that in themselves would not be pure. Indeed, all I/O in Haskell takes place in the IO Monad, which we shall introduce in section 4.5. It is easy to see that I/O actions are impure just by looking at the simple function *getChar* which reads a character from the standard input. By definition, a function is only pure if it always returns the same result given the same starting state, but the whole point of the *getChar* function is to return a different character depending on what the user has entered. This is an event that occurs after the function has been called, so cannot be encoded into some sort of input state to the function and hence is also why the *getChar* function doesn't have any arguments.

As monads are a notion from category theory, where they are in essence an instance of a specific monoid, it is worth looking first at the notion of a monoid.

4.3.1 Monoids

Monoids can be defined as a set S along with a binary operation ($\bullet$) that acts on members of the given set. The binary operation must be closed over the set, and there must be a member of the set that is a (left and right) identity element for the binary operation. The binary operation must also be associative.

Two very simple examples of monoids are of the natural numbers $\mathbf{n}$, along with addition ($+$), which has the identity element zero, or the natural numbers along with multiplication ($*$), which has the identity element one.

In functional programming, we can define monoids in a very similar way. A monoid is a data-type (a), along with a binary operation acting on that data-type ($f :: a \to a \to a$), and an element of the data-type ($id :: a$) that is the identity of the given binary operation. In Haskell, the function f is usually called *mappend*, and the identity element *id* is known as *mempty*. Section 4.4.1 gives more information on Monoids in Haskell, and goes on to give an example of how we can define a monoid in Haskell for sequencing computations.

4.3.2 Monads

Monads are a notion from category theory ([Mac71]), and are a generalisation of monoids. Moreover, a monad over a category $\mathbf{C}$, is given by a functor $T : \mathbf{C} \to \mathbf{C}$, along with two natural transformations $\eta : \mathbf{1_C} \to T$, and $\mu : T^2 \to T$. There are two axioms which these natural transformations must adhere to, which correspond to the generalisation of the associativity law of monoids, and the existence of an identity element. These two axioms can be given respectively as the following commutative diagrams.

$$\begin{tikzcd}
T^3A \arrow[r, "\mu TA"] \arrow[d, "T\mu A"'] & T^2A \arrow[d, "\mu_a"] \\
T^2A \arrow[r, "\mu A"'] & TA
\end{tikzcd}
\qquad
\begin{tikzcd}
TA \arrow[r, "\eta TA"] \arrow[dr, "id_{TA}"'] & T^2A \arrow[d, "\mu A"] & TA \arrow[l, "T\eta A"'] \arrow[dl, "id_{TA}"] \\
 & TA &
\end{tikzcd}$$

The definition of a monad, as given above, can be given equivalently in the form of a Kleisli triple [Mog88]. These Kleisli triples correspond nicely to how we use monads in a computational setting, but don't show as easily the correspondence to monoids. We shall look now at how we can define monads in terms of a Kleisli triple, and how this relates to functional programming.

A Kleisli triple over a category $\mathbf{C}$ is defined by a triple $(T, \eta, \_^*)$, with $T : \mathit{Obj}(\mathbf{C}) \to \mathit{Obj}(\mathbf{C})$, or in terms of functional programming we can think of this as the monadic type-constructor M, whereby any underlying type a has a corresponding monadic type $M\ a$. $\eta_A : A \to TA$ for $A \in \mathit{Obj}(\mathbf{C})$, corresponds to an embedding of a member of an underlying type into the corresponding monadic type. In Haskell, we denote this function by $\mathit{return} :: a \to M\ a$, so for example if we have a value $v :: a$ then we have a corresponding value embedded in the monadic type, $\mathit{return}\ v :: M\ a$. Finally, $\_^*$ takes a function $f : A \to MB$, and lifts it to the function $f^* : MA \to MB$, or again in functional programming terms, lifts a function from the type a to a monadic type $M\ b$, such that it can be applied to a member of the corresponding monadic type $M\ a$. In other words, the application of the given function can be bound to a result in the monadic type. This *binding* leads to this function being called the bind function in Haskell, and is defined by $(\gg=) :: M\ a \rightarrow (a \rightarrow M\ b) \rightarrow M\ b$. There are again some laws which define how these functions must behave (corresponding to the monadic laws given above), and so as to move away from the categorical notions, we shall give these laws in terms of their corresponding Haskell notations.

Firstly, we should have that the *return* function corresponds to a left and right identity for the monad. The left identity is given by the equivalence $\mathit{return}\ a \gg= f \equiv f\ a$. That is, that binding a function (f) to the result of embedding a value from the underlying type ($\mathit{return}\ a$) should be the same as just applying the unbound function to the value in the underlying type ($f\ a$). The right identity, is given by the equivalence $m \gg= \mathit{return} \equiv m$. That is, that binding the return function to a member of the monadic type ($m :: M\ a$) should just give the same member of the monadic type ($m :: M\ a$). Secondly, the bind operation must be associative, and this law is given in Haskell code by the equivalence $(m \gg= f) \gg= g \equiv m \gg= (\lambda x \rightarrow f\ x \gg= g)$. Section 4.4.1 shall go on to look at monads in Haskell in more detail, and we shall give some simple examples of what can be achieved using monads. In section 4.5 we shall go on to see how important monads are in Haskell, and the special syntax that Haskell provides (known as 'do' notation) that makes the use of monads in Haskell even easier.

4.4 Type classes in Haskell

Type classes are used in Haskell so that more generic functions can be defined. As we have seen, functions in Haskell are of a given type, although some polymorphism is available, but what if it doesn't really matter what the type is, so long as there is a particular function that can work on that type. For example, we wrote a function that sorts a list of integers, but mentioned that it is actually possible to sort a list that contains any data-type that comes with an explicit ordering. In general terms, we need to be able to look at two elements of a data-type and decide whether one is *larger*, *smaller*, or the *same size* as the other. For integers we are able to use the functions $(\leqslant)$ to compare if one integer is "less than or equal" to another, and the function $(>)$ to compare if it is "greater than" another. These functions both have the type $\mathit{Int} \rightarrow \mathit{Int} \rightarrow \mathit{Bool}$ and it is the boolean value returned by these functions that is used by Haskell when it is performing a list comprehension. So if we know that we have equivalent operations for an arbitrary ordered data-type a, in $a \rightarrow a \rightarrow \mathit{Bool}$, then we know that we would be able to write a function that sorts a list over type a. Haskell allows us to do exactly this with its use of type classes. As an example, we could define a type class that states that for an arbitrary data-type a to be a member of the type class *Ord*, it must provide the function $(\leqslant) :: a \rightarrow a \rightarrow \mathit{Bool}$ and the function $(>) :: a \rightarrow a \rightarrow \mathit{Bool}$, is derived from this. In Haskell this would be defined by

$$\begin{array}{l} \textbf{class}\ \mathit{Ord}\ a\ \textbf{where} \\ \quad (\leqslant) :: a \rightarrow a \rightarrow \mathit{Bool} \\ \quad (>) :: a \rightarrow a \rightarrow \mathit{Bool} \\ \quad a > b = \neg\,(a <= * \, b) \end{array}$$

and then if we want to write a polymorphic sorting function for lists of any arbitrary type a that fulfils the *Ord* type class, all we have to do is to inform Haskell to check for this as follows:

$$\begin{array}{l} \mathit{sortList}' :: (\mathit{Ord}\ a) \Rightarrow [a] \rightarrow [a] \\ \mathit{sortList}'\ [\,] = [\,] \\ \mathit{sortList}'\ (x : xs) = \mathit{ltx} \mathbin{+\!\!+} [x] \mathbin{+\!\!+} \mathit{gtx} \\ \qquad\qquad\qquad \textbf{where}\ \mathit{ltx} = \mathit{sortList}'\ [y \mid y \leftarrow xs, y \leqslant x] \\ \qquad\qquad\qquad\qquad\quad \mathit{gtx} = \mathit{sortList}'\ [y \mid y \leftarrow xs, y > x] \end{array}$$

Now, all we have to do is create some instances of the type class to be able to use it. So, we know that we have the functions available for integers, so lets define integers as a instance of the *Ord* type class as follows:

```
instance Ord Int where
  (⩽) = (⩽)
```

There can be many data-types that fulfil a type class, for example we could also define an ordering for lists such that a list with more elements is deemed to be *greater* than a list with fewer elements:

```
instance Ord [a] where
  xs ⩽ ys = length xs ⩽ length ys
```

There are many other situations where type-classes come in useful, and Haskell provides many built in type-classes in its standard library. For instance, the *Ord* type-class we have defined above is very similar to a built in type-class (also called *Ord*) for data types with an ordering. Other commonly used type classes include, *Eq* for data-types which have a defined equality, *Num* for numeric types, which define many standard numeric operations (e.g. + - *), and *Show* which defines data-types that can be converted to a string so they can be displayed on the command line or in a Haskell interpreter. In some cases it is possible for Haskell to *derive* an instance of a type class for a specific data-type, which is often the case for the *Show* type class.

4.4.1 Monoids and Monads in Haskell

Monoids in Haskell are defined by a type class. We have seen that a monoid can be thought of as a data-type ($a :: *$) along with a binary operation ($mappend :: a \to a \to a$), and an element of a that is the (left and right) identity to the *mappend* operation ($mempty :: a$). This can easily be translated into a type-class.

```
class Monoid a where
  mempty  :: a
  mappend :: a → a → a
```

Any type that fulfils these two obligations can therefore be defined as a monoid, although it is useful to note that it is up to the programmer to ensure that the monoid laws hold.

A good example of monoids in Haskell is how they can be used to sequence computations, threading a sort of *state* through the computation. This can be achieved by defining a monoid over functions. We can define a stateful computation in terms of transition functions from one state to another. In Haskell we can define this as the data-type:

$\textbf{data}\ \mathit{State}\ s = \mathit{State}\ \{\mathit{runState} :: s \to s\}$

The destructor function *runState* is the inverse to the constructor (*State*), and is generated automatically satisfying the equation $\mathit{runState}\ (\mathit{State}\ f) = f$. We can then define the monoidal structure such that these transition functions can be composed in sequence. As a transition function is just a function in Haskell we are able to use functional composition ($\circ$) to sequence the different transition functions. The identity element of this monoid can then simply be given as the identity function, which just returns its argument as the result.

$\textbf{instance}\ \mathit{Monoid}\ (\mathit{State}\ s)\ \textbf{where}$

$\quad \mathit{mempty} = \mathit{State}\ \mathit{id}$

$\quad (\mathit{State}\ f)\ \text{`}\mathit{mappend}\text{`}\ (\mathit{State}\ g) = \mathit{State}\ (g \circ f)$

A stateful computation could now be defined in terms of these transition functions between states, but the monoidal structure doesn't allow us to look at the states during any intermediary stages of the computation. The whole computation must be evaluated over the input state before an output state is returned. We shall go on now to look at monads in Haskell, and shall return to this idea of stateful programming using monads, which give us a better way of sequencing computations such that intermediary results can be extracted.

Monads in Haskell are also defined by a type-class. We have seen how monads are defined by a type-constructor $m :: * \to *$, along with poly-morphic functions $\mathit{return} :: a \to m\ a$ and (the bind function) $(\gg\!=) :: m\ a \to (a \to m\ b) \to m\ b$. This definition can again be easily translated into a type-class.

$\textbf{class}\ \mathit{Monad}\ m\ \textbf{where}$

$\quad \mathit{return} :: a \to m\ a$

$(\ggg\!=) \;:: m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b$

The requirement that these definitions must uphold the monad laws is also left in the hands of the programmer to check.

Monads in Haskell can be thought of as a type-constructor whose members describe the monadic behaviour available. The simplest monad to first look at in Haskell is the Maybe monad. When defining a monad it is often best to look at the underlying type-constructor for that monad, and understand the behaviour that the monad is trying to define before looking at the monadic functions. For the Maybe monad, we start with the following type-constructor.

```
data Maybe a = Nothing
             | Just a
```

That is, a member of the *Maybe* type is either *Just* a value from the underlying type, or it is *Nothing*. The Maybe monad is often used when a computation might not give a sensible result, and in those cases it is the *Nothing* constructor that is used to describe a failure of the computation. For example, think of a computation that divides two integer arguments. What should the result be when the divisor is given as zero? Without a monadic type, the whole computation may well fail at this point, but using the monadic bind operation we can describe how these *Nothing* results should be threaded through the rest of the computation. We can define the Maybe monad in just such a way, by giving an instance of the Monad type-class.

```
instance Monad Maybe where
  return          = Just
  Nothing >>= f   = Nothing
  (Just x) >>= f  = f x
```

Now, a *Nothing* value is threaded through to the end of the computation, but if a *Just* value is encountered then it is used in the normal way for the value in the underlying type. Before looking at a more in-depth example of a monad in Haskell, I shall finish off my analogy of a division function acting on integers, by defining just such a function.

$$div :: Int \rightarrow Int \rightarrow Maybe\ Int$$

$$\_\ `div`\ 0 = Nothing$$

$$a\ `div`\ b = Just\ (a\ /\ b)$$

Our second example shall be an extension of the State monoid that we defined previously. The state monad still uses transition functions to define the actual computations, but gives access to intermediary results that allow effectful computation to occur. This ability for effectful computation comes from the fact that intermediary result can now effect the computation, or in other words, the values returned by our computations can depend on previous effectful computations and not necessarily just on the original input to the computation. A slight remodelling of our *State* data-type gives us the first glimpse at how these intermediary results can be extracted. The following *StateM* data-type is actually a type-constructor, allowing a transition function over a type s to return values of type a, as well as the new state (again in s).

$$\textbf{data}\ StateM\ s\ a = StateM\ \{runStateM :: s \rightarrow (a, s)\}$$

The monadic structure can now be defined so that the state is threaded through a computation. The *return* function can simply create a transition function that returns the underlying value, but has no effect on the state, and the bind function ($\gg\!=$) returns a transition function that extracts the value and new state from applying its left hand argument to the given state, using these new values in applying its right hand argument.

$$\begin{array}{l} \textbf{instance}\ Monad\ (StateM\ s)\ \textbf{where} \\ \quad \begin{array}{lcl} return\ a & = & StateM\ (\lambda s \rightarrow (a, s)) \\ (StateM\ x) \gg\!= f & = & StateM\ (\lambda s \rightarrow \textbf{let}\ (v, s') = x\ s \\ & & \qquad\qquad\qquad \textbf{in}\ runStateM\ (f\ v)\ s') \end{array} \end{array}$$

Stateful computations can be defined using the given *StateM* monad, but it is often easier to think of stateful programs in *StateM* by defining an interface. In Haskell, we can define what is known as the *MonadState* class to do this

```
class MonadState m s | m → s where
  get :: m s
  put :: s → m ()
instance MonadState (State s) s where
  get = State (λs → (s, s))
  put s = State (λ_ → ((), s))
```

The *get* function is used to return the current state as the returned value, and leaves the overall state unchanged, and the *put* function updates the state to the given value. Since *put* doesn't return any information we are using Haskell's unit type (). The $m \rightarrow s$ in the type definition of the class corresponds to a functional dependency. In Haskell a functional dependency is a hint to the type-checker that one of the argument types (the s in the example given) can be determined given the other argument type (the m in the given example).

We have now seen a bit of an introduction to how monads are used in Haskell to create effectful programs. In fact, Haskell has to make extensive use of monads when dealing with many aspects of computation that would be standard in impure languages. The next section shall look specifically at what is known as the IO Monad in Haskell, which is Haskell's interface to I/O. The extensive use of monads in Haskell has also lead to Haskell having some special syntax for monadic computations known as 'do' notation, and the next section shall also introduce this along with some examples from the IO Monad.

4.5 The IO Monad and 'do' notation

The IO Monad in Haskell is a monad defined in Haskell containing many standard I/O functions. As we have already seen, monads are used in Haskell to deal with side-effects, and as such it is only natural for I/O to take place within a monadic structure. There are many I/O functions defined within the IO Monad so we shall only take a look at a few here, but much more information is available on-line at the Haskell homepage. As an aside it is interesting to note that writing the standard “Hello World” program in Haskell requires the use of the IO Monad (see figure 4.1). The “Hello World” program has the type *IO* () as it doesn’t return any result, but has the side effect of putting the string “Hello, World!” to the standard output. (The function $putStrLn :: String \rightarrow IO\ ()$ is defined as part of the IO Monad.)

$main :: IO\ ()$
$main = putStrLn$ `"Hello, World!"`

Figure 4.1: “Hello World” written in Haskell

We have seen previously, that monadic operations in Haskell are defined with a bind ($\ggg=$) function, and the *return* function. This is also the case for the IO Monad. If we wish to compose functions from the IO Monad then we need to make use of the bind operation, and if we wish to return a result within the IO Monad then we must make use of the *return* function. For example, if we wanted to define a function that prompts a user for their name, and then outputs a string that welcomes the user, we would have to bind together the two occurrences of $putStrLn :: String \rightarrow IO\ ()$ and the function (defined as part of the IO Monad) that reads in a string from the standard input ($getLine :: IO\ String$) as follows:

$welcome :: IO\ ()$

$welcome = putStrLn$ `"Please enter your name:"`

$\ggg= \lambda\_ \rightarrow getLine$

$\ggg= \lambda name \rightarrow putStrLn$ (`"Hello, "` $+\!\!+\ name$)

I have laid out the code in quite a readable manner, but as these monadic functions become longer and longer, it is often easier to think of them in an imperative manner. E.g. within the monadic computation we can bind results from other monadic operations to variables, and then use these results in later operations within the overall monadic computation. Haskell provides us with the ’do’ notation for this purpose. It is in fact just syntactic sugar, and is converted back into the monadic binds at compile time, but it enables the user to have a more imperative style when writing monadic programs. It can be used for any monad defined in Haskell, which is not only useful here for our examples of using the IO Monad, but also later in this thesis when we define the Quantum IO Monad in Haskell (See chapter 5). As a first introduction to 'do' notation we can simply re-write the previous example (*welcome*) using it as follows:

```
welcome' :: IO ()
welcome' = do putStrLn "Please enter your name:"
              name ← getLine
              putStrLn ("Hello, " ++ name)
```

The number of functions available in the IO Monad is quite large, and therefore it is not realistic to look into all the implementations of the functions. Because of this, the IO Monad is usually introduced in terms of the I/O functions it provides, or more abstractly just as an interface to I/O computations in Haskell. In Chapter 5 I shall also take this approach for introducing the QIO Monad as an interface to quantum computations in Haskell, although I shall go on to describe the implementation of the QIO Monad later in Chapter 7. For the rest of this section, I shall introduce a few more common I/O functions that are provided in the IO Monad, and give some examples of monadic programs written in 'do' notation that make use of them.

One such use of the IO Monad, which we shall be using in our implementation of the QIO Monad, is to enable the use of a random number generator. A random number generator is obviously an impure function as we wouldn't want it to return the same value every time it was called. In Haskell, it is possible to create many random number generators, but for this example we shall make use of the "global" random number generator that sits in the IO Monad. There is a type class provided in the library *System.Random* called *Random*, and any instance of this type class (a) must provide a function *randomIO* :: *IO a*, which returns a random element of the given type, and a function *randomRIO* :: $(a, a) \rightarrow$ *IO a* which returns a element of the given type that is in the range given by the argument pair. Some common instances of the *Random* class are booleans, integers, characters, and floating point numbers. As an example we could write a short dice playing game, whereby a user enters their name and tries to throw a 6. (Note that we would have to first import the necessary *System.Random* library.)

```
diceGame :: IO ()
diceGame = do putStrLn "Please enter your name: "
              name ← getLine
              diceGame' name
diceGame' :: String → IO ()
diceGame' name =
  do putStrLn "Press any key to roll the dice..."
     getChar
     x ← randomRIO (1 :: Int, 6)
     putStr ("\b" ++ "You threw a " ++ show x ++ "... ")
     if x ≡ 6 then do putStrLn "You Win!"
                      putStrLn ("Thank you, " ++ name)
              else do putStrLn "You Lose!"
                      diceGame' name
```

Calling the *diceGame* function will prompt the user for their name and pass the given name to the *diceGame′* function. The *diceGame′* uses the random number generator to simulate the throwing of a dice by returning a random number (integer) in the range of 1 to 6. The game repeats until the user has thrown a 6, at which point they have won and the function call can exit. This is a nice example of writing monadic programs in the IO monad, and also shows how the 'do' notation gives our effectful programs are more imperative look.

Another use of the IO Monad (and monads in general) is to enable stateful programs, for example, programs that require mutable data and/or references to data stored in memory. In Haskell, we have *IORef*s to enable the use of mutable references in the IO Monad. It is also useful at this point to mention that there are other monads which are provided in the standard libraries for similar uses, such as the *State* monad which we introduced previously.

4.6 Reversible Computation in Haskell

Haskell doesn't come with a library for reversible computation, but because of the pure nature of functions written in Haskell, it lends itself quite nicely to defining reversible computations. It is the job of the programmer to ensure that the functions we define are reversible, although we could define a universal set of reversible gates and combinators from which we can define reversible computations. This approach would be very similar to the classical subset that can be used in defining computations in the QIO Monad.

A simple example of reversible computation written in Haskell is to implement a toy language that represents reversible circuits, along with an interpreter for *running* the circuits over lists of Boolean values.

The first thing we define for our toy language is the data-type that represents the possible gates in a circuit.

```
data Gate = Empty
  | X Gate
  | Control Gate Gate
  | DWire Gate Gate
```

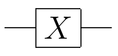
The names of these constructors represent the operations each gate performs. We shall look shortly at what these operations actually are. The last *Gate* argument to each constructor (except the *Empty* constructor) gives us a recursive structure that can be used to build up circuits of gates. As such, it is useful to think of the *Gate* data-type in terms of a monoid.

```
instance Monoid Gate where
  mempty = Empty
  mappend Empty g' = g'
  mappend (X g) g' = X (mappend g g')
  mappend (Control c g) g' = Control c (mappend g g')
  mappend (DWire d g) g' = DWire d (mappend g g')
```

The monoidal identity (*mempty*) is simply the *Empty* constructor, and the sequencing of gates (using *mappend*) can be thought of as pushing the append operation down through the recursive structure until the end of the circuit is reached.

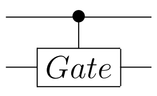
The syntax of the language is given by functions that represent the behaviour of each of the constructors, along with the monoidal constructs to sequence them.

```
x :: Gate
x = X Empty
```



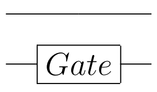
The *x* operation is used to logically negate the first bit in the circuit (or sub-circuit) it represents.

```
control :: Gate → Gate
control g = Control g Empty
```



The *control* operation is used to conditionally apply the gate given as its argument to the corresponding number of bits below it in the circuit (or sub-circuit) it represents.

```
dwire :: Gate → Gate
dwire g = DWire g Empty
```



Finally, the *dwire* operation can be thought of as moving the action of its argument gate down by one wire in the circuit (or sub-circuit) it represents.

As our circuits are designed to be reversible, we can also provide a function that returns the reverse of a given circuit. This *reverse* function works at the level of the *Gate* data-type, simply reversing the order of any gates and sub-gates.

```
reverse :: Gate → Gate
reverse Empty = Empty
reverse (X g) = reverse g `mappend` x
reverse (Control c g) = reverse g `mappend` control (reverse c)
```

$reverse\ (DWire\ d\ g) = reverse\ g\ \text{`}mappend\text{`}\ dwire\ (reverse\ d)$

Now we've defined the syntax of our language, we are able to define some examples of circuits written in it. For example, we can define the Toffoli gate,

$toffoli :: Gate$

$toffoli = control\ (control\ x)$

or the *control′* gate that only runs its argument when the control wire is in the *False* state.

$control' :: Gate \to Gate$

$control'\ g = x\ \text{`}mappend\text{`}\ control\ g\ \text{`}mappend\text{`}\ x$

Finally, to show the use of the *dwire* operation, we can define a circuit that conditionally applies the x operation to the two wires below the control wire.

$controlXX :: Gate$

$controlXX = control\ x\ \text{`}mappend\text{`}\ control\ (dwire\ x)$

If we want to be able to run these circuits, then we must define an evaluation function for our toy language of reversible gates. We can think of a reversible circuit as a function that takes a list of Boolean values, to a list of Boolean values. As such, we define the data-type *Circuit* to represent these functions.

$\mathbf{data}\ Circuit = Circuit\ \{c :: [Bool] \to [Bool]\}$

In order to make the design of the evaluation function easier, it is useful to lift the underlying monoidal behaviour of our *Gate* data-type into an equivalent monoidal behaviour of the *Circuit* data-type.

$\mathbf{instance}\ Monoid\ Circuit\ \mathbf{where}$

$\quad mempty = Circuit\ id$

$\quad (Circuit\ f)\ \text{`}mappend\text{`}\ (Circuit\ g) = Circuit\ (g \circ f)$

The empty circuit is simply the *id* function, and sequencing of circuits is achieved by functional composition.

We can now go on to define the actual *eval* function that formalises the behaviour of circuits (or members of the *Gate* data-type), lifting them to the functional representation of a *Circuit*.

$eval :: Gate \rightarrow Circuit$

$eval\ Empty = mempty$

$eval\ (X\ \ g) = Circuit\ (\lambda(x : xs) \rightarrow ((\neg\ x) : xs))$
$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\text{`}mappend\text{`}\ eval\ g$

$eval\ (Control\ c\ g) = Circuit\ (\lambda(x : xs) \rightarrow (x : (\mathbf{if}\ x\ \mathbf{then}\ (run\ c\ xs)$
$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\mathbf{else}\ xs)))$
$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\text{`}mappend\text{`}\ eval\ g$

$eval\ (DWire\ d\ g) = Circuit\ (\lambda(x : xs) \rightarrow (x : (run\ d\ xs)))$
$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\text{`}mappend\text{`}\ eval\ g$

It's quite easy to see how these functions relate to the behaviour we described for each *Gate* above. Running a computation can now be thought of as applying an evaluated circuit, to a list a Booleans.

$run :: Gate \rightarrow [Bool] \rightarrow [Bool]$

$run\ g\ bs = c\ (eval\ g)\ bs$

With such an evaluator, we could define reversible computations in terms of our reversible circuits, or give irreversible computations by embedding them into such a reversible circuit. An example of an irreversible computation embedded in a reversible circuit would be to define the logical and function (&) by extracting the third Boolean value from running the Toffoli gate with the two input values, and a third value set to *False*.

$(\&) :: Bool \rightarrow Bool \rightarrow Bool$

$a\ \&\ b = (run\ toffoli\ [a, b, False])\ !!\ 2$

Such a toy language shows how the functional style of programming in Haskell lends itself very nicely to defining reversible computations.

Chapter 5

QIO - The Quantum IO Monad

This chapter introduces the Quantum IO Monad (QIO), written in Haskell. Much of the work in this chapter, and the following two chapters, shall also appear in the forthcoming book "Semantic Techniques in Quantum Computation" ([AG10]).

5.1 QIO in Haskell

As we have already seen (in section 4.3), Haskell uses the categorical notion of a monad to enable effectful programs to be designed. Any effects that a program has are described explicitly by the monad in which they take place, allowing Haskell programmers to create monads for describing any arbitrary effects they wish to model. This lends itself very nicely to modelling quantum computations in Haskell, by using a quantum monad that explicitly defines the side effects that occur in a quantum computation. Namely, the side effects that can occur from measurements in a quantum computation (see section 2.3). The Quantum IO Monad (QIO), is a monad defined in Haskell which acts as an interface to quantum programming. It is only a first approximation to a functional interface, and we shall discuss some of its pitfalls at the end of chapter 7. First, this chapter shall introduce the Quantum IO Monad as a Haskell library, which allows quantum computations to be defined using Haskell syntax, in a functional manner. Then, chapter 7 will go over the implementation of the quantum simulator functions that can be used to *simulate* the actual running of the quantum computations defined in the QIO monad.

This implementation of the Quantum IO monad provides a *constructive semantics* for quantum programming, in the sense that the functional programs can also be understood as a mathematical model of quantum computation. The monadic structure of QIO means that the side-effects from measurements are built explicitly into the language, and enables us to keep the definition of unitary operators separate from the monadic structure until we wish to define actual quantum computations over specific qubits. Once quantum computations are defined, the quantum simulator functions enable us to simulate the running of the computation, giving a probabilistic result, or provide a probability distribution over all the possible results of the computation. The approach taken here, is that when a sufficiently sized quantum *register* becomes available (and affordable), it can be swapped in for these simulator functions, or more specifically the *run* simulator function that returns a single probabilistic result.

We shall introduce the Quantum IO Monad in a similar manner as introductions to the IO Monad, using examples to show the available constructs and how they are used in defining quantum computations. Section 5.6 will then give a recap of all the constructs, and a brief discussion on their relation to the categorical model introduced in chapter 3. In the following chapter, we shall go on to give fully developed versions of a few of the most famous quantum algorithms written in QIO, such as Deutsch's algorithm, Quantum teleportation, and even an implementation of Shor's algorithm. The implementation of Shor's algorithm also introduces a library of reversible arithmetic operators written in QIO that are used to define the necessary modular exponentiation function. Shor's algorithm also makes use of the quantum Fourier transform, again defined as a quantum computation in QIO.

5.2 The QIO interface

We think of quantum computations as acting in a quantum device attached to our classical computer. We think of the device abstractly in terms of the commands it can be instructed to carry out. In this sense, the quantum device must contain a register of qubits that can be addressed, and set by the device to one of the computational base states, (i.e. $|0\rangle = \mathit{False}$ or $|1\rangle = \mathit{True}$). The device must also be able to apply unitary operations over one or several of the qubits it contains, and finally it must have the ability to measure the qubits, and return the outcomes of measurements as the corresponding boolean value. As we have seen previously, it is this measurement operation that can lead to side-effects occurring in the rest of the quantum system in the device, and gives rise to our monadic approach. The results of the measurements from the quantum device will be probabilistic. The *run* function we provide implements a classical simulation of this idealised quantum device, although we do also provide a quantum simulator function (*sim*) that (as it is only simulating the quantum state, and hence has full knowledge of the state) is able to return a probability distribution over all the possible results. In this approach, any classical computation which uses results from a quantum computation is able to occur in our classical system, or in this case standard Haskell.

Figure 5.1 is an overview of the Quantum IO monad's API. The type *Qbit* is used to represent the qubits in the quantum device, and the type U is the type of unitary transformations that can be constructed. The type constructor *QIO* is the monadic type constructor for the Quantum IO monad, and for any type a, the type *QIO* a can be thought of as a quantum computation that returns a member of the underlying type a. Specifying that the type constructor *QIO* is a monadic type simply means that we have the standard monadic functions available for it (*return* and $\gg\!=$). This API also shows how we are able to keep the definition of our unitary operators in U separate from the monadic structure of *QIO*, as it is defined with its own monoidal structure, which corresponds to having an *mappend*

$Qbit :: *$
$QIO :: * \to *$
$U :: *$
instance *Monad QIO*
$mkQbit :: Bool \to QIO\ Qbit$
$applyU :: U \to QIO\ ()$
$measQbit :: Qbit \to QIO\ Bool$

instance *Monoid U*
$swap :: Qbit \to Qbit \to U$
$cond :: Qbit \to (Bool \to U) \to U$
$rot :: Qbit \to ((Bool, Bool) \to \mathbb{C}) \to U$
$ulet :: Bool \to (Qbit \to U) \to U$
$urev :: U \to U$

$Prob :: * \to *$
instance *Monad Prob*
$run\ \ :: QIO\ a \to IO\ a$
$sim\ \ :: QIO\ a \to Prob\ a$
$runC :: QIO\ a \to a$

Figure 5.1: The QIO API

operation that is the sequential composition of unitaries, with the identity unitary given by *mempty*.

As has been previously introduced in section 4.5, the monadic structure of *QIO* allows us to use Haskell's **do** notation to give our quantum computation a more imperative feel. In the rest of this section, we shall start to look at the rest of the constructs introduced in figure 5.1. It is quite clear to see how the constructors of the monadic *QIO* type correspond to the behaviour of our generalised quantum device, and to some extent how the underlying members of the *U* data-type correspond to some of the morphisms in our category $\mathbf{FQC}^{\simeq}$ of finite quantum circuits. We shall introduce them here to give a clearer understanding of how they are used, and what they can achieve.

Our first example, is just a simple quantum computation that initialises a qubit into the quantum base state $|0\rangle$, and then returns the result of measuring that qubit.

$hqw :: QIO\ Bool$

$hqw = \mathbf{do}\ q \leftarrow mkQbit\ False$
$\qquad\qquad measQbit\ q$

The computation uses Haskell's **do** notation to assign the variable name q to the initialised qubit, and then bind this assignment into the measurement of the qubit q. Running the *hqw* computation would simply return *False* with probability 1.

It is easy to see that this is the case as the qubit is initialised into the state $|0\rangle$ that corresponds to the classical state *False*, and is then simply measured without any unitary operations having been applied to the qubit.

Computations in QIO become more interesting when we start looking at applying unitary operations to our qubits. To keep things simple, we shall look at the Hadamard transform.

$$\frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

The Hadamard transform is used quite extensively in quantum computation, so we provide it as an instance of a rotation in QIO. We shall see a little later how we have implemented the Hadamard transform, but for now we just need to use the function *uhad* :: *Qbit* → *U*.

```
rnd :: QIO Bool
rnd = do q ← mkQbit False
         applyU (uhad q)
         measQbit q
```

The expression *applyU* :: *U* → *QIO* () is used to apply the supplied argument unitary. It is this operation that allows us to embed the reversible (unitary) operations into the non-reversible quantum computations. The computation can be read as simply initialising the qubit q into the state $|0\rangle$ corresponding to the classical state *False*, then applying the Hadamard transform to qubit q, and finally returning the result of measuring the qubit q. Running this quantum computation will result in a random Boolean result, with each Boolean (*False* or *True*), having an equal probability of 0.5.

In QIO, we could use our quantum simulator function *run* :: *QIO a* → *IO a* to simulate the running of the *rnd* function giving us a probabilistic result. This is possible because the *run* function embeds the *QIO* computation into the *IO* monad, which gives us access to Haskell's random number generator, and hence this form of probabilistic computation. The other quantum simulator function we provide could also be called, using *sim rnd*, and would result in a probability distribution being returned $([(\mathit{True}, 0.5), (\mathit{False}, 0.5)])$. The *sim* function doesn't need to embed the result in the *IO* monad as it doesn't need access to the probabilistic functions of *IO*. The result however, still isn't pure as the computation also has to simulate the side-effects of the quantum aspects of the computation, and thus we have defined the *Prob* monad in which we can embed these probability distributions. We shall look at the *Prob* monad in more detail later in Chapter 7.

Although, we have now introduced all three monadic constructs of *QIO*, we haven't actually given any computations that are truly quantum in their nature (the *rnd* function is simply a classical probabilistic computation). We move on now to look more at the unitary transformations we can define, of type U, and the first truly quantum computation we look at takes advantage of the entangling properties of the conditional unitary *cond* to produce a bell state, which as we have seen is a maximally entangled two-qubit state. The conditional unitary (like our choice morphism in $\mathbf{FQC}^{\simeq}$) doesn't measure the control qubit, and as such, if the control qubit is in a super-position we can introduce *quantum parallelism* into our computations. For example, given $q :: \mathit{Qbit}$ and $t, u :: U$ the expression $\mathit{cond}\ q\ (\lambda b \rightarrow \mathbf{if}\ b\ \mathbf{then}\ t\ \mathbf{else}\ u)$ intuitively runs the unitary t or u depending on q. However, in any case where the qubit q is in a super-position, both unitaries t and u will contribute to the result, giving an entangled state, where the result of t is entangled with the $|1\rangle$ part of q, and the result of u is entangled with the $|0\rangle$ part of q.

To create a bell state, we can use a one-sided version of the conditional (*ifQ*), that is a conditional that applies the empty computation when the control qubit is in the state $|0\rangle$.

```
testBell :: QIO (Bool, Bool)
testBell = do qa ← mkQbit False
              qb ← mkQbit False
              applyU (uhad qa)
```

```
$applyU\ (ifQ\ qa\ (unot\ qb))$
$a \leftarrow measQbit\ qa$
$b \leftarrow measQbit\ qb$
$return\ (a, b)$
```

The computation can be thought of as initialising two qubits (qa and qb) into the base state $|0\rangle$, and then applying the Hadamard transform to one of these qubits (qa) to give us a qubit in the equal super-position state $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$. This qubit is then used as the control qubit in our one-sided conditional (ifQ) to apply the not rotation ($unot$) to the second qubit (qb). This can be thought of as taking the original state $\frac{1}{\sqrt{2}}(|00\rangle + |10\rangle)$ to the state $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle$, as the unitary is only applied to the part of the state in which the control qubit is in the state $|1\rangle$. Our computation then goes on to measure the two-qubits individually, and return the pair of the results of the two measurements.

Evaluating *sim testBell* reveals that the two apparently independent measurements always agree: $[((True, True), 0.5), ((False, False), 0.5)]$. This is because when we measure one of the qubits in the entangled state $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$, the entire quantum state must be projected into one of the states $|00\rangle$ and $|11\rangle$, depending upon the outcome of the measurement, leaving the second qubit in the same state as the first. The the probability of either measurement is $\frac{1}{2} = |\frac{1}{\sqrt{2}}|^2$ (This exactly corresponds to the example we gave in Chapter 2.3.3).

We shall look later at some of the draw-backs of this implementation of conditionals, as the syntax does allow us to define conditionals that don't actually give rise to a semantically unitary operation (see section 7.4). This problem arises from the type-system of Haskell being too weak to define that the state of the control qubit must be separable from the state of any qubits that are used in the branches. In this Haskell implementation of *QIO*, we are able to catch errors of this kind at run-time, but a more-expressive system could allow these exceptions to be caught at compile time by the type checker, before any code is actually run on a quantum system. This, and a few other semantic side-conditions are described in section

7.4, and are the main reasons for the re-development of *QIO* in Agda, which is presented later in Chapter 9.

Using Haskell as the language in which we embed *QIO* means that we are able to use all the functional abstractions that it provides. These functional abstractions give rise to many ways to organise our programs more succinctly, and means we are able to reuse functions that have previously been defined. This approach could be used to provide a library of *QIO* computations, and unitaries (in U) that can be imported along with the *QIO* API, so they can be used in other quantum programs. As an example, the reversible arithmetic functions we define are in a separate file, and can be imported when arithmetic functions are needed in other *QIO* programs (See the on-line code repository ([Gre09]) for more details).

One example is that of quantum sharing. The idea of quantum sharing is that instead of being able to copy a quantum state (which is impossible due to the no-cloning theorem) we are able to share a quantum state amongst groups of entangled qubits. For example, quantum sharing is used in QML ([AG05]) and for the linear-algebraic λ-calculus ([AD08]) to model non-linear use of quantum variables. Also, we have already made use of quantum sharing in our definition of the *testBell* example. In the *testBell* example, we share the state of the qubit *qa*, when it is in the state $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$, with the qubit *qb*. We can define the function *share* as a quantum computation in *QIO*, that when given a qubit, returns a new qubit with which it now shares its state.

```
share :: Qbit → QIO Qbit
share qa = do qb ← mkQbit False
              applyU (ifQ qa (unot qb))
              return qb
```

It is important to realise that the quantum state is not copied (as this would contradict the no-cloning theorem), but merely shared, with the state of the original qubit exactly defining the entangled state of the pair of qubits. That is, if the input qubit is in an arbitrary state $\alpha\,|0\rangle + \beta\,|1\rangle$, then the state after sharing is

$\alpha\,|00\rangle + \beta\,|11\rangle$.

Another example is of creating a qubit already in a super-position, for example as a way of creating qubits in a different basis to the computational basis. For example, one such basis is the $|+\rangle$, $|-\rangle$ basis, which is simply a Hadamard rotation of the computational basis. Functions to create qubits in these states can be defined by:

```
|+⟩ :: QIO Qbit                      |−⟩ :: QIO Qbit
|+⟩ = do q ← mkQbit False            |−⟩ = do q ← mkQbit True
         applyU (uhad q)                      applyU (uhad q)
         return q                             return q
```

In our *testBell* example, we measured the bell state to show the correspondence between the entangled qubits, but if we wanted to use a bell state in a further computation then we would want a function that just returns the pair of qubits that are in the entangled bell state. Using the functional abstractions we have defined above, this could now simply be given by:

```
bell :: QIO (Qbit, Qbit)
bell = do qa ← |+⟩
          qb ← share qa
          return (qa, qb)
```

The function *testBell* that measures the bell state can now be given by:

```
testBell = do (qa, qb) ← bell
              a ← measQbit qa
              b ← measQbit qb
              return (a, b)
```

Looking at our computations now, it seems that we could make our programs more succinct if we were able to define measurement operations for larger quantum data-structures than just the single qubits. The next section shall look at how we can achieve this by defining a type-class in Haskell of Quantum data-types. We shall come back later to look at the other members of the U datatype that haven't been introduced yet.

5.3 Quantum datatypes for QIO

We mentioned briefly how we can use Haskell's class system to define quantum data-structures that are larger than just single qubits, and how we can use this to combine measurements of these quantum data-structures such that we don't have to measure each qubit individually. In fact, if we look at the monadic structures available in *QIO*, we can see a kind of symmetry between the $\mathit{mkQbit} :: \mathit{Bool} \rightarrow \mathit{QIO}\ \mathit{Qbit}$ and $\mathit{measQbit} :: \mathit{Qbit} \rightarrow \mathit{QIO}\ \mathit{Bool}$ functions. They define a kind of relationship between values of type *Bool* (or bits), and their quantum counter-part *Qbit* (or qubits). This is a relationship that can be extended to other data-structures that can provide a quantum counter-part, such that a member of the quantum type can be initialised from the classical type, and the measurement of the quantum structure gives rise to a member of the classical type. A simple example would just be the correspondence between a pair of Boolean values, and a pair of qubits. More precisely, if we had functions $\mathit{mk2Qbits} :: (\mathit{Bool}, \mathit{Bool}) \rightarrow \mathit{QIO}\ (\mathit{Qbit}, \mathit{Qbit})$ and $\mathit{meas2Qbits} :: (\mathit{Qbit}, \mathit{Qbit}) \rightarrow \mathit{QIO}\ (\mathit{Bool}, \mathit{Bool})$, then they exactly define the correspondence. In fact, if we had these functions we could already have made use of them in the *testBell* example in the previous section. In more general terms, if we can define similar functions for any corresponding classical and quantum data-types then we could use them in our *QIO* computations. This requirement is analogous to the example in section 4.4 (that any type with an ordering can be sorted), and it is easy to see that in this case we can also use a type-class to define such a requirement.

As such, we can introduce the type-class *Qdata*, which defines that for any types to fulfil the class requirements, they must provide the functions that define the correspondence. That is, for a classical type a and a corresponding quantum type qa, we must be able to define the functions $\mathit{mkQ} :: a \rightarrow \mathit{QIO}\ qa$ and $\mathit{measQ} :: qa \rightarrow \mathit{QIO}\ a$. That is, that the *Qdata* type-class is given by

```
class Qdata a qa | a → qa, qa → a where
  mkQ :: a → QIO qa
```

```
    measQ :: qa → QIO a
```

In fact, we can start to look at other operations we have that act on qubits, and see if they can be generalised over larger quantum data-structures. For example, we can also generalise the conditional operation to have a larger quantum data structure as its control, such as defining the function $cond2Qbits : (Qbit, Qbit) \rightarrow ((Bool, Bool) \rightarrow U) \rightarrow U$, that gives a conditional acting over two qubits. We'll see later how the *ulet* constructor can also be generalised in this way, but shall explain more about that when we've actually introduced the behaviour of *ulet*. For now, we shall just extend our definition of the *Qdata* type-class to also include the generalised version of the conditional unitary.

```
  class Qdata a qa | a → qa, qa → a where
    mkQ :: a → QIO qa
    measQ :: qa → QIO a
    condQ :: qa → (a → U) → U
```

The simplest instance of the *Qdata* type-class is with the correspondence between Booleans and qubits which we have already been using. The type-class functions are just given by the underlying *QIO* constructors.

```
  instance Qdata Bool Qbit where
    mkQ = mkQbit
    measQ = measQbit
    condQ q br = cond q br
```

The correspondence we gave as an example between pairs of boolean values, and pairs of qubits could also be given as an instance of the type-class. We can however give this in more general terms, that given any two instances of *Qdata*, with a and b being the underlying classical types, and qa and qb being the corresponding underlying quantum datatypes, we have an instance of *Qdata* between the pairs (a, b) and (qa, qb). That is, that *Qdata* is *closed* under pairing, with the necessary functions defined in terms of the underlying functions of each member of the pair.

```
instance (Qdata a qa, Qdata b qb) ⇒ Qdata (a, b) (qa, qb) where
  mkQ (a, b) = do qa ← mkQ a
                  qb ← mkQ b
                  return (qa, qb)
  measQ (qa, qb) = do a ← measQ qa
                      b ← measQ qb
                      return (a, b)
  condQ (qa, qb) br = condQ qa (λx → condQ qb (λy → br (x, y)))
```

Other instances of this class include the closure of *Qdata* over lists

```
instance Qdata a qa ⇒ Qdata [a] [qa] where
  mkQ n = sequence (map mkQ n)
  measQ qs = sequence (map measQ qs)
  letU as xsu = letU′ as []
    where letU′ [] xs = xsu xs
          letU′ (a : as) xs =
               letU a (λx → letU′ as (xs ++ [x]))
  condQ qs qsu = condQ′ qs []
    where condQ′ [] xs = qsu xs
          condQ′ (a : as) xs =
               condQ a (λx → condQ′ as (xs ++ [x]))
```

and in section 6.3 we'll also present a quantum integer datatype (*QInt*), that is a member of *Qdata* along with *Int* as its corresponding classical datatype.

Before looking back at the *QIO* constructors we haven't yet introduced, we'll use our generic operation *measQ* to redefine the *testBell* function once more.

```
testBell :: QIO (Bool, Bool)
testBell = do qab ← bell
              measQ qab
```

5.4 More on the QIO interface

Looking back at the *QIO* API in figure 5.1, we'll see that there are still some constructs that we haven't explained yet. We have explained some of the members of the *U* data-type, which is the type of unitary transformations in *QIO*. This type is defined as a monoid, and as such comes with the binary operator, *mappend* in Haskell, that corresponds to sequential composition. The identity of this monoidal structure, *mempty*, is just the empty computation.

It is useful to point out here, that although we have chosen our unitary constructs to follow on from the constructs we looked at in defining the category $\mathbf{FxC}^{\simeq}$ of circuits, the monoidal structure here doesn't correspond to the monoidal structure present in $\mathbf{FxC}^{\simeq}$ that denotes parallel composition of circuits. In *QIO*, we can think of the corresponding parallel composition of circuits as the sequential composition of unitaries acting on different qubits in the system. The syntax itself doesn't define any form of parallelising unitaries that act on different qubits, and neither does the semantics of our simulator functions. However, it would be possible if the "plug-in" quantum hardware supports the parallel running of unitaries over different qubits, that a compiler could figure out optimisations where this could occur.

We can now give our definition of the one-sided conditional *ifQ* that has been used previously in our examples. It is simply defined using a conditional unitary that applies the *mempty* computation for the *false* part of the computation

$$\begin{aligned}
&\mathit{ifQ} :: \mathit{Qbit} \rightarrow U \rightarrow U\\
&\mathit{ifQ}\ q\ u = \mathit{cond}\ q\ (\lambda x \rightarrow \textbf{if}\ x\ \textbf{then}\ u\ \textbf{else}\ \mathit{mempty})
\end{aligned}$$

To increase readability, and simplify the presentation, the rest of this thesis shall be written using *mappend* as ► and *mempty* as •.

Rotations in *QIO* are another primitive operation in U, they are defined in a very similar manner to our one "bit" operations in $\mathbf{FQC}^{\simeq}$, by any unitary 2×2 complex valued matrix. These matrices are given a functional definition by representing them as functions in $(\mathit{Bool}, \mathit{Bool}) \rightarrow \mathbb{C}$. The matrix that corresponds to an arbitrary rotation $f :: (\mathit{Bool}, \mathit{Bool}) \to \mathbb{C}$ can be thought of as;

$$\begin{bmatrix} f(False, False) & f(False, True) \\ f(True, False) & f(True, True) \end{bmatrix}$$

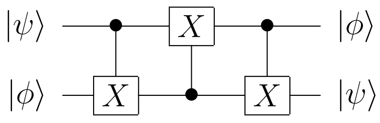
the *rot* construct of U, lifts these rotations to be unitary operators in U by defining which qubit they are to be applied to. We can can define the *unot* and *uhad* operations we have used previously, along with the *uphase* operation that when given a real argument $(r :: \mathbb{R})$, corresponds to the phase rotation with phase $\theta = r$.

```
    $unot :: Qbit \to U$
    $unot\ x = rot\ x\ (\lambda(x, y) \to \textbf{if}\ x \equiv y\ \textbf{then}\ 0\ \textbf{else}\ 1)$
    $uhad :: Qbit \to U$
    $uhad\ x = rot\ x\ (\lambda(x, y) \to \textbf{if}\ x \wedge y\ \textbf{then} - h\ \textbf{else}\ h)$
                $\textbf{where}\ h = (1\ /\ sqrt\ 2)$
    $uphase :: Qbit \to \mathbb{R} \to U$
    $uphase\ x\ r = rot\ x\ (rphase\ r)$
    $rphase :: \mathbb{R} \to Rotation$
    $rphase\ \_\ (False, False) = 1$
    $rphase\ r\ (True, True)\ \ = exp\ (0 :+ r)$
    $rphase\ \_\ (\_, \_)\qquad\quad = 0$
```

For example, the Pauli-Z gate is an instance of a phase gate, with the argument π. Or in other words, we could define the rotation *upauli_z* by

```
    $upauli\_z :: Qbit \to U$
    $upauli\_z\ x = uphase\ x\ pi$
```

Another primitive operation that is provided in U is the *swap* construct. The operation *swap* x y can be thought of as swapping the position of the two qubits x and y within the quantum register. The *swap* operation could actually be defined in terms of three controlled not operations, which it is already possible to define as a *QIO* computation. The circuit



would just be implemented as

$$swap :: Qbit \to Qbit \to U$$
$$swap\ qa\ qb = ifQ\ qa\ (unot\ qb)$$
$$\blacktriangleright\ ifQ\ qb\ (unot\ qa)$$
$$\blacktriangleright\ ifQ\ qa\ (unot\ qb)$$

However, we include *swap* as a primitive construct in *U* for reasons of efficiency.

The two constructs we have left to look at are the *urev* function and the *ulet* primitive. The *ulet* primitive is used to introduce ancillary qubits into our computation, and as such has quite a complicated behaviour. We shall go on to look at the behaviour of the *ulet* operation in the next section, but for now we shall finish off this section by introducing the *urev* function.

The *urev* function comes about because of the unitary nature of our *U* data-type. That is, that every definable member of the *U* datatype should correspond to a unitary operator that can be applied to a quantum system. By definition, a unitary operator has an inverse that is also a unitary operator. The *urev* function simply takes a member of the *U* data-type and returns the member of the *U* data-type that corresponds to the inverse of the original. In fact, this is simply achieved at a syntactic level. The fact that *urev* can work at the syntactic level should arise from the fact that members of our *U* data-type only define unitary operations. This is the case in *QIO*, up to the problems we talk about in section 7.4. However, all these problems are caught with errors at run-time, and their inverses are treated in exactly the same way. We shall have a brief look later (in Chapter 7) at how the *urev* function is implemented.

5.5 Ancillary qubits, and the use of ulet in QIO

So far, the use of quantum structures in our computations has to be dealt with explicitly by the programmer. Any use of qubits has to be preceded by an initialisation of the qubit, and any initialised qubits must be kept track of throughout the whole computation. This doesn't necessarily lead to a natural way of programming, especially when we look at how a lot of quantum algorithms make use of ancillary qubits, or ancillary registers of qubits. The structure of many of the algorithms, arising from the reversible nature of the unitary transforms, means that once the result is calculated many of the intermediary computations are run in reverse, and these ancillary qubits are returned to their original state. In fact, it is this behaviour that lead us to the modelling of heap and garbage in the **FxC** category of finite computations. In the specific cases where any auxiliary qubits are completely reset to their original values, these auxiliary qubits have no effect on the unitarity of the computation. In essence, the programmer should only have to deal with initialisations and measurements of the information bearing qubits of a computation, and if certain auxiliary qubits are required, and guaranteed to not effect the unitarity of the computation, then they can be dealt with implicitly by the system. This behaviour is introduced to *QIO* in the form of the quantum (unitary) let function *ulet*. A programmer can introduce temporary qubits into a computation, as long as they can guarantee that the qubit is back in its original state by the end of that part of the computation. Although this may sound like a hard task on the part of the programmer, as qubits can become entangled etc., we are able to use this *ulet* structure in many of our implementations of quantum algorithms. In fact, the *ulet* construct is used extensively in the definition of the quantum arithmetic functions, which we look at in section 6.3. We shall now move on to look at the *ulet* construct in more detail, and show how this form of quantum let can also be extended over our quantum datatypes, and hence added to the *Qdata* type-class.

The *QIO* API enables us to use ancillary qubits with the use of the function

$ulet :: Bool \rightarrow (Qbit \rightarrow U) \rightarrow U$. The expression $ulet\ b\ f$ can be thought of as introducing an ancillary qubit initialised in the base state relating to b, and passes this new qubit q as the argument to the function f, giving us the sub-computation (or more specifically, unitary operation) in which the ancillary qubit is used $(f\ q)$. Once this sub-computation is finished, the qubit q is simply removed from the overall state of the system, and left back in an uninitialised state with all the other un-used qubits. The requirement that after running $f\ q$, the qubit q must be back in the base state b has to be fulfilled by the programmer. This is another semantic side-condition that is caught at runtime by our current system, and is again another reason for the redevelopment of *QIO* in a stronger type system (See chapter 9).

The *ulet* constructor is another function that relates Boolean values with their quantum counter-part; qubits. As such, we are able to generalise the *ulet* structure over larger quantum data-structures, in a very similar manner as we have for the other members of the *Qdata* type-class. In fact, we are able to extend the *Qdata* type-class to impose this generalisation.

$$\begin{array}{l}
\textbf{class}\ Qdata\ a\ qa \mid a \rightarrow qa, qa \rightarrow a\ \textbf{where} \\
\quad \vdots \\
\quad letU :: a \rightarrow (qa \rightarrow U) \rightarrow U
\end{array}$$

The *letU* function is now the generalisation of the *ulet* construct over any quantum data-types that fulfil the *Qdata* type-class along with their classical counter-parts. As we have extended the *Qdata* type-class, we must also extend the instances of the type-class to include the corresponding *letU* function. For example, we can easily extend the correspondence between Booleans and qubits with

$$\begin{array}{l}
\textbf{instance}\ Qdata\ Bool\ Qbit\ \textbf{where} \\
\quad \vdots \\
\quad letU\ b\ xu = ulet\ b\ xu
\end{array}$$

and similarly for the other instances we have defined, E.g pairs of *Qdata*.

$$\textbf{instance}\ (Qdata\ a\ qa, Qdata\ b\ qb) \Rightarrow Qdata\ (a, b)\ (qa, qb)\ \textbf{where}$$

$\vdots$

$$letU\ (a, b)\ xyu = letU\ a\ (\lambda x \rightarrow letU\ b\ (\lambda y \rightarrow xyu\ (x, y)))$$

We have now introduced all the constructs available in the *QIO* API, and given some simple examples of how they are used. The next chapter look at the implementations of some of the algorithms we have seen in section 2.6, written in *QIO*. The final section of this chapter will give a bried recap of the constructs available in *QIO*, and a discussion on their relation to the categorical model of circuits introduced in chapter 3

5.6 QIO Design

Having introduced all the constructs of *QIO* in an example driven manner, it is useful to have a brief recap of all the contructors available. This section aims to list all the *QIO* contructors, along with a brief description, and their relation to the categorical model of circuits introduced in chapter 3. In fact, it is mainly the choice of unitary operators that is influenced by the categorical model, as the circuits we introduced didn't model the initialisation, or explicit measurement of qubits. As such we shall first recap the contructs of the monadic *QIO* type, and then move onto the constructs of the pure monoidal U type of unitary operations.

- $Qbit :: *$

The type *Qbit* is a reference to a *physical* qubit within the system.

- $QIO :: * \rightarrow *$

  instance *Monad QIO*

QIO is a monadic type constructor, that contains the following primitive operations for defining quantum computations.

- $mkQbit :: Bool \rightarrow QIO\ Qbit$

The *mkQbit* construct is used to initialise a qubit into the base state given by the argument *Bool* value.

- $applyU :: U \rightarrow QIO\ ()$

The *applyU* construct is used to apply a unitary operation (of type U which is introduced below) to the current quantum state of a system.

- $measQbit :: Qbit \rightarrow QIO\ Bool$

The *measQbit* construct is used to measure qubits, and return a Boolean result depending on the measurement outcome. It is this measurement construct that can be effectful, causing side effects to the quantum state of the rest of the system, and gives rise to the monadic structure of *QIO*.

- $U :: *$

  instance *Monoid U*

The U data type defines the unitary operations that can be defined for use in *QIO* computations. Its monoidal structure gives rise to an ▶ operation that sequences operations, and a • element that corresponds to the idenity operator. The following primitives can be used to define unitary operations (of type U).

- $swap :: Qbit \rightarrow Qbit \rightarrow U$

The *swap* operation is used to swap the states of two *physical* qubits within the overall quantum state. This corresponds exactly to one of the *wires* constructs as introduced in the categorical model, and any other *wires* construct can be defined in terms of multiple *swap* operations. A brief discussion on the inclusion of the *swap* operation is included at the end of this section.

- $cond :: Qbit \rightarrow (Bool \rightarrow U) \rightarrow U$

The *cond* operation allows conditional operations to be defined, whereby the unitary that is applied depends on the value of a control qubit. The *cond* operation corresponds to the controlled operations introduced in the categorical model, whereby the special cases in which one of the branches is • correspond exactly to the two control structures in the categorical model.

- $rot :: Qbit \to ((Bool, Bool) \to \mathbb{C}) \to U$

The *rot* operation applies a single qubit rotation to the given argument qubit. Instances of the *rot* operation correspond exactly to the one qubit rotations introduced in the categorical model. The second argument $(Bool, Bool) \to \mathbb{C}$ defines exactly the unitary matrix that represents the rotation, with instances of *rot* including the Hadamard rotation, Not rotation (pauli X), and a Phase rotation. Although any single qubit rotation can be defined. The classical subset of *QIO* is only restricted by the instances of *rot* which can be used (E.g. only Not and Id rotations are classical).

- $ulet :: Bool \to (Qbit \to U) \to U$

ulet allows the introduction of ancilliary quibts into a *QIO* computation. The programmer must ensure unitarity by only using the ancilliary qubit in such a way as to return it to its original value.

- $urev :: U \to U$

As all members of the *U* data type are unitary, the *urev* function is able to calculate the inverse (or reverse) of the given argument unitary. This is achieved at the syntactic level.

- $Prob :: * \to *$

  instance *Monad Prob*

The *Prob* type contrsuctor is defined to model the probablistic nature of running quantum computations. As such it has a monadic structure, that is used to create probability distributions instead of a pure return value.

- $run :: QIO\ a \to IO\ a$

The *run* function embeds *QIO* programs into the *IO* monad, using the random number generator to simulate the running of a *QIO* computation.

- $sim :: QIO\ a \to Prob\ a$

The *sim* function embeds *QIO* programs into the *Prob* monad, creating a probability distribution for the results of running a *QIO* computation.

- $runC :: QIO\ a \to a$

The *runC* function is able to run *QIO* programs that only use the classical subset of *QIO*, and can return a pure value as the classical subset doesn't give rise to an effectful model of computation.

- **class** $Qdata\ a\ qa \mid a \to qa, qa \to a$ **where**

  $mkQ :: a \to QIO\ qa$

  $measQ :: qa \to QIO\ a$

  $condQ :: qa \to (a \to U) \to U$

  $letU :: a \to (qa \to U) \to U$

The type class *Qdata* introduces a way of extending the quantum data types for which a *QIO* computation can have access. *mkQ* initialises a quantum data structure, from its classical counterpart, and *measQ* measures a quantum data structure, collapsing it to a single element of the classical counterpart. *condQ* allows a quantum data structure to be used as the control in a conditional unitary, and *letU* allows the quantum data structure to be introduced as an ancilliary structure in a unitary.

- **instance** $Qdata\ Bool\, Qbit \ldots$

  instance $(Qdata\ a\ qa, Qdata\ b\ qb) \Rightarrow Qdata\ (a, b)\ (qa, qb)$ ...

  instance $Qdata\ a\ qa \Rightarrow Qdata\ [a]\ [qa]$ ...

Example instances of the *Qdata* class include the correspondence between Booleans and qubits, and any quantum data structures can be extended over pairs and lists.

The *Qdata* class allows us to overload the operations for each member of the class. As *Bool* and *Qbit* combine to form a member of this class, this leaves the underlying *mkQbit*, *measQbit*, *cond*, and *ulet* constructors looking somewhat redundant. However, it is quite important to introduce these constructs to show that at the lowest level, any member of the *Qdata* class must be defined in terms of the individual qubits that make up the quantum data structure.

It has also been noted that the *swap* operation on qubits looks a little redunant, as unitaries can explicitly reference the qubits on which they act. However, it is left as a construct in *QIO* as it can be useful for moving qubits about within a larger quantum data structure, without having to then explicitly reference the individual qubits within the data structure when you want to pass the whole thing as an argument.

The following chapter looks at some of the most famous quantum algorithms, as introduced in section 2.6, developed in *QIO*.

Chapter 6

Quantum algorithms in QIO

This section goes over the implementation of a few of the famous quantum algorithms we have seen previously in section 2.6. The algorithms we present implemented in *QIO* include Deutsch's algorithm, Quantum teleportation, Quantum (or reversible) arithmetic functions, and the Quantum Fourier Transform (QFT). The final section goes on to look at the development in *QIO* of an implementation of Shor's algorithm, which makes use of some of the previously defined algorithms.

6.1 Deutsch's algorithm

Deutsch's Algorithm ([Deu85]), as we have seen in section 2.6.1, was presented as one of the first and simplest quantum algorithms that could be shown to provide a more efficient solution than its classical counterpart. In Haskell, we can think of the problem as being given a function $f :: Bool \to Bool$, and being asked to calculate whether the given function is balanced or constant.

In the QIO monad the algorithm can easily be modelled: we initialise two qubits in the $|+\rangle$ and $|-\rangle$ states, and then conditionally negate the second qubit depending on the outcome of applying f to the first qubit. The resulting entanglement describes the properties of f which we wish to find out. Applying the Hadamard rotation to the first qubit, and then measuring it, enables us to extract this information.

$$
\begin{aligned}
&\mathit{deutsch} :: (\mathit{Bool} \to \mathit{Bool}) \to \mathit{QIO}\ \mathit{Bool}\\
&\mathit{deutsch}\,f = \mathbf{do}\ x \leftarrow |+\rangle\\
&\qquad\qquad\quad\ \ y \leftarrow |-\rangle\\
&\qquad\qquad\quad\ \ \mathit{applyU}\ (\mathit{cond}\ x\ (\lambda b \to \mathbf{if}\ f\ b\ \mathbf{then}\ \mathit{unot}\ y\ \mathbf{else}\ \bullet))\\
&\qquad\qquad\quad\ \ \mathit{applyU}\ (\mathit{uhad}\ x)\\
&\qquad\qquad\quad\ \ \mathit{measQ}\ x
\end{aligned}
$$

We can see from the code that the function f only appears to be called once, in the argument to a conditional unitary, and this is indeed the case. The quantum control aspects of the *cond* construct allow us to have effectively run the function f over a quantum state. In either of the cases where f was a constant function then the measurement will yield *False* (with probability 1), and in the cases where f is a balanced function the measurement will yield *True* (again with probability 1).

Evaluating, $\mathit{sim}\ (\mathit{deutsch}\ \neg)$ gives $[(\mathit{True}, 1.0)]$, $\mathit{sim}\ (\mathit{deutsch}\ \mathit{id})$ also gives $[(\mathit{True}, 1.0)]$. $\mathit{sim}\ (\mathit{deutsch}\ (\lambda x \to \mathit{False}))$ gives $[(\mathit{False}, 1.0)]$, and $\mathit{sim}\ (\mathit{deutsch}\ (\lambda x \to \mathit{True}))$ also gives $[(\mathit{False}, 1.0)]$.

6.2 Quantum Teleportation

Quantum teleportation has been introduced in section 2.6.7, and can be thought of as the transfer of an arbitrary quantum state (in this case a single qubit) using a pair of entangled quantum states (or qubits) and the communication of two classical bits. With-in the *QIO* monad, we can think of this in three stages. First, before any communication has occurred we must initialise the entangled pair of qubits. Secondly, Alice uses one of these qubits to entangle it with the qubit she wishes to teleport, and produces the classical data from measuring the two qubits now in her possession. The third stage involves Bob receiving the classical measurements, and performing the necessary unitary on his qubit from the entangled pair. To model the requirement that Alice and Bob share an entangled pair of qubits means that the whole process must take part in a single quantum system. In terms of the *QIO* monad, this means that the whole process must take part with-in a single instance of the *QIO* monadic structure.

Alice has her initial qubit (*aq*) and one of the entangled pair of qubits (*eq*). All she has to do is apply a controlled not between these two qubits, and then perform the Hadamard rotation on the first one. The application of the controlled not rotation is very similar to the *share* operation previously defined, however, we cannot simply use the *share* function as we require that the state of the input qubit *qa* is somehow shared with the already entangled state of the *eq* qubit. Finally she has to measure these two qubits and send the results of this measurement to Bob. The results of the measurement of these two qubits corresponds exactly to the requirement that two classical bits of data must be sent to Bob, and without this classical information Bob is unable to determine anything about the state of the qubit he has. We implement Alice's part of the teleportation protocol by the function *alice*.

```
alice :: Qbit → Qbit → QIO (Bool, Bool)
alice aq eq = do applyU (ifQ aq (unot eq))
                 applyU (uhad aq)
                 measQ (aq, eq)
```

Bob will also have a qubit from the entangled pair (*eq*). He will receive the classical data (*cd*) from Alice, and depending upon this data, has to apply the correct unitary (or unitaries) to his entangled qubit *eq*. We model Bob's part of the teleportation protocol by the function *bob*.

```
uZ :: Qbit → U
uZ qb = (uphase qb 0.5)
bob :: Qbit → (Bool, Bool) → QIO Qbit
bob eq (a, b) = do applyU (if b then (unot eq) else •)
                   applyU (if a then (uZ eq) else •)
                   return eq
```

In order to actually *run* the teleportation protocol, we have the requirement that Alice and Bob share an entangled pair of qubits. This entangled pair corresponds exactly to the Bell state we defined in *QIO* previously (*bell*::*QIO* (*Qbit*, *Qbit*)). We can now combine the three stages of the teleportation protocol as follows:

```
teleportation :: Qbit → QIO Qbit
teleportation iq = do (eq1, eq2) ← bell
                      cd ← alice iq eq1
                      tq ← bob eq2 cd
                      return tq
```

The *teleportation* function takes a qubit as an argument, which is the qubit whose state we wish to teleport. The protocol first initialises the entangled pair, and then Alice uses one of the pair along with the input qubit to create the classical data *cd*. Bob can then use this classical data, along with the other qubit from the entangled pair to reconstruct the state of the original input qubit.

6.3 Reversible arithmetic

We first looked at reversible arithmetic because an implementation of Shor's algorithm requires a quantum circuit that calculates the necessary modular exponentiation function over a quantum register. There are quite a few proposals for circuits that perform arithmetic in such a manner [BCDP96, CDKM04, Dra00, Gos98], with most using clever tricks to reduce the overall number of qubits required for the circuit. This section aims to introduce our library of reversible arithmetic functions, written in *QIO* that follow the work in [VBE95] to build up to a modular exponentiation circuit, from smaller reversible circuits that implement addition, modular addition, modular multiplication, and finally the required modular exponentiation. These circuits only use elements from the classical subset of our unitary operators, so the classical simulator function *runC* can be used in testing the circuits over classical inputs. Although the resultant circuits seem to be of a classical nature, it is the way in which we can use them over quantum input states that means they can be used in our implementation of Shor's algorithm later in this chapter (see section 6.5).

As classically it is easier to think of arithmetic circuits acting on decimal integer values, and not on their underlying binary representation, we can first define a type of *quantum integer* that is essentially a wrapper for a list of qubits, along with Haskell's *Int* data-type as its classical counter-part.

```
data QInt = QInt [Qbit] deriving Show
```

Functions that convert between classical integers, and lists of Booleans of fixed length, given by *qIntSize* :: *Int*, are used as an intermediary step in our definition of the *QInt*,*Int* instance of the *Qdata* class. These functions (*int2bits* and *bits2int*) are coded in the normal way, to essentially give a binary representation of the corresponding integers. We define the instance of *Qdata* by lifting the corresponding list operations over the *QInt* datatype.

```
instance Qdata Int QInt where
  mkQ n = do qn ← mkQ (int2bits n)
             return (QInt qn)
  measQ (QInt qbs) = do bs ← measQ qbs
                        return (bits2int bs)
  letU n xu = letU (int2bits n) (λbs → xu (QInt bs))
  condQ (QInt qi) qiu = condQ qi (λx → qiu (bits2int x))
```

The approach for constructing a reversible addition function is now achieved by defining a function that calculates the sum of three qubits, and a function that calculates the corresponding carry qubit. These can be used to work recursively through the list of qubits to keep track of the overall sum and any overflow that might occur.

```
sumq :: Qbit → Qbit → Qbit → U
sumq qc qa qb =
  cond qc (λc →
```

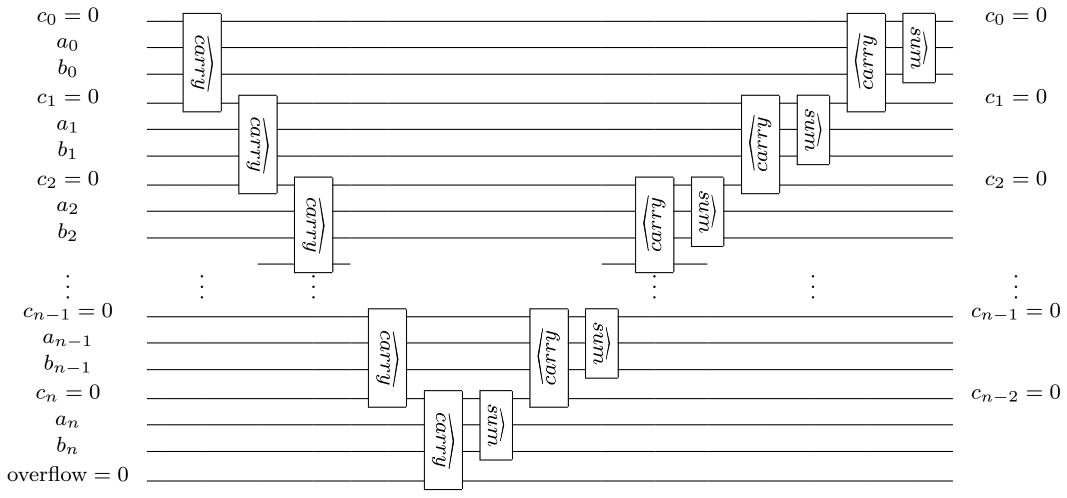


Figure 6.1: A reversible circuit for addition (taken from [VBE95]). The reversible $\widehat{carry}$ and $\widehat{sum}$ functions are evaluated in the direction of the $\frown$ symbol, such that all the $\widehat{carry}$ operations are *undone* after the final overflow value has been calculated, and the carry-bits have been used in their part of the overall sum.

```
        cond qa (λa → if a ≢ c then unot qb else •))
    carry :: Qbit → Qbit → Qbit → Qbit → U
    carry qci qa qb qcsi =
      cond qci (λci →
        cond qa (λa →
          cond qb (λb →
            if ci ∧ a ∨ ci ∧ b ∨ a ∧ b then unot qcsi else •)))
```

We note that *carry* needs access to the current and the next carry-qubit, while *sumq* only depends on the current qubits.

The reversible addition circuit can now be defined in terms of these circuits, whereby the *carry* operations are performed in reverse after the overall overflow bit has been calculated. This corresponds to the circuit diagram for addition given in [VBE95] which is reproduced here in figure 6.1.

```
    qadd :: QInt → QInt → QInt → Qbit → U
    qadd (QInt qas) (QInt qbs) (QInt qcs) qc = qadd′ qas qbs qcs qc
        where qadd′ [] [] [] qc
```

$$
\begin{aligned}
&\quad = \bullet\\
&qadd'\ [qa]\ [qb]\ [qci]\ qc\\
&\quad = carry\ qci\ qa\ qb\ qc \blacktriangleright\\
&\quad\quad sumq\ qci\ qa\ qb\\
&qadd'\ (qa : qas)\ (qb : qbs)\ (qci : qcsi : qcs)\ qc\\
&\quad = carry\ qci\ qa\ qb\ qcsi \blacktriangleright\\
&\quad\quad qadd'\ qas\ qbs\ (qcsi : qcs)\ qc \blacktriangleright\\
&\quad\quad urev\ (carry\ qci\ qa\ qb\ qcsi) \blacktriangleright\\
&\quad\quad sumq\ qci\ qa\ qb
\end{aligned}
$$

This implementation of reversible addition uses three registers of qubits, and an extra qubit that will encode whether there has been any overflow. The first two are simply the registers containing the inputs that are to be added (the resultant sum overwrites the second of these two registers). The third register is an additional register that must be initialised to the state $|\vec{0}\rangle$, which corresponds to a *QInt* representation of the integer zero. This third register is used within the computation to hold each of the intermediary carry-bits, with the final carry computation having the overflow qubit as its output. The qubits in this third register are then reset back to the zero values by the inverse of the carry computation being performed back down through the registers. This behaviour can be seen in the *QIO* implementation by noticing that in the general recursive case, a *urev* of the *carry* function is called corresponding exactly to the call to the *carry* function above it.

This behaviour is also exactly the behaviour we have designed the *ulet* constructor to model, and instead of the user having to keep track of this auxiliary register of qubits, we are able to use our *letU* constructor so that the system deals with these auxiliary qubits implicitly, presenting the circuit as a unitary structure to the user that just requires the two input registers, and an extra qubit that contains any overflow information. Without the use of *letU* to present the algorithm as a unitary structure in this way, the circuit could not be used to define our other unitary arithmetic circuits in a modular fashion, as the user would have to keep track of auxillary qubits used in each and every call to the addition function *qadd* explicitly.

We have given the implementation above to show how our *qadd* function relates to the circuit in figure 6.1, but now give the re-implementation using the *letU* structure to introduce each of the carry qubits into the computation as and when required. If the algorithm didn't ensure these qubits were restored to their original state, then a run-time error would be thrown as we described when introducing the *ulet* construct (in section 5.5).

```
qadd :: QInt → QInt → Qbit → U
qadd (QInt qas) (QInt qbs) qc' =
  letU False (qadd' qas qbs)
  where qadd' [] [] qc = ifQ qc (unot qc')
        qadd' (qa : qas) (qb : qbs) qc =
          ulet False (λqc' → carry qc qa qb qc' ▶
                             qadd' qas qbs qc' ▶
                             urev (carry qc qa qb qc')) ▶
          sumq qc qa qb
```

We can see straight away from the type of this new *qadd* function that it only requires the two registers of qubits that represent the two inputs to be summed, along with the qubit that will encode any overflow of the operation.

As an aside, it is nice to note that we can write a simple add function for integers using this reversible circuit. The computation is written in *QIO*, but as we can use the pure *runC* classical simulator function, we have a pure addition function that has been defined in a reversible manner.

```
add' :: Int → Int → QIO Int
add' a b = do (qa, qb) ← mkQ (a, b)
              overflow ← mkQ False
              applyU (qadd qa qb overflow)
```

$$measQ\ qb$$

$$(+) :: Int \rightarrow Int \rightarrow Int$$

$$a + b = runC\ (add'\ a\ b)$$

This addition function could now be used to test the implementation of the addition circuit, but we must take note that it doesn't deal with overflow if any has occurred. This addition function does indeed behave as would be expected (E.g. calling $8 + 5$ gives 13).

The library of arithmetic functions is now built using this reversible adder unitary (*qadd*) as the basis of larger unitaries that define the more complex functions. We follow the definitions given in [VBE95] to create the functions as given below. The full functions can be found as part of the reversible arithmetic library included in the on-line source code [Gre09], but we simply give the types here with a brief outline of how each function is defined in terms of the functions that preceded it. Each successive arithmetic function is a step towards the goal of creating a reversible circuit that computes modular exponentiation.

The first step is in defining a circuit that implements modular addition using the *qadd* circuit above. These modular addition circuits can be thought of as being indexed by the classical value N, which is the modulus of the function, and given as the first argument to the function that defines the unitary.

$$adderMod :: Int \rightarrow QInt \rightarrow QInt \rightarrow U$$

The next step is to use the modular addition unitary to define a modular multiplication unitary. The first classical argument is again the modulus of the function (N). The other classical input is one of the operands of the overall (modular) multiplication, and can be given classically as its constituent bits are used in classical if statements to control the addition (modulo N) of (quantum representations of) increasing powers of two to the second qubit register.

$$multMod :: Int \rightarrow Int \rightarrow QInt \rightarrow QInt \rightarrow U$$

This modular multiplication function has the requirement that the second qubit register is initialised to the state $|\vec{0}\rangle$. This cannot be achieved implicitly by a *letU* construct as the register isn't restored to this original value, and indeed it is actually this second register that encodes the result of the multiplication (modulo N) after the calculation has occurred.

To go on to construct the actual modular exponentiation function, we require that the modular exponentiation function can be controlled conditionally, depending on the state of a control qubit. We can achieve this very simply in *QIO* by using the *ifQ* unitary operation we defined previously.

$\mathit{condMultMod} :: \mathit{Qbit} \rightarrow \mathit{Int} \rightarrow \mathit{Int} \rightarrow \mathit{QInt} \rightarrow \mathit{QInt} \rightarrow U$

$\mathit{condMultMod}\ q\ n\ a\ x\ y = \mathit{ifQ}\ q\ (\mathit{multMod}\ n\ a\ x\ y)$

We can then go on to define a modular exponentiation unitary in terms of this (quantum) controlled modular multiplication unitary. The exponential unitary is again indexed by the modulus N as its first input. The second input is the base of the exponential, and can be given classically, as it is used as the basis for an iterated squaring method that is passed as the classical input to the multiplication (modulo N) unitary from above.

$\mathit{modExp} :: \mathit{Int} \rightarrow \mathit{Int} \rightarrow \mathit{QInt} \rightarrow \mathit{QInt} \rightarrow U$

The first quantum register must encode the exponent of the computation, and the second quantum register must be initialised to $|\vec{1}\rangle$, or in other words as a *QInt* representation of the integer one. It is this second register that will contain the result of the exponential after the calculation has occurred. The definition of this *modExp* function also makes use of a *letU* structure, which introduces another register in the state $|\vec{0}\rangle$, but which is returned to this state at the end by the design of the algorithm (This register is again given explicitly in the circuit representation in [VBE95] p.151).

With a library of reversible arithmetic functions, we can now go on to look at the final piece required before an implementation of Shor's algorithm can be defined. Looking back at the circuit given in figure 2.7 we can see that the last piece we need is an implementation of the inverse quantum Fourier transform.

6.4 Quantum Fourier transform

The QFT as introduced in section 2.6.5, is given by the circuit diagram in figure 2.6. We can notice that the structure is somewhat recursive with respect to the controlled R_k rotations. In fact, the structure seems to be inversely recursive, with the base case of the recursive structure (a single Hadamard rotation) acting on the last qubit in the register. As we can represent the qubit register in terms of a list of qubits, this *inverse* recursive structure lends itself nicely to an accumulative definition in Haskell, similar to the list reversing example given previously (in section 4.2). In fact, we can use our conditional construct over lists of qubits to define the QFT in terms of an accumulator function over a list of Boolean values.

```
qft :: [Qbit] → U
qft qs = condQ qs (λbs → qftAcu qs bs [])

qftAcu :: [Qbit] → [Bool] → [Bool] → U
qftAcu [] [] _                = •
qftAcu (q : qs) (b : bs) cs = qftBase cs q ▶ qftAcu qs bs (b : cs)

qftBase :: [Bool] → Qbit → U
qftBase bs q = f′ bs q 2
  where f′ []        q _ = uhad q
        f′ (b : bs) q x = if b then (rotK x q) ▶ f′ bs q (x + 1)
                                     else f′ bs q (x + 1)

rotK :: Int → Qbit → U
rotK k q = uphase q (1.0 / (2.0 ↑ k))
```

The accumulator function (*qftAcu*) ensures that each recursive call does the necessary controlled *rotK* rotations, by means of the recursive function *qftBase*.

Shor's algorithm, as we shall see in the following section, actually makes use of the inverse QFT. Fortunately, the inverse QFT is just the inverse of the QFT circuit as would be expected, and can be given in *QIO* by use of the *urev* function. Namely, the inverse QFT is given by (*urev qft*) :: [*Qbit*] → *U*.

```
shorU :: QInt → QInt → Int → Int → U
shorU k i1 q N = hadamards k ▶
                 modExp N a k q ▶
                 qftI k
```

Figure 6.2: Shor's algorithm in QIO: The unitary that represents the quantum part of Shor's algorithm, corresponding to the circuit diagram in figure 2.7.

6.5 Shor's algorithm in QIO

If we look look back at figure 2.7, we can see the essential building blocks for an implementation of Shor's algorithm, or at least the quantum period finding part of the algorithm. We have developed the necessary modular exponentiation function, which acts exactly as is required taking the input registers from the state $|k\rangle \otimes |\vec{1}\rangle$ to the state $|k, a^k modN\rangle$, where the value for N is the number to be factored, and the value a has been computed classically to be co-prime to N. We have also developed a unitary for the quantum Fourier transform on a register of qubits. So we can think of Shor's algorithm as acting on a quantum register representing an integer, we can just lift the inverse QFT as a function that acts on a *QInt*.

```
qftI :: QInt → U
qftI (QInt i) = urev (qft i)
```

The last piece we need to define is also the simplest, and just applies the Hadamard rotation to each of the qubits in a *QInt*.

```
hadamards :: QInt → U
hadamards (QInt []) = •
hadamards (QInt (q : qs)) = uhad q ▶ hadamards (QInt qs)
```

The full unitary that represents the quantum part of Shor's algorithm can now be constructed from these pieces, leading to definition given in figure 6.2.

To give the complete computation that represents the quantum part of Shor's algorithm, we must ensure that the two *QInt* inputs are initialised to the values $|\vec{0}\rangle$ and $|\vec{1}\rangle$ as required. The computation still takes N the number to be factorised, and a corresponding a value (co-prime to N) as its inputs, and the output is the value measured in the top register after the application of the unitary, this value is a candidate for the period of the given modular exponentiation function.

```
shor :: Int → Int → QIO Int
shor a N = do i0 ← mkQ 0
              i1 ← mkQ 1
              applyU (shorU i0 i1 a N)
              measQ i0
```

To create the actual function which uses this quantum algorithm to compute the factors of an input integer can now be defined classically. As not every value of a gives rise to a solution, the algorithm is probabilistic. This means we can use a random number generator to pick candidates for $a < N$ and use the euclidean algorithm to efficiently calculate the greatest common divisor of the two. In the case this is 1, we can use the value of a in the quantum part (if the greatest common divisor of a and N isn't 1, then this is in fact a factor of N). Once we have the period from the quantum part of the algorithm, we can do some classical post-processing to either find the factors of the input N, or have to repeat the quantum part of the algorithm again with a different value for the a candidate. The classical pre- and post-processing can easily be coded in Haskell to give the full factorisation function (*factor* :: *Int* → *QIO* (*Int*, *Int*)). The full implementation of such a function is given in the on-line source-code [Gre09], and we can already simulate a typical evaluation of running such a QIO computation. E.g. *run* (*factor* 15) gives $(5, 3)$.

Chapter 7

Implementing QIO in Haskell

Having introduced the *QIO* API, and gone through some examples of quantum computation written in *QIO*, we can now go on to look at the implementation. More specifically, we look at how the *run*, *runC*, and *sim* functions are realised, concentrating on how measurement gives rise to the use of a monadic structure. It is these simulator functions, most specifically the *run* function, that give a corresponding semantics to our *QIO* computations.

In theory, if we had a quantum platform on which to run our *QIO* computations then the *run* function would be much more efficient, but as it stands, we *simulate* the quantum behaviour using a classical random number generator. This classical simulation of quantum behaviour gives rise to large overheads in storing representations of quantum states (exponential in the number of qubits), and also of running our unitaries over such representations of quantum state (again exponential in the number of qubits).

To represent quantum states within a classical setting, we can follow the formalism that a quantum state can be described in terms of a sum over all its base states and their corresponding complex amplitudes. This can be generalised in Haskell by defining a **class** of *Heaps* that represent the base states, and a corresponding **class** of *Vectors* that represent a super-position of these base states, whereby each base state in the vector is assigned a complex amplitude. We have designed our vector as a class dubbed *VecEq*, where all the primitive operations on the vectors are implemented such that they keep the overall state of the vector normalised. This normalisation equates to summing the amplitudes of any equal base states, and as such, the number of heap structures within the vector will never be more than the number of base states required to fully describe the corresponding quantum state. Building the normalisation directly into the vector class follows from the fact that only types with definable equality can be normalised (such as our implementation of heaps) in this way. It is nice to note here that the *runC* function, that can only simulate the classical subset of *QIO*, only needs a single heap structure to represent the entire classical state as would be expected.

7.1 Heaps

We define the *Heap* class by

```
class Eq h ⇒ Heap h where
  initial :: h
  update :: h → Qbit → Bool → h
  (?) :: h → Qbit → Maybe Bool
  forget :: h → Qbit → h
  hswap :: h → Qbit → Qbit → h
  hswap h x y = update (update h y (fromJust (h ? x)))
                       x (fromJust (h ? y))
```

So, any data structure that provides the necessary functions can be used as a heap structure for our implementation of *QIO*. Defining heaps in terms of a class means that the actual underlying implementation can be changed if a more efficient data structure is found (although, as we'll see later, our use of Haskell's *Map* data type is efficient for the task at hand). We can go on now to look now at what the functions in the *Heap* class actually represent.

The first thing that a heap must provide is an element that represents an empty heap structure, in which no qubits have yet been initialised. This element is called the *initial* element, as it represents the initial state of a heap before anything has occurred.

The *update* function is used to accommodate the application of a *mkQbit* primitive. Given a current heap structure, a reference to a qubit, and the Boolean state in which to initialise this qubit, the update function returns a new heap, in which the referenced qubit is assigned the given Boolean value. If the qubit isn't already represented in the heap then it is added, as is the case when a *mkQbit* primitive occurs, but if the qubit is represented in the heap, then its value is updated to the given Boolean. This means that the *update* function can also be used when a rotation primitive occurs, as the only effect that a rotation can have on an individual base state is to apply the classical not function to one of the qubits in the base state.

A query function (?) must also be provided that simply returns the current value of the qubit being referenced. The *Maybe* monad is used here so that uninitialised qubits can have the state *Nothing*, and not give rise to an undefined heap. If a *Nothing* value occurs during the evaluation of a computation, then a suitable error must be thrown. The main case where an error of this sort occurs in practise is in a conditional where the control qubit appears as a variable in one of the branches. To accommodate the throwing of a run-time error whenever such a conditional occurs, the control qubit is temporarily *forgotten* from each base state using the provided *forget* function and hence, if it is used in one of the branches, the query function will fail, returning the error.

The *hswap* function returns the given heap, but with the values of its two argument qubits swapped around. This, in essence, swaps the positions of the two qubits with-in the heap, and is used whenever a *swap* primitive occurs in a *QIO* computation

As we previously mentioned, any data structure that provides an instance of the *Heap* class could be used as the basis for heaps in *QIO*. Our implementation makes use of Haskell's *Map* data type, which can be used to define a correspondence between Booleans and qubits in exactly the way our heaps require. The Map data type is an efficient implementation of key-value maps, and as such is a good candidate for our heaps. The operations for heaps can be directly translated into the primitive operations provided by the *Map* type.

```
type HeapMap = Map.Map Qbit Bool
instance Heap HeapMap where
  initial = Map.empty
  update h q b = Map.insert q b h
  h ? q = Map.lookup q h
  forget h q = Map.delete q h
```

As the classical simulator function *runC* only requires a single heap structure to represent the entire state of the system, we can already define it by translating each of the *QIO* primitives into their corresponding primitives on the *HeapMap* data-type. In fact, the system must also keep track of the next available qubit so that a *mkQbit* primitive is guaranteed to return a reference to a new qubit, meaning that we can define a classical state by the type *StateC* as follows.

```
data StateC = StateC {freeC :: Int, heap :: HeapMap}
```

It is the pure nature of all these heap operations that also means we can give the *runC* classical simulator as a pure function.

We move on now to look at our vector structures that are used by the *QIO* simulator functions *run* and *sim* to represent fully quantum states. These functions translate the *QIO* primitives into the primitive functions available for our *VecEq* structures, which, as we'll also see, in many cases corresponds to mapping the underlying heap primitives over every heap in the vector.

7.2 Vectors

To hold our quantum states we define a **class** of vectors. As we mentioned, we have designed these vectors such that all the primitive operations available for them, keep the overall state of the vector normalised. This normalisation equates to combining equal base state terms by summing their complex amplitudes (an action that can lead to deconstructive interference, as is expected).

The vectors we define should be able to be an instance of the *Monad* type-class in Haskell, such that it is possible to define the semantics of our monadic *QIO* computations in terms of monadic operations acting on these vectors. We shall look at how this is achieved after introducing the *VecEq* class of vectors. The requirement that our underlying type in the vectors has definable equality causes a few problems with this monadic definition, but we will see how we can make use of a trick suggested in [Sit08] to achieve such a definition.

The *VecEq* class is given by

```
class VecEq v where
  vzero :: v x a
  (⊕) :: (Eq a, Num x) ⇒ v x a → v x a → v x a
  (⊗) :: (Num x) ⇒ x → v x a → v x a
  (@) :: (Eq a, Num x) ⇒ a → v x a → x
  fromList :: [(a, x)] → v x a
  toList   :: v x a → [(a, x)]
```

and we can now go over each of these class functions to describe the actions they must define. As an aside, these vectors can have any numeric type x to represent the amplitudes, and be over any type a with definable equality. To define quantum computation we use them with the complex numbers, and over heaps, and as such often use the terminology associated with this specific case.

The *vzero* element is just an empty vector, which is used in a similar way to the *initial* element of the *Heap* class, that is to describe the state of an empty quantum system before any qubits have been initialised.

The ($\oplus$) operation is used to combine two vectors, and it is specifically this operation that keeps the vectors normalised. In combining two vectors, we require that the underlying type the vector holds has a definable equality, that is, that it is a member of the type-class *Eq*. When combining two vectors, this operation will simply sum the amplitudes of any underlying base states that are present in both input vectors.

The ($\otimes$) operation is a form of scalar multiplication for vectors, and is used to multiply all the complex amplitudes in a vector by the given scalar.

The (@) function is a kind of look-up function, that corresponds to finding the current amplitude of a given base state within the overall state represented by the vector.

The *fromList* and *toList* functions are simply helper functions that are used in defining the *VecEq* class as a monadic structure. In fact, the implementation we have used for an instance of the *VecEq* class is simply a data-type that represents a vector as a list of pairs, and these operations are simple to define (as the constructor and deconstructor elements of that data-type).

```
data VecEqL x a = VecEqL { unVecEqL :: [(a, x)] } deriving Show
```

The *vzero* element of such a list based vector is just the empty list.

```
vEqZero :: VecEqL x a
vEqZero = VecEqL []
```

The ($\oplus$) function (given by *vEqPlus*) can be given using a fold operation that steps through the elements in the second argument vector, and uses the *add* function to check if the current element is already a member of the first vector. If this is the case, then the corresponding amplitudes are added, but if this isn't the case then the element is concatenated onto the end of the first vector.

```
vEqPlus :: (Eq a, Num x) ⇒
               VecEqL x a → VecEqL x a → VecEqL x a
(VecEqL as) `vEqPlus` vbs = foldr add vbs as
add :: (Eq a, Num x) ⇒ (a, x) → VecEqL x a → VecEqL x a
add (a, x) (VecEqL axs) = VecEqL (addV′ axs)
  where addV′ [ ] = [(a, x)]
        addV′ ((by@(b, y)) : bys) | a ≡ b = (b, x + y) : bys
                                  | otherwise = by : (addV′ bys)
```

The (⊗) function (given by *vEqTimes*) simply maps the multiplication operation by the given scalar to each complex amplitude in the vector, and the (@) function (given by *vEqAt*) steps through the vector until the matching base state is found. If the base state is not found in the vector then then an amplitude of zero is returned.

```
vEqTimes :: (Num x) ⇒ x → VecEqL x a → VecEqL x a
l ‘vEqTimes‘ (VecEqL bs) | l ≡ 0 = VecEqL [ ]
                         | otherwise
                           = VecEqL (map (λ(b, k) → (b, l ∗ k)) bs)
vEqAt :: (Eq a, Num x) ⇒ a → VecEqL x a → x
a ‘vEqAt‘ (VecEqL [ ]) = 0
a ‘vEqAt‘ (VecEqL ((a′, b) : abs)) | a ≡ a′ = b
                                   | otherwise
                                     = a ‘vEqAt‘ (VecEqL abs)
```

These definitions can now be used to give *VecEqL* as an instance of the *VecEq* class.

```
instance VecEq VecEqL where
  vzero = vEqZero
  (⊕) = vEqPlus
  (⊗) = vEqTimes
  (@) = vEqAt
  fromList as = VecEqL as
  toList (VecEqL as) = as
```

The requirement that the underlying type is a member of *Eq* leads to a problem that we can not define *VecEq* as a monad, which we require so we can sequence the monadic *QIO* operations over it. We follow the technique suggested in ([Sit08]), but for restricting the monad by means of an instance of *Eq* instead of *Ord*. We can define an *EqMonad* class, whose members correspond to the restricted monad operations we wish to define.

```
class EqMonad m where
  eqReturn :: Eq a ⇒ a → m a
  eqBind :: (Eq a, Eq b) ⇒ m a → (a → m b) → m b
```

We can't just use this definition in place of a monad as it would prevent us from being able to use **do** notation. We can however define *VecEq* as a member of this class, with the functions corresponding to the monadic behaviour we require.

```
instance (VecEq v, Num x) ⇒ EqMonad (v x) where
  eqReturn a = fromList [(a, 1)]
  eqBind va f = case toList va of
    ([]) → vzero
    ((a, x) : []) → x ⊗ f a
    ((a, x) : vas) → (x ⊗ f a) ⊕ ((fromList vas) ‘eqBind‘ f)
```

In order to define *EqMonad* as a form of monad in Haskell, we need to *embed* it in a monadic structure that can be defined by the *AsMonad* data-type.

```
data AsMonad m a where
  Embed :: (EqMonad m, Eq a) ⇒ m a → AsMonad m a
  Return :: EqMonad m ⇒ a → AsMonad m a
  Bind   :: EqMonad m ⇒
                AsMonad m a → (a → AsMonad m b) → AsMonad m b
instance EqMonad m ⇒ Monad (AsMonad m) where
  return = Return
  (≫=) = Bind
```

The overall *AsMonad* structure is indeed an instance of the *Monad* class in Haskell, and as such can be used with the **do** notation as we required. However, to be actually able to use our *embedded* instance of the *EqMonad* we need a function *unEmbed*, which as the name suggests unembeds the *EqMonad* structure.

$$\begin{aligned}
&\mathit{unEmbed} :: \mathit{Eq}\ a \Rightarrow \mathit{AsMonad}\ m\ a \rightarrow m\ a\\
&\mathit{unEmbed}\ (\mathit{Embed}\ m) = m\\
&\mathit{unEmbed}\ (\mathit{Return}\ a) = \mathit{eqReturn}\ a\\
&\mathit{unEmbed}\ (\mathit{Bind}\ (\mathit{Embed}\ m)\ f) = m\ `\mathit{eqBind}`\ (\mathit{unEmbed} \circ f)\\
&\mathit{unEmbed}\ (\mathit{Bind}\ (\mathit{Return}\ a)\ f) = \mathit{unEmbed}\ (f\ \ a)\\
&\mathit{unEmbed}\ (\mathit{Bind}\ (\mathit{Bind}\ m\ f)\ g) = \mathit{unEmbed}\\
&\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{Bind}\ m\ (\lambda x \rightarrow \mathit{Bind}\ (f\ \ x)\ g))
\end{aligned}$$

This gives us a fully monadic implementation of the *VecEq* type of vectors, and as such we can now go on to look at how the primitive *QIO* operations can be translated into the primitive operations of the *VecEq* type.

7.3 Evaluating QIO computations

So we now have a structure *HeapMap* that can be used to represent the base states in the description of a quantum state, and a structure *VecEqL* that can be used to define a sum of these base states, along with their corresponding amplitudes. As such, we can now define the specific instance of the *VecEqL* type that we shall be using to represent the *pure* quantum states in our computations. Specifically, a *Pure* state is described by a *VecEqL* structure over the *HeapMap* structure, along with complex numbers $\mathbb{C}$ to represent the amplitudes as would be expected.

$$\textbf{type}\ \mathit{Pure} = \mathit{VecEqL}\ \mathbb{C}\ \mathit{HeapMap}$$

We can now think of a unitary operator in terms of the effect it has on each base state within a pure state. In fact, we can define the type *Unitary* to represent our unitary operators in such a way. The extra *Int* argument represents the next available qubit, just as we needed to keep track of it for the classical simulator.

$$\textbf{data}\ \mathit{Unitary} = U\ \{\mathit{unU} :: \mathit{Int} \rightarrow \mathit{HeapMap} \rightarrow \mathit{Pure}\}$$

When evaluating the application of a *QIO* unitary (in U), these new *Unitary* functions are mapped over each base state in the whole pure state of the system.

The results of this mapping are then joined using the underlying $\oplus$ operation defined for the *Pure* data-type. The approach we take to evaluate our *QIO* computations in this way is to define a member of the *Unitary* data-type for every corresponding primitive member of the *U* datatype. So that the monoidal behaviour of the *U* data-type can also be modelled in this way, we must define a monoidal behaviour for the *Unitary* data-type. This monoidal behaviour makes use of the underlying (*embedded*) monadic behaviour of the *Pure* data-type.

```
instance Monoid Unitary where
  • = U (λfv h → unEmbed (return h))
  (U f) ▶ (U g) = U (λfv h → unEmbed (do h′ ← Embed (f fv h)
                                          h″ ← Embed (g fv h′)
                                          return h″))
```

With all the structure in place, we can now define the actual function *runU*, which lifts members of the *U* data-type to their corresponding *Unitary* function.

```
runU :: U → Unitary
runU UReturn = •
runU (Rot x a u) = uRot x a ▶ runU u
runU (Swap x y u) = uSwap x y ▶ runU u
runU (Cond x us u) = uCond x (runU ∘ us) ▶ runU u
runU (Ulet b xu u) = uLet b (runU ∘ xu) ▶ runU u
```

The functions *uRot*, *uSwap*, *uCond* and *uLet* actually convert each of the underlying *U* type into a *Unitary*. For example, the *uRot* function is as follows:

```
uRot :: Qbit → Rotation → Unitary
uRot q r = (uMatrix q (r (False, False),
                       r (False, True),
                       r (True, False),
                       r (True, True)))
uMatrix :: Qbit → (ℂ, ℂ, ℂ, ℂ) → Unitary
uMatrix q (m00, m01, m10, m11) = U (λfv h → (
if (fromJust (h ? q))
then (m01 ⊗ (unEmbed (return (update h q False))))
        ⊕ (m11 ⊗ (unEmbed (return h)))
else  (m00 ⊗ (unEmbed (return h)))
        ⊕ (m10 ⊗ (unEmbed (return (update h q True))))))
```

We can look now at defining how the other monadic constructors of *QIO* are dealt with during evaluation. In order to do this, we must first define a data structure that can be used to keep track of the state of our system. This data structure (*StateQ*) consists of a member of the *Pure* data-type, that corresponds to the current quantum state, and an integer that represents the next free qubit. We again must keep track of the next available qubit so that the system can assign a new qubit when necessary.

```
data StateQ = StateQ {free :: Int, pure :: Pure}
```

With our *StateQ* structure defined, we can look at how the monadic constructors of *QIO* are evaluated. As we can now think of this evaluation as a stateful computation, we are able to use Haskell's *State* monad, as was introduced in section 4.4.1, to define our evaluation functions. We will look shortly at what the *PMonad* class requirement is, but for now we can just think of it as providing a *merge* function that describes how measurements and results are presented back to the user, and we'll see how the two instances of *PMonad* we provide relate to the only differences between the *sim* and *run* functions. This evaluation function is called *evalWith* as a state is stored implicitly by the use of the *State* monad, and the *evalWith* function can be thought of as evaluating the given *QIO* computation with that state.

```
evalWith :: PMonad m ⇒ QIO a → State StateQ (m a)
```

The *QReturn* primitive is just the *return* of the monad.

```
evalWith (QReturn a) = return (return a)
```

The *MkQbit* qubit primitive updates the overall state by increasing the free variable, and updating the pure state with a qubit initialised in the state corresponding to the given Boolean

```
evalWith (MkQbit b g) = do (StateQ f p) ← get
                           put (StateQ (f + 1)
                             (updateP p (Qbit f) b))
                           evalWith (g (Qbit f))
```

The *ApplyU* primitive uses the *runU* function as described above, and updates the current state accordingly.

```
evalWith (ApplyU u q) = do (StateQ f p) ← get
                           put (StateQ f (unEmbed (
                                             do x ← Embed (p)
                                                x' ← Embed (uu f x)
                                                return x'
                              )))
                           evalWith q
                             where U uu = runU u
```

The *Meas* primitive is evaluated by *splitting* the *Pure* state about the given qubit, creating two *Pure* states that represent the state of the system for each of the possible measurement outcomes. The computation then proceeds over both of these *Pure* states, giving a pair of results. The *merging* of these two results is left to the *merge* function of the underlying *PMonad*. We shall look at the definition of a *PMonad* shortly, and how this *merge* function defines the behaviour of the overall evaluation.

```
evalWith (Meas x g) = do (StateQ f p) ← get
                         (let Split pr ift iff = split p x
                           in if pr < 0 ∨ pr > 1
                             then error "pr < 0 or >1"
                             else do put (StateQ f ift)
                                     pift ← evalWith (g True)
                                     put (StateQ f iff)
```

$$piff \leftarrow evalWith\ (g\ False)$$
$$return\ (merge\ pr\ pift\ piff))$$

We've seen how the behaviour of the evaluation function depends on the *merge* function provided by an underlying *PMonad* structure. In fact, we define a *PMonad* as just a *Monad* along with this extra *merge* function.

$$\textbf{class}\ Monad\ m \Rightarrow PMonad\ m\ \textbf{where}$$
$$merge :: \mathbb{R} \rightarrow m\ a \rightarrow m\ a \rightarrow m\ a$$

The *merge* function can be thought of as a function that defines how two computations of the same type can be combined, depending on a real argument to that function. This real argument represents the probability of each of the computations occurring, and as such, we are able to define instances of the *PMonad* that give rise to the two different behaviours of our simulation functions (*sim* and *run*). In fact, to be more specific, the real argument represents the probability of the first computation occurring (p), and the probability of the second computation occurring can be calculated as $1 - p$.

For the *run* function, we define the *IO* monad as an instance of a *PMonad*. To do this, we define the *merge* function to make use of the *IO* monad's random number generator to choose one of the two *merged* computations to perform, probabilistically depending on the given probability.

$$\textbf{instance}\ PMonad\ IO\ \textbf{where}$$
$$merge\ pr\ ift\ iff = Random.randomRIO\ (0, 1.0)$$
$$\ggg\!= \lambda pp \rightarrow \textbf{if}\ pr > pp\ \textbf{then}\ ift\ \textbf{else}\ iff$$

For the *sim* function however, we define a type constructor that represents a probability distribution over any type a. This type constructor is similar to the *VecEq* type constructor introduced for quantum states, but doesn't have the requirement that the underlying type is a member of *Eq*, and hence doesn't keep the vectors normalised in the same way.

$$\textbf{data}\ Prob\ a = Prob\ \{unProb :: Vec\ \mathbb{R}\ a\}$$

The primitive operations available on the *Vec* type constructor are pretty much the same as for the *VecEq* type constructor, just without the normalisation behaviour. The $\mathbb{R}$ values that correspond to each member of the vector are the probabilities of each member in the distribution. So that we can use this structure in our *merge* operation, we must define this *Prob* type constructor as an instance of the *PMonad* class, and this first entails that we first define it as an instance of the *Monad* class.

$$\begin{aligned}
&\textbf{instance}\ \mathit{Monad}\ \mathit{Prob}\ \textbf{where}\\
&\quad \mathit{return} = \mathit{Prob} \circ \mathit{return}\\
&\quad (\mathit{Prob}\ \mathit{ps}) \gg\!= f = \mathit{Prob}\ (\mathit{ps} \gg\!= \mathit{unProb} \circ f)
\end{aligned}$$

We can now define the corresponding *merge* operation, and note how it gives rise to the probability distributions when used by the *sim* function.

$$\begin{aligned}
&\textbf{instance}\ \mathit{PMonad}\ \mathit{Prob}\ \textbf{where}\\
&\quad \mathit{merge}\ \mathit{pr}\ (\mathit{Prob}\ \mathit{ift})\ (\mathit{Prob}\ \mathit{iff}) = \mathit{Prob}\ ((\mathit{pr} \otimes \mathit{ift})\\
&\qquad\qquad\qquad\qquad\qquad\qquad \oplus ((1 - \mathit{pr}) \otimes \mathit{iff}))
\end{aligned}$$

The *merge* function multiplies each of the underlying probability distribution arguments by their respective probabilities, and then joins these to create the new probability distribution, as gained by the result of a measurement.

Lazy evaluation is specifically useful here as if the *merge* function only requires one of the *Pure* states (as is the case for the *run* function), then the other *Pure* state is never evaluated.

We can finish this section by giving the final definitions for the *run* and *sim* functions. In fact, they both call the same *eval* function (which calls *evalWith* over an empty initial state). The types given for each of the simulator functions inform the type-system which *PMonad* should be used in the evaluation of the *QIO* computations.

$$\begin{aligned}
&\mathit{eval} :: \mathit{PMonad}\ m \Rightarrow \mathit{QIO}\ a \rightarrow m\ a\\
&\mathit{eval}\ p = \mathit{evalState}\ (\mathit{evalWith}\ p)\ \mathit{initialStateQ}\\
&\mathit{run} :: \mathit{QIO}\ a \rightarrow \mathit{IO}\ a\\
&\mathit{run} = \mathit{eval}
\end{aligned}$$

$sim :: QIO\ a \rightarrow Prob\ a$

$sim = eval$

This chapter has so far introduced an implementation of the *QIO* monad in Haskell. The next section goes on to discuss some of the pitfalls of this implementation, and lays out the case for reimplementing *QIO* in a stronger type-system, as is the case in chapter 9.

7.4 Remarks on QIO in Haskell

This implementation of *QIO* in Haskell provides a monadic interface to quantum programming from Haskell. As it stands, it is possible to create quantum algorithms in terms of unitary structures away from the monadic interface, and then define quantum computations as instances of these algorithms, along with initialisations and measurements of the necessary quantum data-types. This monadic interface should be more natural to functional programmers, who are used to writing monadic structures, such as computations in the *IO* monad, and as such should provide a nice footstep into the world of quantum programming too. In defining the monadic structure, along with the *simulator* functions, we have borrowed techniques from works such as [Swi08, SA07], which also helps us to reason about our monadic programs because of the constructive semantics assigned to their evaluation. However, as it stands, their are some definable computations in *QIO*, or more specifically unitary structures in *U* that don't actually represent a unitary semantics, and as such have to be caught at run-time by the evaluation functions. This behaviour is very bad in the long term, as we are using the classical properties of our simulator functions to catch these errors. In an actual quantum system, these type of errors could not necessarily be caught in the same way as there is no access to the state of the system (as measurements would disrupt the quantum state). As such, we cannot use such an implementation of *QIO* to reason about the quantum algorithms written in it, and it would make sense to look at ways in which we could use a stronger type system to achieve this.

The specific problems that can occur in this implementation are un-unitary versions of each of the *urot*, *cond*, and *ulet* constructors. The *urot* construct doesn't check the unitarity of its argument matrix, the *ulet* construct has the side condition that any temporary qubits must be set back to their initial state, and the *cond* has to ensure that the control qubits are *separable* from the qubits used in its sub-unitary expression. For instance, a valid member of the *QIO* unitary syntax (U) would be the following *notUnitary* structure.

$\mathit{notUnitary} :: U$

$\mathit{notUnitary} = \mathit{cond}\ q\ (\lambda x \rightarrow \textbf{if}\ x\ \textbf{then}\ \mathit{unot}\ q\ \textbf{else}\ \bullet)$

This doesn't define a unitary operation as the qubit q is always set to the state $|0\rangle$. This error is caught at run-time, as the qubit q is clearly not separable from the sub-unitary, as it is used in the sub-unitary. However, it would be more practical, and make more sense computationally, if such an un-unitary structure wasn't a valid member of the syntax.

Dependent types are one solution to this problem and we shall look in chapter 9 at a reimplementation of *QIO* in such a dependently typed language (specifically Agda). There has been work on *faking* dependent types in Haskell [McB02], which would mean we still have all the abstractions of Haskell at hand, but it makes more sense to use a system where the dependent type-system has been specifically designed for such a purpose. The next chapter (chapter 8) introduces Agda, and shows how dependent types can be used to give a formal verification of the specification of our programs. We'll also see how we can use such a type system to ensure that the new version of the U data-type is able to ensure the unitarity of the structures within it. In this sense, we are moving slightly away from functional programming, and into the realms of Type Theory, which also gives us a more formal semantics of our quantum computations, and thus gives us more power to reason about quantum algorithms written in *QIO*.

Another underlying design factor of the *QIO* monad is its relation to the gate model of quantum computation. In fact, most of this thesis has built upon the gate model from introducing in it chapter 2, modelling it categorically in chapter 3, to the design of the unitaries in *QIO*. As we have also seen previously in chapter 2, there are other models of quantum computation, although the gate model is still the most widely used. It would be possible to provide a different semantics for *QIO* in many of these models, and as recent research has been showing that the one-way model of quantum computation [BB06] could provide a more realistic way for creating a *real world* quantum computer, one such model we would like to model is a semantics based on the measurement calculus [DKP07]. Although such work is not looked at in this thesis, it would be one direction in which further research into the area could proceed.

Chapter 8

Dependent Types

We have now introduced the implementation of an interface to quantum programming (*QIO*) from the functional programming language Haskell, and mentioned how moving to a setting with a stronger type system, such as a dependently typed system, could be used to overcome some of the pitfalls of the implementation, namely the need for run-time errors. We go on to look now at such a reimplementation of the quantum IO monad in a dependently typed setting, but shall first introduce the concept of dependent types and the language Agda.

Agda [Nor07] is a dependently typed language influenced heavily by Haskell and the dependently typed language Epigram [McB99]. It defines itself as a dependently typed language able to be used to define inductive families that are indexed by values, and not just by types. It can also be thought of as a proof assistant as it is based on intuitionistic type theory [ML84].

The following section aims to introduce the reader to the language with some simple examples of dependent types. More information on Agda is available online from the Agda wiki `http://wiki.portal.chalmers.se/agda/`. The wiki also contains links to other more in depth tutorials on the language, and to a standard library containing many useful data-types and their corresponding functions.

After introducing Agda, we shall go on to look at how we can use Agda as a proof assistant, and develop code that can be thought not just as a program, but also as a proof of its own specification. This formalism comes from the types given by Per Martin-Lőf's intuitionistic type theory [ML84], and we go on to give examples of how reversible programs can be specified in Agda, which are also proofs of their own reversibility. The last section of this chapter also shows how despite Agda not having a type-class system akin to Haskell's, we can define monoids and monads in Agda in the form of type constructors.

8.1 An introduction to Agda

Agda allows us to define data types as inductive families, so a simple data type to define is that of the natural numbers. Agda allows the use of Unicode, so we are able to define the type of natural numbers as the type ℕ.

```
data ℕ : Set where
  zero : ℕ
  suc : ℕ → ℕ
```

This definition of the natural numbers states that zero is a natural number, and given any other natural number n, we are able to create another natural number, namely the successor of n, or suc n. This definition of the natural numbers is just the Peano definition. We can could now define the natural numbers one, two, and three as follows:

```
one : ℕ
one = suc zero
two : ℕ
two = suc one
three : ℕ
three = suc two
```

It is important to note here, that as the type system of Agda is stronger than the type system of Haskell, it is necessary for the type of a function to always be given by the programmer.

Once a type is defined it is possible to define functions over these types in a very similar manner as in Haskell. For example, we can define the addition function acting on natural numbers.

```
_+_ : ℕ → ℕ → ℕ
zero + n = n
suc m + n = suc (m + n)
```

In defining a function, Agda allows the programmer to use the underscore character to enable the creation of in-line functions. Each underscore character that appears in the name of a function is treated as the position of an argument to that function. Agda is also a total language, meaning that the functions defined must be total with respect to the input types. In essence, this means that the definition of a function in Agda must include a definition clause for every valid element of each of the input types specified by the functions type signature.

So, we have created the type of natural numbers, but this isn't a dependent type and could have been defined in a very similar manner in Haskell. It is how we can use values in the definition of data types that makes Agda a dependently typed language, so we can now define the type of vectors indexed by their length.

```
data Vec (a : Set) : ℕ → Set where
  [] : Vec a zero
  _::_ : ∀ {n} (x : a) (xs : Vec a n) → Vec a (suc n)
```

That is, a vector over any arbitrary type a is either the empty vector ([]) of length zero, or given any element x of type a and a vector over type a of length n we can create a new vector over type a with length suc n. Compare this to how we defined lists in Haskell, the only difference is that these vectors are indexed by their length. This definition shows how we can use implicit arguments in the definition of functions, that the type checker is able to infer at compile time. In the definition above of the _::_ constructor, the argument n is an implicit argument as the type-checker is able to infer it directly from the type of the argument vector xs, and as such is given in the type of the function surrounded by curly brackets.

With a dependently typed data structure, we can create functions that only take certain members of the type as arguments. For example, it does not make sense to ask for the head element of an empty vector. In Haskell, we would have to return an error, but in Agda we are able to define our head function such that only non-empty vectors can be passed to it as an argument.

```
head : ∀ {a n} → Vec a (suc n) → a
head (x :: xs) = x
```

The type of the head function states that the input vector must have length of at least the successor of any natural number (n : ℕ), meaning that the empty vector isn't a valid argument to the function. We can also use previously defined functions to act on the values within the types in the type-signature of a new function definition. For example, the concatenation function acting on two vectors of length m and n, will return a vector of length m + n.

```
_++_ : ∀ {a m n} → Vec a m → Vec a n → Vec a (m + n)
[] ++ ys = ys
(x :: xs) ++ ys = x :: (xs ++ ys)
```

Another interesting type that can be defined in dependently typed programming languages is that of a type indexed by how many elements it contains. That is a finite set of n elements for any n : ℕ. We can do this in Agda quite simply using our definition of ℕ.

```
data Fin : ℕ → Set where
  zero : ∀ {n : ℕ} → Fin (suc n)
  suc : ∀ {n : ℕ} → (i : Fin n) → Fin (suc n)
```

zero is an element of all finite sets, and given an element i in a finite set of size n, suc i is in the finite set of size suc n. The Fin data-type is useful if we want to restrict the size of an argument to a function. For example, it is possible to think of members of Fin n as members of the natural numbers less than n, (and Agda allows us to re-use constructor names) and hence can define a subtraction function that ensures its second argument is less than the first.

```
_-_ : (n : ℕ) → Fin n → ℕ
zero - ()
suc y - zero = suc y
suc y - suc i = y - i
```

The () is a dummy argument that the type-checker uses to state that there is no valid argument to the function, or more precisely in this specific case that the type Fin zero is uninhabited.

Another useful function that makes use of the Fin data-type to restrict the input arguments is the lookup function for vectors.

```
lookup : ∀ {a n} → Fin n → Vec a n → a
lookup zero (x :: xs) = x
lookup (suc i) (x :: xs) = lookup i xs
```

The Fin argument in this case ensures that the given index is indeed a valid position in the vector, and the function returns the element of the vector at that position.

Defining functions in Agda is achieved using pattern matching in a very similar manner as it is used in Haskell. However, it also provides what is known as the **with** construct to allow functions to be defined by pattern matching over intermediate results. A nice example of this is the filter function which acts on lists. The filter function creates a sublist of the given list, of all elements from the original list that fulfil some given Boolean function.

```
data List (a : Set) : Set where
  [] : List a
  _::_ : a → List a → List a

filter : ∀ {a : Set} → (f : a → Bool) → List a → List a
filter f [] = []
filter f (x :: xs) with f x
... | true = x :: filter f xs
... | false = filter f xs
```

The **with** clause evaluates the Boolean function over the head (x) of the list and then pattern matches over the result of this evaluation to either leave x in the returned list or drop it from the returned list. The recursive call is again used to traverse the rest of the list.

An important construct in dependently typed languages are what are known as Σ-types. These are often referred to as dependent pairs, but can be generalised to tuples of any length. A Σ-type can be thought of as a pair such that the type of the second value of the pair depends on the first value of the pair. For example, this can be useful if we want to know properties about a specific type as we can construct a Σ-type whose first member is of the given type, and whose second member is a proof of some property of that member of the type. The following section (8.2) goes into more details of how proofs can be constructed in Agda, and will also give some examples of how Σ-types can be used.

```
data Σ (A : Set) (B : A → Set) : Set where
  _,_ : (x : A) (y : B x) → Σ A B
```

The type definition clearly shows that the first element of the pair can be in any arbitrary type A, but that the type of the second argument can depend on the value x : A. We can also define a simple non-dependent product as a member of the Σ-type by passing a dummy argument to the second type.

```
_×_ : (A B : Set) → Set
A × B = Σ A (λ _ → B)
```

It is often useful to have projection functions for these dependent pairs, which simply return the element required.

```
proj₁ : ∀ {A B} → Σ A B → A
proj₁ (x, y) = x
proj₂ : ∀ {A B} → (p : Σ A B) → B (proj₁ p)
proj₂ (x, y) = y
```

8.2 Proofs in Agda

We briefly mentioned how Agda can be thought of as a proof assistant as it is based on Per Martin-Löf's intuitionistic type theory. Proofs can be constructed by defining a type which defines some property that we wish to prove, and then if we can construct an element of that type we have our proof. Equality types are a relatively simple, but also powerful construct that we can use in Agda to construct proofs of equality between elements of a data-type. The equality type is defined as a type with only one constructor, namely the proof of reflexivity, that every element of any arbitrary type is equal to itself.

```
data _≡_ {a : Set} (x : a) : a → Set where
  refl : x ≡ x
```

If we want to prove that two values are equal then we need to construct an element of the _≡_ data type, that the type-checker of Agda is able to normalise down to a proof of reflexivity. For example, we can look at our function for addition over the natural numbers from the previous section and construct an element of the equality type that is a proof that the function is commutative, that is, given two arbitrary natural numbers m and n we can construct a member of the type $(m + n) \equiv (n + m)$.

```
+-identity : ∀ {n} → n + zero ≡ n
+-identity {zero} = refl
```

```
+-identity {suc n}  =  cong suc +-identity

m+1+n≡1+m+n  :  ∀ m n  →  m + suc n ≡ suc (m + n)
m+1+n≡1+m+n zero n  =  refl
m+1+n≡1+m+n (suc m) n  =  cong suc (m+1+n≡1+m+n m n)

+-comm  :  ∀ m n  →  m + n ≡ n + m
+-comm zero n  =  sym +-identity
+-comm (suc m) n  =  trans (cong suc (+-comm m n))
                           (sym (m+1+n≡1+m+n n m))
```

The definition uses helper functions, which are actually proofs of other equality types. We use the transitivity and symmetry of the equality type, and the congruence of the equality type over function application.

```
trans  :  ∀ {m n o}  →  m ≡ n  →  n ≡ o  →  m ≡ o
sym  :  ∀ {m n}  →  m ≡ n  →  n ≡ m
cong  :  ∀ {A B m n}  →  (f : A → B)  →  m ≡ n  →  f m ≡ f n
```

In the proofs given, we use this congruence over the successor constructor, i.e. that a proof of m ≡ n can be lifted to a proof of suc m ≡ suc n. The +-identity function is a proof that zero is a right identity of addition (zero is also a left identity of addition and the proof of this is trivial due to the definition of _+_). The m+1+n≡1+m+n function is also a proof of the equivalence given by its type, and is used along with the trans function to in essence move a successor construct between either side of an addition.

The given proof of the commutativity of the addition function over natural numbers is quite a simple proof, but it is not necessarily obvious how to construct it from the given helper functions. Agda comes to the aid here allowing holes to be left in the definition of functions, so functions (and hence proofs) can be constructed in a methodological manner. At any time, the type-checker is able to tell you the type required to fill the current hole, along with the variables currently in scope.

We shall move on now to look at how monoids and monads can be defined in Agda, and then shall look at how reversible computations in Agda can make use of these proof structures we have just seen to provide a formally verified proof of their reversibility.

8.3 Monoids and Monads in Agda

In the next chapter (chapter 9) we go on to define a new version of the Quantum IO Monad in Agda using some of the techniques and constructs we have introduced above. It is useful to note that the notion of Monoids and Monads translate very nicely from their Haskell counterparts. This section shall just give a quick overview of Monads and Monoids in Agda, and shows how despite Agda not having type-classes we are easily able to use Agda's **record** type to define them. Agda's **record** data-type can be thought of a generalisation of the Σ type, as the type of each **field** can depend on the types of the previous **field**s in the **record**.

```
record Monoid  :  Set₁ where
  field
    carrier   : Set
    _≈_       : carrier → carrier → Set
    _•_       : carrier → carrier → carrier
    ϵ         : carrier
    isMonoid  : IsMonoid _≈_ _•_ ϵ
```

A Monoid in Agda is a record that contains the five fields given in its definition above. The carrier field defines the underlying Set or data type over which we would like to define our monoid. The _≈_ field must define an equivalence relation over the carrier set. The _•_ field defines the binary operation of the monoid, which was denoted by mappend in the Haskell implementation, and the ϵ field defines the identity element for the given binary operation which corresponds to the mempty element of the Haskell implementation. The last field, isMonoid, corresponds to a data-type that encodes the monoid laws using the given equivalence relationship. We can look at this in more detail by giving its definition as another **record** type.

```
record IsMonoid {A} (_≈_ : A → A → Set)
                    (_•_ : A → A → A)
                    (ϵ : A) : Set where
  field
    refl      : ∀ {x} → x ≈ x
    sym       : ∀ {i j} → i ≈ j → j ≈ i
    trans     : ∀ {i j k} → i ≈ j → j ≈ k → i ≈ k
    assoc     : ∀ x y z → ((x • y) • z) ≈ (x • (y • z))
    •-pres-≈  : ∀ {x y u v} → x ≈ y → u ≈ v → x • u ≈ y • v
    identity  : (∀ {x} → (ϵ • x) ≈ x) × (∀ {x} → (x • ϵ) ≈ x)
```

The first three fields prove that the given relation is indeed an equivalence relation. refl is a proof that ≈ is reflexive, sym is a proof that ≈ is symmetric, and trans is a proof that ≈ is transitive. The next two fields prove that the carrier set, along with the • operation form a semi-group. That is, that • is associative, and that • preserves the given equivalence relationship. The final field proves that the given element ϵ is indeed a right- and left-identity to the given • operation. In fact, Agda's standard library splits the laws into data-types that prove these sub structures (The IsEquivalence and IsSemigroup data-types respectively). The following section (section 8.4) goes on to look at a reimplementation of our toy language of reversible circuits written in Agda, which uses two instances of a monoid as we have defined. We shall now look at how we can also define monads in Agda in a similar way as we have defined monoids above.

The standard library of Agda currently only contains an implementation of what it calls raw monads. Raw monads are an implementation of monads that don't contain proofs of the monad laws. It would be possible to define a type of monad in Agda that encodes the proofs of the monad laws, although in this thesis I shall just be using the raw monads as defined in the standard library. The raw monad actually makes use of an underlying indexed raw monad, that is indexed by the unit type, which carries no extra information. It useful to give the definition of a raw indexed monad here.

A record type is once again used for the definition of a raw indexed monad.

```
record RawIMonad {I : Set} (M : I → I → Set → Set) : Set₁ where
  field
    return : ∀ {i A} → A → M i i A
    _»=_ : ∀ {i j k A B} → M i j A → (A → M j k B) → M i k B
```

The return and _»=_ operations are exactly indexed versions of the operators we used in Haskell. Thinking of monads in terms of their abstract behaviour, the indices can be thought of as an index at the beginning of the monadic behaviour, and the index at the end of the monadic behaviour. As such, the return operator doesn't effect the index, and the _»=_ operator can only bind together two monadic constructs whose indices match appropriately. We shall actually be using this form of indexed monad later in the definition of QIO, whereby the indices will relate to the set of qubits in a quantum computation. For now, we shall look at the specific instance of a raw indexed monad that relates to a raw monad as we have been using in Haskell. As mentioned previously, this form of raw monad is simply a raw indexed monad whose indices are the unit type (⊤).

```
RawMonad : (Set → Set) → Set₁
RawMonad M = RawIMonad {⊤} (\_ _ → M)
```

To finish off this section on monads, I shall give the implementation of a simple Maybe monad defined in Agda. The Maybe data-type must first be defined, exactly as in the Haskell implementation.

```
data Maybe (A : Set) : Set where
  Nothing : Maybe A
  Just : (a : A) → Maybe A
```

To give the definition of Maybe as a monad, we must define the bind operator, which we can then use in the definition of monadMaybe.

```
bindMaybe : ∀ {A B} → Maybe A → (A → Maybe B) → Maybe B
bindMaybe Nothing f = Nothing
bindMaybe (Just a) f = f a

monadMaybe : RawMonad Maybe
monadMaybe = record {return = \a → Just a
                    ; _»=_ = bindMaybe
                    }
```

8.4 Reversible Computation in Agda

We can now start to look at how formally verified proofs can be useful in designing reversible computations in Agda. The next chapter will look at how these techniques are used in defining a new version of the Quantum IO Monad in Agda. Reversible computation requires that for every function we define, we can also define the inverse of that function. This means that a reversible computation can be thought of as a pair of functions along with proofs that these functions are inverses of one another. We can use Agda's **record** type to define a reversible computation as such.

```
record Reversible (A B : Set) : Set where
  field
    f   : A → B
    f⁻¹ : B → A
    p   : ∀ {x : A} → f⁻¹ (f x) ≡ x
    p⁻¹ : ∀ {x : B} → f (f⁻¹ x) ≡ x
```

We can now go about defining some instances of reversible computations that fulfil this Reversible data-type. For example, the simplest instance would be of the not operation acting on a single Boolean value.

```
not' : Bool → Bool
not' true = false
not' false = true
```

We can define the not operation as a reversible computation by defining a member of the Reversible data-type in which not is its own inverse along with the relevant proof.

```
notnot : ∀ {x : Bool} → not' (not' x) ≡ x
notnot {true} = refl
notnot {false} = refl
¬ : Reversible Bool Bool
¬ = record {f = not'
           ;f-1 = not'
           ;p = notnot
           ;p-1 = notnot
           }
```

Another operation we could define is the Toffoli gate using the if construct.

```
if_then_else_ : ∀ {a : Set} → Bool → a → a → a
if true then a else _ = a
if false then _ else a = a
```

The Toffoli gate can be thought of as an if statement, acting on three Boolean inputs.

```
toffoli' : Bool × Bool × Bool → Bool × Bool × Bool
toffoli' (a, b, c) = if (a ∧ b) then (a, b, not' c)
                                else (a, b, c)
```

We know again that the Toffoli gate is self inverse, so we can define the Toffoli gate as a reversible computation by creating a member of the Reversible record type.

```
tt : ∀ {x : Bool × Bool × Bool} → toffoli' (toffoli' x) ≡ x
tt {true, true, true} = refl
tt {true, true, false} = refl
tt {true, false, z} = refl
tt {false, y, z} = refl

toffoli : Reversible (Bool × Bool × Bool) (Bool × Bool × Bool)
toffoli = record {f = toffoli'
                 ; f-1 = toffoli'
                 ; p = tt
                 ; p-1 = tt
                 }
```

We can now define the and function as an irreversible function embedded into this reversible Toffoli computation. The projection function for the 3rd element of a (non-dependent) 3-tuple is used.

```
proj3 : {A : Set} → A × A × A → A
proj3 (a, b, c) = c

_and_ : Bool → Bool → Bool
a and b = proj3 ((Reversible.f toffoli) (a, b, false))
```

It's also possible to go as far as to prove that this "reversible" and operation is equivalent to the standard ∧ function defined in the standard library.

```
proof : ∀ {a b : Bool} → (a ∧ b) ≡ (a and b)
proof {true} {true} = refl
proof {true} {false} = refl
proof {false} {b} = refl
```

To finish off this section on reversible computation we shall revisit the toy language we defined earlier in Haskell, and use the Reversible structure we have just given to re-implement the language in Agda as a formally verified reversible language.

We define the data-type Gate in a very similar manner as before, although now, to help with the verification process, we are able to index our gates by the number of bits they act upon. In fact, we can think of this as defining families of gates, whereby an arbitrary number of wires can be run in parallel below the gate.

```
data Gate : ℕ → Set where
  Empty : ∀ {n} → Gate n
  X : ∀ {n} → Gate (one + n) → Gate (one + n)
  Control : ∀ {n} → Gate n → Gate (one + n) → Gate (one + n)
  DWire : ∀ {n} → Gate n → Gate (one + n) → Gate (one + n)
```

The Empty constructor is a family of gates over any arbitrary number of bits. The X gate requires at least one bit to work upon, and the Control and DWire constructs require at least one more bit than the number of bits in their sub gates. We still have an extra Gate argument on the end of all but the Empty construct that defines that only gates over the same number of bits can be joined in sequence. This sequencing is again formalised by a monoidal structure.

In order to define a monoidal structure over gates, we must first define all the functions and proofs that are required in the definition of a monoid. We already know that our carrier set is to be the Gate data type, and that the ϵ element is the Empty constructor. We can also use the equality type as our equivalence relation (_≡_ is defined as an equivalence relation in the standard library), but we must go on to define the $\bullet$ operation along with all the relevant proofs before we can actually give the monoidal definition for our gates.

The $\bullet$ operation corresponds to the sequencing of gates, and is given as the following gate• function. This function is almost identical to the definition given in the Haskell implementation.

```
gate• : ∀ {n} → Gate n → Gate n → Gate n
gate• Empty g = g
gate• (X g) g' = X (gate• g g')
gate• (Control c g) g' = Control c (gate• g g')
gate• (DWire d g) g' = DWire d (gate• g g')
```

The next step is to prove that this gate• function, along with the equality type (_≡_) form a semi-group (we use the instance of IsEquivalence for _≡_ as given in the standard library). The first proof we must construct, is that the given gate• function is associative. This is achieved by using the congruence relation over each of the Gate constructors.

```
gateAssoc : ∀ {n} → (x y z : Gate n) →
              gate• (gate• x y) z ≡ gate• x (gate• y z)
gateAssoc Empty y z = refl
gateAssoc (X y) y' z = cong X (gateAssoc y y' z)
gateAssoc (Control y y') y0 z = cong (Control y) (gateAssoc y' y0 z)
gateAssoc (DWire y y') y0 z = cong (DWire y) (gateAssoc y' y0 z)
```

We must also prove that the gate• function preserves the equivalence relation, which can be achieved by using congruence over each of the arguments to gate•.

```
gate•≈ : ∀ {n} → {x y u v : Gate n} → (x ≡ y) → (u ≡ v)
           → gate• x u ≡ gate• y v
gate•≈ {n} {x} {y} {u} {v} p p' = trans (cong (gate• x) p')
                                        (cong (\y' → gate• y' v) p)
```

We can now build our proof of a semi-group using the definitions given above.

```
gateIsSemigroup : ∀ {n} → IsSemigroup _≡_ (gate• {n})
gateIsSemigroup {n} = record {isEquivalence = isEquivalence
                             ; assoc = gateAssoc
```

```
                                  ; •-pres-≈ = gate•≈
                                  }
```

To extend this proof to a proof that the given operations form a monoid over the Gate data-type, we must now show that the Empty constructor is both a left- and right-identity to the gate• function. The left-identity proof is trivial due to the definition of the gate• function, and we can again use the congruence relation over each of the Gate constructors to give that Empty is a right-identity to the gate• function.

```
gateID : ∀ {n} → (x : Gate n) → gate• x Empty ≡ x
gateID Empty = refl
gateID (X y) = cong X (gateID y)
gateID (Control y y') = cong (Control y) (gateID y')
gateID (DWire y y') = cong (DWire y) (gateID y')
```

Putting all these proofs together, we come up with the following proof of the monoidal structure.

```
gateIsMonoid : ∀ {n} → IsMonoid _≡_ (gate• {n}) Empty
gateIsMonoid {n} = record {isSemigroup = gateIsSemigroup
                          ; identity = (\x → refl), gateID
                          }
```

With the proofs constructed we are able to define the monoid.

```
monoidGate : ∀ {n} → Monoid
monoidGate {n} = record {carrier = Gate n
                        ; _≈_ = _≡_
                        ; _•_ = gate•
                        ; ϵ = Empty
                        ; isMonoid = gateIsMonoid
                        }
```

To give us easier access to the monoidal operations of the Gate structure, we are able to open monoidGate in the same way as we import and open modules from the standard library. This gives us direct access to all the fields in monoidGate (E.g. _•_ and ϵ).

```
open module mG {n} = Monoid (monoidGate {n})
```

We continue in a very similar way to give a semantics for our language in terms of the primitive operations, and the monoidal operations to sequence them.

```
x : ∀ {n} → Gate (one + n)
x = X ϵ

control : ∀ {n} → Gate (one + n) → Gate (two + n)
control g = Control g ϵ

dwire : ∀ {n} → Gate (one + n) → Gate (two + n)
dwire g = DWire g ϵ
```

We could also give the reverse operation here that works on the syntactic level, but as we are going to embed our evaluator into the Reversible structure defined above, any evaluated circuit will have a corresponding inverse semantics anyway. We can now give the same examples as we did in the Haskell version, the only difference being that we have to state the size of the gates in their types.

```
toffoli : ∀ {n} → Gate (three + n)
toffoli = control (control x)

control' : ∀ {n} → Gate (one + n) → Gate (two + n)
control' g = x • control g • x

controlXX : ∀ {n} → Gate (three + n)
controlXX = control x • control (dwire x)
```

Instead of defining a new Circuit data-type to represent our computations in terms of functions, this is where we use the Reversible structure from above. This means that we must now define the Reversible data-type as a monoidal structure, so that we can lift the monoidal structure of our Gate data-type during evaluation. The most complicated step of this process is in defining the functions that combine the proofs of the two Reversible structures which are to be joined using the monoidal _•_ operation.

The proofs are given by transitivity of the underlying p (and p⁻¹) proofs of the given Reversible structures, although we have to inform the system to evaluate these proofs over the corresponding intermediate states.

```
combineP : {a : Set} → {a' : a} →
  (r : Reversible a a) → (r' : Reversible a a) →
  Reversible.f⁻¹ r
    (Reversible.f⁻¹ r'
      (Reversible.f r'
         (Reversible.f r a'))) ≡ a'
combineP {_} {a'} r r' with Reversible.p r {a'}
  | cong (Reversible.f⁻¹ r)
         ((Reversible.p r') {(Reversible.f r) a'})
...| a | b = trans b a
combinePR : {a : Set} → {a' : a} →
            (r : Reversible a a) → (r' : Reversible a a) →
  Reversible.f r'
    (Reversible.f r
      (Reversible.f⁻¹ r
         (Reversible.f⁻¹ r' a'))) ≡ a'
combinePR {_} {a'} r r' with Reversible.p⁻¹ r' {a'}
  | cong (Reversible.f r')
    ((Reversible.p⁻¹ r) {(Reversible.f⁻¹ r') a'})
...| a | b = trans b a
```

Composition (_•_) of Reversible structures can now be given by the functional composition of the underlying f (and f⁻¹) functions, along with the combined proofs as defined above.

```
reversible• : ∀ {a} → Reversible a a → Reversible a a → Reversible a a
reversible• r r' = record
  {f = \a' → (Reversible.f r') ((Reversible.f r) a')
  ;f⁻¹ = \a' → (Reversible.f⁻¹ r) ((Reversible.f⁻¹ r') a')
  ;p = combineP r r'
  ;p⁻¹ = combinePR r r'
  }
```

The identity element (ϵ) corresponds to a reversible structure whose functions are the identity function (acting on the underlying type a), and the proofs are simply given by reflexivity.

```
reversibleϵ : ∀ {a} → Reversible a a
reversibleϵ = record {f = \x → x
                     ;f⁻¹ = \x → x
                     ;p = refl
                     ;p⁻¹ = refl
                     }
```

In order to define a monoidal structure we also need some form of equivalence relation between members of the Reversible data-type. This can be achieved by defining a **record** type that contains proofs that the underlying f and f⁻¹ functions of the two Reversible structures are equivalent using the normal _≡_ type. This is sufficient (as we shall see) and we don't need to provide proofs that the underlying proofs are equivalent (which is difficult as by definition they have different types).

```
record _≡R_ {a : Set} (r r' : Reversible a a) : Set where
  field
    f : Reversible.f r ≡ Reversible.f r'
    f⁻¹ : Reversible.f⁻¹ r ≡ Reversible.f⁻¹ r'
```

To show that this relation operation (_≡R_) is an equivalence relation we give proofs of reflexivity, transitivity, and symmetry. In essence, these proofs are lifting the underlying proofs of the _≡_ equivalence type.

```
≡RIsEquivalence : ∀ {a} → IsEquivalence (_≡R_ {a})
≡RIsEquivalence = record
  {refl = λ {x'} →
      record {f = refl
             ;f⁻¹ = refl
             }
  ; trans = λ {i j k} → \p p' →
      record {f = trans (_≡R_.f p) (_≡R_.f p')
             ;f⁻¹ = trans (_≡R_.f⁻¹ p) (_≡R_.f⁻¹ p')
             }
  ; sym = λ {i j} → \p →
      record {f = sym (_≡R_.f p)
             ;f⁻¹ = sym (_≡R_.f⁻¹ p)
             }
  }
```

We must go on to build up the proofs that are required in order to define the Reversible monoid, and as such must give proofs that the composition function we have defined is associative (with respect to the equivalence relation _≡R_), and also that the composition operator (_•_) preserves equivalence. Firstly, the associativity proof is given.

```
reversibleAssoc : ∀ {a} → (x y z : Reversible a a)
   → reversible• (reversible• x y) z ≡R reversible• x (reversible• y z)
reversibleAssoc x y z = record {f = refl
                               ;f⁻¹ = refl
                               }
```

The proof that _•_ preserves our equivalence relation is just a congruence proof over the underlying functions (f and f⁻¹) of the given Reversible structures.

```
reversible•≈ : ∀ {a} → {x y u v : Reversible a a} → (x ≡R y)
                           → (u ≡R v) → reversible• x u ≡R reversible• y v
reversible•≈ {a} {x} {y} {u} {v} xy uv
    = record {f = trans (cong (\x' a' →
                                          Reversible.f u (x' a')) (_≡R_.f xy))
                                      (cong (\x' a' →
                                          x' (Reversible.f y a')) (_≡R_.f uv))
                      ; f⁻¹ = trans (cong (\x' a' →
                                          Reversible.f⁻¹ x (x' a')) (_≡R_.f⁻¹ uv))
                                      (cong (\x' a' →
                                          x' (Reversible.f⁻¹ v a')) (_≡R_.f⁻¹ xy))
                      }
```

These proofs can now be used in defining that the Reversible structure, along with the given composition and equivalence, form a semi-group.

```
reversibleIsSemigroup : ∀ {a} → IsSemigroup (_≡R_ {a}) (reversible•)
reversibleIsSemigroup = record {isEquivalence = ≡RIsEquivalence
                                              ; assoc = reversibleAssoc
                                              ; •-pres-≈ = reversible•≈
                                              }
```

The last proofs required to show that we have a monoidal structure, are proofs that the identity element reversibleϵ is indeed a left and right identity to the composition function.

```
reversibleLID : ∀ {a} → (x : Reversible a a)
    → reversible• reversibleϵ x ≡R x
reversibleLID x = record {f = refl
```

```
                                ; f⁻¹  =  refl
                                }
reversibleRID : ∀ {a} → (x : Reversible a a)
   → reversible• x reversibleϵ ≡R x
reversibleRID x = record {f = refl
                                ; f⁻¹  =  refl
                                }
```

We can put all the previous proofs together to give our proof that the Reversible structure, along with the given composition, equivalence and identity element form a monoid.

```
reversibleIsMonoid : ∀ {a} → IsMonoid (_≡R_ {a}) reversible• reversibleϵ
reversibleIsMonoid = record {isSemigroup = reversibleIsSemigroup
                                  ; identity = reversibleLID,
                                               reversibleRID
                                  }
```

With all the necessary functions and proofs in place, we are able to define the monoid.

```
monoidReversible : ∀ {a} → Monoid
monoidReversible {a} = record {carrier = Reversible a a
                                    ; _≈_ = _≡R_
                                    ; _•_ = reversible•
                                    ; ϵ = reversibleϵ
                                    ; isMonoid = reversibleIsMonoid
                                    }
```

To prevent name clashes, instead of opening the **record** like we did for the monoidGate instance, we shall give functions that act as an interface to the required elements.

```
ϵϵ : ∀ {a} → Reversible a a
ϵϵ = Monoid.ϵ monoidReversible

_••_ : ∀ {a} → Reversible a a → Reversible a a → Reversible a a
_••_ = Monoid._•_ monoidReversible
```

We can now think of our evaluated circuits (over n bits) as members of the Reversible data-structure acting between vectors of Booleans of length n. As the Reversible data-structure is a **record** type, it is useful to first define the evaluation functions for each individual gate, and then look at the proofs that we must provide for each gate. Firstly, the evaluation of a single X gate will just negate the first element of the argument vector. The type of the X gate ensures that the argument vector to the evaluation function will always have at least one element, and as such we don't need to worry about an empty argument vector.

```
evalX : ∀ {n} → Vec Bool (one + n) → Vec Bool (one + n)
evalX (x :: xs) = (not' x) :: xs
```

The evaluation of a Control gate will use a recursive call to the main evaluation function to extract the corresponding behaviour of its sub-gate structure. As such, the evaluation function for a single Control gate will be given the corresponding function to run over the tail of the argument vector if the head element of this vector is true.

```
evalC : ∀ {n} → (Vec Bool n → Vec Bool n)
                → Vec Bool (one + n) → Vec Bool (one + n)
evalC g (x :: xs) = x :: (if x then g xs else xs)
```

The evaluation of a single DWire gate is very similar to the evaluation of a single Control gate, although in this instance, the function corresponding to the sub-gate structure is always evaluated.

```
evalD : ∀ {n} → (Vec Bool n → Vec Bool n)
                → Vec Bool (one + n) → Vec Bool (one + n)
evalD g (x :: xs) = x :: (g xs)
```

As X is its own inverse, we only have to provide one proof that the evaluation of an X gate is indeed reversible (although we do have to use it twice when constructing the main evaluation function).

```
evalXP : ∀ {n} → ∀ {xs : Vec Bool (one + n)} → evalX (evalX xs) ≡ xs
evalXP {n} {true :: xs} = refl
evalXP {n} {false :: xs} = refl
```

Because of the recursive nature of the evaluation of both the Control and DWire gates, we must tell Agda that the proofs of the reversibility of these evaluation functions require a form of mutual recursion with the main evaluation function. This is done in Agda by putting all the mutually recursive functions in a **mutual** block. Within such a **mutual** block, we shall give the definition of the main evaluation function (reversibleCircuit) first, and then give the remaining proofs.

The main evaluation function (reversibleCircuit) pattern matches over the Gate argument to give the first Gate structure (X, Control, or DWire). The monoidal structure defined for the Reversible data-type is used to recursively evaluate the rest of the Gate structure. Other than this, the evaluation function creates a member of the Reversible type by calling the evaluation functions for each structure, along with their relevant proofs.

```
mutual
  reversibleCircuit : ∀ {n} → Gate n →
                      Reversible (Vec Bool n) (Vec Bool n)
  reversibleCircuit Empty = ϵϵ
  reversibleCircuit (X g') = record
    {f = evalX
    ; f⁻¹ = evalX
    ; p = evalXP
    ; p⁻¹ = evalXP
    } •• reversibleCircuit g'
  reversibleCircuit (Control g g') = record
    {f = evalC (Reversible.f (reversibleCircuit g))
    ;f-1 = evalC (Reversible.f-1 (reversibleCircuit g))
    ;p = evalCP g
    ;p-1 = evalCPR g
    } •• reversibleCircuit g'
  reversibleCircuit (DWire g g') = record
    {f = evalD (Reversible.f (reversibleCircuit g))
    ;f-1 = evalD (Reversible.f-1 (reversibleCircuit g))
    ;p = evalDP g
    ;p-1 = evalDPR g
    } •• reversibleCircuit g'
```

The proof required for a single Control gate (evalCP) is a simple reflexivity proof in the case that the head of the argument vector is false, and in the case that the head of the argument vector is true, we can use a congruence of the proof of the underlying sub-gate. The inverse proof (evalCPR) just uses the inverse proof of the underlying sub-gate.

```
evalCP : ∀ {n} → {xs : Vec Bool (one + n)} → (g : Gate n) →
         evalC (Reversible.f-1
                  (reversibleCircuit g))
                (evalC (Reversible.f
                          (reversibleCircuit g)) xs) ≡ xs
evalCP {n} {true :: xs} g
  with Reversible.p (reversibleCircuit g) {xs}
...| p = cong (_::_ true) p
evalCP {n} {false :: xs} g = refl
evalCPR : ∀ {n} → {xs : Vec Bool (one + n)} → (g : Gate n) →
          evalC (Reversible.f
                    (reversibleCircuit g))
                  (evalC (Reversible.f⁻¹
                          (reversibleCircuit g)) xs) ≡ xs
evalCPR {n} {true :: xs} g
  with Reversible.p⁻¹ (reversibleCircuit g) {xs}
...| p = cong (_::_ true) p
evalCPR {n} {false :: xs} g = refl
```

The proofs for a single DWire gate don't depend on the head of the argument vector, so can just be given as a congruence of the underlying proofs from their sub-gate.

```
evalDP : ∀ {n} → {xs : Vec Bool (one + n)} → (g : Gate n) →
        evalD (Reversible.f⁻¹
                (reversibleCircuit g))
              (evalD (Reversible.f
                        (reversibleCircuit g)) xs) ≡ xs
evalDP {n} {x :: xs} g
  with Reversible.p (reversibleCircuit g) {xs}
...| p = cong (_::_ x) p
evalDPR : ∀ {n} → {xs : Vec Bool (one + n)} → (g : Gate n) →
         evalD (Reversible.f
                 (reversibleCircuit g))
               (evalD (Reversible.f⁻¹
                        (reversibleCircuit g)) xs) ≡ xs
evalDPR {n} {x :: xs} g
  with Reversible.p⁻¹ (reversibleCircuit g) {xs}
...| p = cong (_::_ x) p
```

The last thing we give here is a run function that uses the evaluator to generate the reversible computation from a circuit, and evaluates the running of the computation (or more specifically the computation in its forward direction) over the given vector of Booleans. This run function can again be used to embed the & function into the t∘ff∘li circuit.

```
run : ∀ {n} → Gate n → Vec Bool n → Vec Bool n
run g xs = Reversible.f (reversibleCircuit g) xs

_&_ : Bool → Bool → Bool
a & b = lookup (suc (suc zero)) (run t∘ff∘li (a :: b :: false :: []))
```

Chapter 9

The Quantum IO Monad in Agda

We have previously introduced the idea of the Quantum IO Monad, which introduces a monadic structure to explicitly deal with the effects inherent in quantum computations. The choice of Haskell as the parent language followed mainly from its built-in support for monads, but also more generally because pure functional programming languages are in a unique position because of the way that any sort of effects have to be dealt with explicitly. Indeed, it is the effects inherent in quantum computation that lead to some of the most interesting unclassical properties, and making these effects explicit would be impossible in a paradigm in which effects form an implicit part of computation. The Quantum IO Monad in Haskell is only a first approximation to a language with a built-in monadic structure capable of explicitly dealing with these effects, and gives a way of looking more at how such a language might be defined. The final chapter on the Haskell implementation (7) discusses some of the restrictions that arise. Keeping those in mind, it becomes necessary for such a language to be able to ensure the unitarity of operations that can be defined. We have restrictions on the values of types that can be used in certain situations, such as the restriction that the qubits used in a control structure cannot also be used in the branches of the controlled operations. This type of dependency between types and values is exactly the type of problem that can be overcome using a dependent type system. In moving to Agda as a parent language, we are able to keep the pure aspects of functional programming that were the initial inspiration behind a monadic structure for QIO, but also introduce a dependent type system, that gives us the stronger control over types and values that we require to fully model the unitary operations we'd like to implement. From the work introduced in this thesis, it becomes apparent that to implement a language such as QIO, it is really necessary that we work in a pure setting, so we can model effects explicitly, but also that we work within a dependent type system, so we can have strict control over restricting the values of types that can be used with-in the sub-unitary structures, such as control branches and ancilliary qubits. In this instance, Agda is the natural choice of parent language for QIO because its close relation to the syntax of Haskell leaves us with a new implementation of QIO which has a very similar syntax to the previous Haskell implementation.

9.1 Introduction

This chapter introduces work which goes towards a reimplementation of the Quantum IO Monad in the dependently typed language Agda. The approach started with an implementation of the classical subset of QIO in Agda, which is introduced in the next section. The following Chapter (10) then goes on to describe how this approach could be extended to the full set of quantum operations available in the original Haskell implementation of QIO.

One of the problems that is encountered is that there is currently no library that defines the real numbers in Agda. Much work is on-going into the study of real numbers in dependently typed languages (such as [GN02]), but as it is only certain properties of the reals, and complex numbers that we require, we are able to use floating point numbers in this implementation as an approximation to the reals, and can postulate certain properties of the real numbers that might not actually hold for the floating point interpretation.

The properties we need are simply the properties that arise from the fact that the real numbers form a field. We introduce these properties as postulated types, and then extend them to proofs that the complex numbers also form a field. These field axioms can then be used in the proofs inherent in our unitary operators.

This chapter looks more at the properties of the unitary operators in Agda, and how we use the stronger type-system to ensure their unitarity. The full implementation of QIO is also covered, but only a few examples of QIO computations are given. The source can again be found on-line [Gre09].

9.2 Classical QIO in Agda

To start looking at a reimplementation of QIO in Agda, it is first useful to look at a reimplementation of the classical subset of QIO. We first define a type that represents the semantics of a unitary operator (USem), and then define constructor functions for members of this type that correspond to the syntax of unitary operators. It is in designing the types of these syntax functions that we can start to look at using the stronger type system to ensure we no longer need the semantic side conditions from the Haskell implementation. Once we have defined the unitary operators, we can then give a definition for the other parts of the QIO monad, and implement the classical run function for *simulating* these new QIO computations.

Although we are only implementing the classical subset of QIO here, we shall try and keep with the original QIO terminology where possible. As before, computations are going to act on qubits, so we must define what a qubit in QIO is. In the Haskell implementation, we simply had individual qubits as an integer reference to the qubit they represented, but it is more useful to think of a QIO computation as indexed by the number of qubits it uses. Hence we use a type for qubits that corresponds to a set indexed by n, the number of qubits available. This is pretty much a type-synonym of the Fin data-type that we saw in the introduction to dependent types in Agda.

```
data Qbit : ℕ → Set where
  qzero : {n : ℕ} → Qbit (suc n)
  qsuc : {n : ℕ} (i : Qbit n) → Qbit (suc n)
```

The individual qubits can still be thought of as references to the available qubits, but having the set of qubits indexed by the number of qubits gives us more information to work with.

9.2.1 A formally verified semantics for unitary operations

As we are only working with the classical subset of QIO, we are able to define the semantics of a unitary as a function acting on vectors of Booleans of length n, where n is still the number of qubits.

```
record USem (n : ℕ) : Set where
  field
    f : Vec Bool n → Vec Bool n
    f⁻¹ : Vec Bool n → Vec Bool n
    p : ∀ {x : Vec Bool n} → f⁻¹ (f x) ≡ x
    p⁻¹ : ∀ {x : Vec Bool n} → f (f⁻¹ x) ≡ x
```

Compare this code to the Reversible data-type we introduced earlier and you will see that it is very similar. Every operation we wish to provide must also provide a semantics that is a member of the USem data-type, thus providing a semantics for unitaries that is formally verified to be unitary (or in this case reversible).

Before we look at the operations we wish to define, we can already declare these semantics to be a monoid. The construction of the monoid is very similar to the construction of the monoidReversible given in the previous chapter. The first thing we can define is an element of USem that corresponds to the identity element of the monoid.

```
mempty : ∀ {n} → USem n
mempty = record {f = \x → x
  ; f^-1 = \x → x
  ; p = refl
  ; p^-1 = refl
  }
```

In defining the operation that corresponds to the monoidal composition function, we must again think about how we can combine the proofs of the two underlying USem structures. This can be achieved by transitivity of the underlying proofs, although we must again instantiate these proofs with respect to the intermediary results of the computation.

```
combineP : ∀ {n} → {xs : Vec Bool n} → (u u' : USem n)
  → USem.f^-1 u (USem.f^-1 u' (USem.f u' (USem.f u xs))) ≡ xs
combineP {n} {xs} u u' with USem.p u {xs}
  | cong (USem.f^-1 u) ((USem.p u') {(USem.f u) xs})
...| a | b = trans b a
combinePR : ∀ {n} → {xs : Vec Bool n} → (u u' : USem n)
  → USem.f u' (USem.f u (USem.f^-1 u (USem.f^-1 u' xs))) ≡ xs
combinePR {n} {xs} u u' with USem.p?^1 u' {xs}
  | cong (USem.f u') ((USem.p^-1 u) {(USem.f^-1 u') xs})
...| a | b = trans b a
```

These proofs can now be used in defining the mappend operation, with the combined functions corresponding to the functional composition of the corresponding functions from the underlying USem structures.

```
mappend : ∀ {n} → (USem n) → (USem n) → (USem n)
mappend u u' = record {f = \xs → (USem.f u') ((USem.f u) xs)
  ; f^-1 = \xs → (USem.f^-1 u) ((USem.f^-1 u') xs)
```

```
      ; p  =  combineP u u'
      ; p⁻¹  =  combinePR u u'
      }
```

Before we can define a monoidal structure using the mempty and mappend operations from above, we must define an equivalence relation for the USem structures. We have a similar problem as in the Reversible example, and as such define an equivalence relation in the form of a **record** containing fields that are proofs that the underlying f and f⁻¹ functions are equivalent.

```
record _U_ {n : ℕ} (r r' : USem n) : Set where
  field
    p  : USem.f r ≡ USem.f r'
    p⁻¹  : USem.f⁻¹ r ≡ USem.f⁻¹ r'
```

We are now able to start building up all the proof terms that are required in defining our monoidal structure. The first proof term we need to show is that the relation we have defined above is indeed an equivalence relation, and this can be achieved by lifting the underlying refl, trans, and sym proofs for the main equivalence type (_≡_).

```
isEquivalenceU : ∀ {n} → IsEquivalence (_U_ {n})
isEquivalenceU =
  record {refl  =  record {p  =  refl
                          ; p⁻¹  =  refl
                          }
         ; trans  =  \p p' →
           record {p  =  trans (_U_.p p) (_U_.p p')
                  ; p⁻¹  =  trans (_U_.p⁻¹ p) (_U_.p⁻¹ p')
                  }
         ; sym  =  \p  →  record {p  =  sym (_U_.p p)
                                 ; p⁻¹  =  sym (_U_.p⁻¹ p)
```

```
                                        }
                }
```

We must also show that the mappend operation is associative with respect to the equivalence relation we have defined (_U_)

```
assocMappend : ∀ {n} → (x y z : USem n)
   → mappend (mappend x y) z U mappend x (mappend y z)
assocMappend x y z = record {p = refl
                            ; p⁻¹ = refl
                            }
```

and that our mappend operation preserves the equivalence relation.

```
mappend-pres-U : ∀ {n} → {x y u v : USem n} → x U y → u U v
                     → mappend x u U mappend y v
mappend-pres-U {n} {x} {y} {u} {v} xy uv =
  record {p = trans (cong (\u' x' → (USem.f u (u' x')))
                          (_U_.p xy))
                    (cong (\u' x' → u' (USem.f y x'))
                          (_U_.p uv))
         ; p⁻¹ = trans (cong (\u' x' → (USem.f⁻¹ x (u' x')))
                          (_U_.p⁻¹ uv))
                    (cong (\u' x' → u' (USem.f⁻¹ v x'))
                          (_U_.p⁻¹ xy))
         }
```

Putting the proofs so far defined together, we can construct a proof that the USem datatype, along with the mappend operation, and the _U_ equivalence relation, form a semi-group.

```
isSemiGroupUSem : ∀ {n} → IsSemigroup (_U_ {n}) mappend
isSemiGroupUSem = record {isEquivalence = isEquivalenceU
```

```
                                        ; assoc = assocMappend
                                        ; •-pres-≈ = mappend-pres-U
                                        }
```

In order to prove that we have a full monoidal structure, it is now necessary to show that the mempty element we have defined is indeed both a left and right identity to the mappend operation.

```
usemLID : ∀ {n} → (x : USem n) → mappend mempty x U x
usemLID x = record {p = refl
                   ; p⁻¹ = refl
                   }
usemRID : ∀ {n} → (x : USem n) → mappend x mempty U x
usemRID x = record {p = refl
                   ; p⁻¹ = refl
                   }
```

Putting all the proofs together, we end up with a proof that the USem datatype, along with mappend, mempty, and the equivalence relation _U_, form a monoid

```
isMonoidUSem : ∀ {n} → IsMonoid (_U_ {n}) mappend mempty
isMonoidUSem = record {isSemigroup = isSemiGroupUSem
                      ; identity = usemLID, usemRID
                      }
```

and we are able to define it as such.

```
monoidUSem : ∀ {n} → Monoid
monoidUSem {n} = record {carrier = USem n
                        ; _≈_ = _U_
                        ; _•_ = mappend
                        ; ϵ = mempty
```

```
                              ; isMonoid  =  isMonoidUSem
                              }
```

As the inverse of each unitary is inherent in the structure of the USem datatype, we can create the revere (or inverse) of a unitary, trivially, by swapping the two functions and the two proofs within the USem structure.

```
reverse : ∀ {n} → USem n → USem n
reverse u = record {f = USem.f⁻¹ u
                   ; f⁻¹ = USem.f u
                   ; p = USem.p⁻¹ u
                   ; p⁻¹ = USem.p u
                   }
```

9.2.2 Unitaries in QIO Agda

Now that we have a suitable representation for the semantics of our unitaries, we can start defining the unitary operations that we would like to model in QIO. These operations will correspond to the operations that formed the classical subset of the operations we had in the Haskell implementation of QIO. Namely, we shall define operations corresponding to the swap, cond, rot, and ulet operations, although in this case we shall restrict the rotations available to just the negation operation, which we shall refer to as x (after its relation to the Pauli X rotation in the quantum realm).

As our semantics are defined in terms of functions acting on vectors of Booleans, and the corresponding proofs of unitarity, we are able to construct members of the USem type by first defining the necessary functions and proofs, and putting them together in a USem structure. Firstly, we can implement the swap unitary in terms of two standard vector functions. That is, the function insert : ∀ {A n} → A → Vec A n → (Qbit n) → Vec A n which inserts the given element into the given vector at the given (qubit) index, and the `_!!_ : ∀ {A n} → Vec A n → (Qbit n) → A` lookup function, that returns the element in the given vector at the given (qubit) index. The definition of Qbit ensures that the given index is always valid with respect to the vector.

```
swapQ : ∀ {n : ℕ} → (Qbit n) → (Qbit n)
   → Vec Bool n → Vec Bool n
swapQ qzero qzero (x :: xs) = x :: xs
swapQ qzero (qsuc i) (x :: xs) =
  insert x (((x :: xs) !! (qsuc i)) :: xs) (qsuc i)
swapQ (qsuc i) qzero (x :: xs) =
  insert x (((x :: xs) !! (qsuc i)) :: xs) (qsuc i)
swapQ (qsuc i) (qsuc i') (x :: xs) = x :: (swapQ i i' xs)
```

We know that the swap operation is self inverse, so the only proof we need to construct is that this is the case. To do this, we are able to make use of two sub-proofs. The first (pinsert) is a proof that the element in a vector at the position we have inserted it, is indeed the element that we inserted.

```
pinsert : ∀ {n} → (x : Bool) → (xs : Vec Bool n) → (i : Qbit n)
   → (insert x xs i !! i) ≡ x
pinsert x [] ()
pinsert true (x :: xs) qzero = refl
pinsert false (x :: xs) qzero = refl
pinsert x (x' :: xs) (qsuc i) = pinsert x xs i
```

The second is a proof that reinserting an element into the same index of a vector from where it came, leaves us back with the original vector.

```
pinsert' : ∀ {n} → (x : Bool) → (xs : Vec Bool n) → (i : Qbit n)
   → (insert (xs !! i) (insert x xs i) i) ≡ xs
pinsert' x [] ()
```

```
pinsert' x (x' :: xs) qzero  =  refl
pinsert' x (x' :: xs) (qsuc i)  =  cong (_::_ x') (pinsert' x xs i)
```

Now, the overall proof that swapQ is self inverse can be constructed using the _:≡:_ constructor that takes a proof that the head of two vectors are equivalent, and that the tail of two vectors are equivalent, and returns a proof that the overall vectors are equivalent.

```
pswapQ : ∀ {n : ℕ} → (y y' : Qbit n) → (x : Vec Bool n)
   → swapQ y y' (swapQ y y' x) ≡ x
pswapQ () () []
pswapQ qzero qzero (x :: xs)  =  refl
pswapQ qzero (qsuc i) (x :: xs)  =  (pinsert x xs i) :≡: (pinsert' x xs i)
pswapQ (qsuc i) qzero (x :: xs)  =  (pinsert x xs i) :≡: (pinsert' x xs i)
pswapQ (qsuc i) (qsuc i') (x :: xs)  =  cong (_::_ x) (pswapQ i i' xs)
```

Putting all the pieces together, we can define the Swap constructor. Note that the type of the given Swap constructor is almost identical to the type of its Haskell counter-part, and that the monoidal structure for the USem datatype is used to append the following u onto the end of the given USem structure.

```
Swap : ∀ {n : ℕ} → (Qbit n) → (Qbit n) → USem n → USem n
Swap x y u  =  record {f  =  \xs → swapQ x y xs
  ; f⁻¹  =  \xs → swapQ x y xs
  ; p  =  λ {xs} → pswapQ x y xs
  ; p⁻¹  =  λ {xs} → pswapQ x y xs
  } • u
```

The definition for the X member of USem is also unspectacular, and as such is omitted from this thesis. The full code can be found online [Gre09]. It is only when we start to look at the behaviour of the cond and ulet constructors that we can start to use the extra power of dependent types in our definitions so as to overcome the side-conditions that were necessary in the Haskell implementation. If we look at the two side-conditions for these constructors, we can actually notice a big similarity. In the case of a conditional, we need that the control qubit is left in a state separable from the qubits in the conditional unitaries that it is controlling, and in the case of the ulet structure, we require that the auxiliary qubit is left in a state separable from the other qubits in the unitary in which it is used. In the classical case, this corresponds to the control/ulet qubit being left in the state in which it started, after the application of the sub-unitary. In Agda, this side condition can be stated in terms of a proof that this is indeed the case, and as such, a QIO program must provide an element of this proof type, or it won't compile. We can define the Sep datatype in terms of a **record** containing two proofs. The first (s) is exactly the proof described above, and the second (s^{-1}) is the same proof but for the inverse of the given USem.

```
record Sep {n : ℕ} (i : Qbit n) (u : USem n) : Set where
  field
    s : ∀ {xs} → ((USem.f u xs) !! i ≡ xs !! i)
    s⁻¹ : ∀ {xs} → ((USem.f⁻¹ u xs) !! i ≡ xs !! i)
```

We shall see, that in defining the cond and ulet structures, we need exactly the proofs provided by the Sep datatype to give overall proofs of their unitarity. Because of this extra requirement, the structures we're going to define to represent the cond and ulet operations will have different types than their Haskell counterparts. Firstly, the type of our new conditional operation will be Cond : ∀ {n : ℕ} → (i : Qbit n) → (Bool → (Σ (USem n) (\u → Sep {n} i u))) → USem n → USem n. That is, that each branch of the conditional must contain the unitary for that branch along with a member of the Sep datatype that provides a proof that the control qubit is separable from the application of that unitary. The behaviour of a conditional unitary is given exactly by the Boolean function, so the semantics of Cond will simply use the semantics of the unitary given when the Boolean function is applied to the value of the control qubit.

```
condQ : ∀ {n} → (i : Qbit n) →
    (Bool → (Σ (USem n) (\u → Sep {n} i u)))
        → Vec Bool n → Vec Bool n
condQ i fb xs = USem.f (proj₁ (fb (xs !! i))) xs
```

The inverse of the condQ function is given by condQR, and just extracts the underlying inverse function (f⁻¹) instead. As many of the proofs for condQR are very similar to the proofs for condQ they shall be omitted too. The proof that condQR is the inverse of condQ uses a helper function (condQPhelper) that is basically a congruence proof.

```
condQPhelper : ∀ {n} → {s t : USem n} → {xs : Vec Bool n}
    → (s ≡ t) → USem.f⁻¹ t (USem.f s xs) ≡ USem.f⁻¹ s (USem.f s xs)
condQPhelper refl = refl
```

The overall proof uses the underlying proof (USem.p) of its sub unitary using the proof given in the Sep data-type to lift this to a proof of the whole cond operation.

```
condQP : ∀ {n} → {xs : Vec Bool n} → (i : Qbit n)
    → (fb : (Bool → (Σ (USem n) (\u → Sep {n} i u))))
        → condQR i fb (condQ i fb xs) ≡ xs
condQP {n} {xs} i fb = trans
    (condQPhelper (cong (\b → proj₁ (fb b))
                             (sym (Sep.s (proj₂ (fb (xs !! i)))))))
    (USem.p (proj₁ (fb (xs !! i))) {xs})
```

We can now construct the overall definition for our Cond structure, again using the monoidal structure of USem to append the following u onto the end of the main conditional definition.

```
Cond : ∀ {n : ℕ} → (i : Qbit n)
    → (Bool → (Σ (USem n) (\u → Sep {n} i u)))
    → USem n → USem n
Cond {n} q fb u = record {f = \xs → condQ q fb xs
  ; f⁻¹ = \xs → condQR q fb xs
  ; p = λ {xs} → condQP {n} {xs} q fb
  ; p⁻¹ = λ {xs} → condQRP {n} {xs} q fb
  } • u
```

The definition of the Ulet member of USem is remarkably similar to the conditional, with the type Ulet : ∀ {n} → (b : Bool) → ((i : (Qbit (suc n))) → Σ (USem (suc n)) (\u → Sep {suc n} i u)) → USem n → USem n. With all the structures now defined, we are able to give a syntax for our unitaries that corresponds very well to the original syntax given in the Haskell implementation.

```
ux : ∀ {n : ℕ} → (Qbit n) → USem n
ux qn = X qn ϵ
uswap : ∀ {n : ℕ} → (Qbit n) → (Qbit n) → USem n
uswap q1 q2 = Swap q1 q2 ϵ
ucond : ∀ {n : ℕ} → (i : Qbit n)
   → (Bool → (Σ (USem n) (\u → Sep i u))) → USem n
ucond q fb = Cond q fb ϵ
ulet : ∀ {n : ℕ} → (b : Bool) → ((i : (Qbit (suc n)))
   → (Σ (USem (suc n)) (\u → Sep {suc n} i u))) → USem n
ulet b fq = Ulet b fq ϵ
```

With our syntax defined, we are now able to write unitary operators. As a simple example we can create the quantum if, that only performs the given unitary if the control qubit is true. In the case that the control qubit is false then the monoidal identity of USem is used. As such, we can require that the user provides a separability proof for the given unitary, from the control qubit, and provide a simple proof that any control qubit is separable from the monoidal identity (ϵ).

```
qif : ∀ {n : ℕ} → (i : Qbit n) → (u : USem n) → Sep {n} i u
    → USem n
qif {n} i u s = ucond i (\b → if b then u, s
  else ε,
    record {s = refl
           ; s⁻¹ = refl
           })
```

An example use of the qif operation is to create the cnot gate. We can't provide a concrete proof that the cnot gate is unitary for any control/target qubits, as it isn't unitary when these two qubits coincide, as such we are able to give a proof that the cnot gate is unitary if we have a proof that the control qubit isn't the same qubit as the target qubit. The _≢_ proof is actually a a synonym for the type _≡_ → ⊥, where ⊥ is an uninhabited type. The definition also makes use of the function lower≢ : ∀ {n} → {x y : Qbit n} → (qsuc x ≢ qsuc y) → (x ≢ y).

```
cnotP : ∀ {n} → (xs : Vec Bool n) → (cq q : Qbit n) → (cq ≢ q)
    → negate q xs !! cq ≡ xs !! cq
cnotP [] () () p
cnotP (x :: xs) qzero qzero p with p refl
...| ()
cnotP (x :: xs) qzero (qsuc i) p = refl
cnotP (x :: xs) (qsuc i) qzero p = refl
cnotP (x :: xs) (qsuc i) (qsuc i') p = cnotP xs i i' (lower≢ p)
```

Using this proof, we are able to define the cnot gate, whereby the user must give a proof that the control qubit and the target qubit are distinct from one another.

```
cnot : ∀ {n : ℕ} → (cq q : Qbit n) → (cq ≢ q) → USem n
cnot {n} cq q p = qif cq (ux q) (
  record {s = λ {xs} → cnotP xs cq q p
```

```
            ; s⁻¹ = λ {xs} → cnotP xs cq q p
            })
```

Another example unitary operation we can give is that of the Toffoli gate. In this instance, we are able to create separability proofs as long as the target qubit is distinct from both of the control qubits.

```
toffoliP : ∀ {n : ℕ} → (xs : Vec Bool n) → (q1 q2 q3 : Qbit n)
   → (q1 ≢ q3) → (cnp : q2 ≢ q3)
     → USem.f (cnot q2 q3 cnp) xs !! q1 ≡ xs !! q1
toffoliP xs q1 q2 q3 p cnp with xs !! q2
...| true = cnotP xs q1 q3 p
...| false = refl

toffoliPR : ∀ {n : ℕ} → (xs : Vec Bool n) → (q1 q2 q3 : Qbit n)
   → (q1 ≢ q3) → (cnp : q2 ≢ q3)
     → USem.f⁻¹ (cnot q2 q3 cnp) xs !! q1 ≡ xs !! q1
toffoliPR xs q1 q2 q3 p cnp with xs !! q2
...| true = cnotP xs q1 q3 p
...| false = refl
```

The Toffoli gate can now be given as a unitary, whereby the user must provide proofs that the target qubit is distinct from both of the control qubits.

```
toffoli : ∀ {n : ℕ} → (q1 q2 q3 : Qbit n)
   → (q1 ≢ q3) → (q2 ≢ q3) → USem n
toffoli {n} q1 q2 q3 p cnp =
  qif q1 (cnot q2 q3 cnp) (
    record {s = λ {xs} → toffoliP xs q1 q2 q3 p cnp
           ; s⁻¹ = λ {xs} → toffoliPR xs q1 q2 q3 p cnp
           })
```

Now that we have defined the new unitary operations for QIO in Agda, we would like to actually be able to use them in computations. The following section looks at how we can define the monadic operations of QIO as an indexed monad in Agda, and use it in defining formally verified QIO computations.

9.2.3 QIO as an indexed monad in Agda

Defining QIO in Agda is now a case of defining the monadic structures that enable us to initialise qubits, apply unitaries to these qubits, and finally measure qubits. It is useful in Agda think of this in terms of an indexed monad. Our indices will relate to the number of qubits a monadic computation requires when it starts, and the number of qubits that are in the system when it has finished. In fact, as our measurements don't remove qubits from the system (they only collapse their state into a base state), we know that the only operation we have that changes the number of qubits in the system is when we initialise a new qubit. Keeping this in mind, we are able to define the constructors of QIO.

```
data QIO (A : Set) : ℕ → ℕ → Set where
  QReturn : ∀ {n : ℕ} → A
              → QIO A n n
  MkQbit : ∀ {n m : ℕ} → Bool
             → (Qbit (suc n) → (QIO A (suc n) m))
             → QIO A n m
  ApplyU : ∀ {n m : ℕ} → USem n → QIO A n m
             → QIO A n m
  MeasQbit : ∀ {n m : ℕ} → (Qbit n) → (Bool → QIO A n m)
               → QIO A n m
```

Having the computations indexed by the number of qubits they require in scope, and the number of qubits that are still in scope afterwards enables the monadic behaviour to only bind suitably indexed QIO computations. This indexing also allows us to inform our simulation functions that they require a computation starting with zero qubits in scope, as any computation that requires a certain initialisation of qubits will have to define this initialisation when we want to run it. The bind operation of the monad (QIObind) can be defined as expected.

```
QIObind : ∀ {n m l : ℕ} → ∀ {A B : Set} → QIO A n m
            → (A → QIO B m l) → QIO B n l
QIObind (QReturn a) f = f a
QIObind (MkQbit b g) f = MkQbit b (\x → QIObind (g x) f)
QIObind (ApplyU u q) f = ApplyU u (QIObind q f)
QIObind (MeasQbit x g) f = MeasQbit x (\b → QIObind (g b) f)
```

and can be used in the definition of QIO as a raw indexed monad.

```
monadQIO : RawIMonad {ℕ} (\m n A → QIO A m n)
monadQIO = record {return = QReturn
  ; _»=_ = QIObind
  }
```

Now we have our monadic constructors, we are able to define the syntax we wish to use for creating QIO computations in Agda.

```
mkQbit : ∀ {n : ℕ} → Bool → QIO (Qbit (suc n)) n (suc n)
mkQbit b = MkQbit b QReturn

applyU : ∀ {n : ℕ} → USem n → QIO ⊤ n n
applyU u = ApplyU u (QReturn tt)

measQbit : ∀ {n : ℕ} → Qbit n → QIO Bool n n
measQbit x = MeasQbit x QReturn
```

As it stands however, the indexing of the monad gives us some complications when it comes to computations acting over multiple qubits. For example, if we wanted to define a computation that uses the toffoli gate unitary to create the logical and function, we would like to write the code as follows.

```
_and_ : Bool → Bool → QIO Bool zero 3
x and y = mkQbit x ≫= \qx →
          mkQbit y ≫= \qy →
          mkQbit false ≫= \qf →
          applyU (toffoli qx qy qf
                         qx≢qf qy≢qf) ≫
          measQbit qf
```

However, this code will not type check as the type of each of the three qubits is different. That is, qx : Qbit 1, qy : Qbit 2, and qz : Qbit 3. What we do know, is that elements of Qbit n can be lifted to elements of Qbit (suc n), so we are able to define the lift function to do this.

```
lift : ∀ {n : ℕ} → Qbit n → Qbit (suc n)
lift qzero = qzero
lift (qsuc q) = qsuc (lift q)
```

This lift function must now be used explicitly in our code, giving rise to the definition of _and_ as follows.

```
_and_ : Bool → Bool → QIO Bool zero 3
x and y = mkQbit x ≫= \qx →
          mkQbit y ≫= \qy →
          mkQbit false ≫= \qf →
          applyU (toffoli (lift (lift qx)) (lift qy) qf
                         qx≢qf qy≢qf) ≫
          measQbit qf
```

Although this gives rise to uglier code for our QIO programs, there are techniques that could be used to overcome this. A similar problem has been described in [Swi08] chapter 6. In the sections on weakening, the author looks into ways that the system can automatically weaken the necessary references (equating to the lifting of our qubits), although the full technique described for doing so cannot be applied in this instance as the qubits used in a unitary aren't explicitly referenced by the monadic constructors of QIO.

Now that we are able to define QIO computations, we would also like to be able to *run* them. The following section looks at the evaluation of classical QIO computations in Agda.

9.2.4 Evaluating classical QIO computations in Agda

The evaluation of QIO computations as defined above relies on the underlying USem structure of our unitary operators. Indeed, when evaluating the application of a unitary (ApplyU), we shall just be using the functions embedded in the USem structure applied to the current state of the system, in this case a suitably initialised vector of Boolean values. The MkQbit constructor will just append a new boolean value to the end of the current vector, and the MeasQbit constructor will return the value in the vector that is indexed by the qubit being measured.

```
runQIO : ∀ {n m : ℕ} → ∀ {A : Set} → QIO A n m
   → Vec Bool n → A
runQIO (QReturn a) v = a
runQIO {n} {m} (MkQbit b fq) v = runQIO (fq (newQbit n)) (v ::ʳ b)
runQIO (ApplyU u q) v = runQIO q ((USem.f u) v)
runQIO (MeasQbit q fb) v = runQIO (fb (v !! q)) v
```

The overall runC function for the running of classical QIO computations in Agda will just call the runQIO function with an empty vector. The classical nature of the computations again means that the return type doesn't need to be monadic, and indeed a QIO computation of type A is able to return a value of type A upon evaluation.

```
runC : ∀ {m : ℕ} → ∀ {A : Set} → QIO A zero m → A
runC q = runQIO q []
```

We finish this section by giving a simple proof that running the _and_ function as defined above is equivalent to the _∧_ function acting on Boolean values.

```
andproof : ∀ {a b : Bool} → a ∧ b ≡ runC (a and b)
andproof {true} {true} = refl
andproof {true} {false} = refl
andproof {false} {b} = refl
```

We now move on to look at how this Agda implementation of the classical subset of QIO can be extended into a fully quantum version.

Chapter 10

Quantum QIO in Agda

In order to extend our classical version of *QIO* in Agda, we need to look at how we are able to model a quantum state classically. This is because the semantics of our *QIO* computations needs to be given in terms of functions acting on these quantum states. The first problem we come across when looking at how we modelled quantum state in our Haskell implementation is that we used an implementation of the complex numbers in order to assign each base state within an overall quantum state a corresponding amplitude. In fact, it is easy to model the complex numbers in terms of their real and imaginary parts, so all we really need is an implementation of the real numbers. In Agda, there is currently no implementation of the real numbers, and re-implementing something such as the constructive reals defined in Coq ([GN02]), into Agda, is beyond the scope of this research. Fortunately, we are able to *simulate* real numbers in Agda by telling it to compile a postulated type of reals into the underlying floating point representation used in Haskell. This, however, leads to a lack of ability in reasoning about computations that use real numbers as we know nothing about their values until after compilation. Instead, we must postulate facts that we know hold for the real numbers, and use these in creating our proofs. The following section looks at how we are able to simulate complex numbers in Agda by postulating the real numbers and the properties given to us by knowing that the real numbers form a field.

10.1 Simulating the Complex number field in Agda

As previously mentioned, we don't have an implementation of the real numbers available to us in Agda. We can however define the type ℝ as an uninhabited type that we inform Agda to compile down to the underlying Haskell implementation of floating point numbers.

```
data ℝ : Set where

{-# COMPILED_DATA ℝ Float #-}
```

As Agda will still treat ℝ as an uninhabited type it would seem that we would be unable to define anything useful over it. However, Agda allows us to add postulates to our programs. We can also add compiled types for postulated functions, allowing some of our postulated types and functions to be approximated at run-time. In fact, the only information we need about the real numbers is that they form a field, and so it is the field axioms that we must postulate for ℝ. Firstly, we must postulate the elements and operations that a field contains.

```
postulate
  zeroℝ : ℝ
  oneℝ : ℝ
  _+ℝ_ : ℝ → ℝ → ℝ
  -ℝ : ℝ → ℝ
  _×ℝ_ : ℝ → ℝ → ℝ
  ÷ℝ : ℝ → ℝ
```

zeroℝ will be the additive identity, and will compile to the floating point number 0.0. oneℝ will be the multiplicative identity, and will compile to the floating point number 1.0. Addition (+ℝ) and multiplication (×ℝ) are assumed to have closure with-in ℝ, which is inherent in their types, and shall compile to + and * respectively. The additive inverse is given by -ℝ and compiles to the operation

-ℝ x $= 0.0 - x$. Similarly, the multiplicative inverse is defined by ÷ℝ and compiles to the operation ÷ℝ x $= 1.0/x$. We can now go on to postulate all the field properties of ℝ.

```
postulate
  assoc+ℝ : ∀ {a b c : ℝ} → a +ℝ (b +ℝ c) ≡ (a +ℝ b) +ℝ c
  assoc×ℝ : ∀ {a b c : ℝ} → a ×ℝ (b ×ℝ c) ≡ (a ×ℝ b) ×ℝ c
  com+ℝ : ∀ {a b : ℝ} → a +ℝ b ≡ b +ℝ a
  com×ℝ : ∀ {a b : ℝ} → a ×ℝ b ≡ b ×ℝ a
  zero+ℝ : ∀ {a : ℝ} → a +ℝ zeroℝ ≡ a
  one×ℝ : ∀ {a : ℝ} → a ×ℝ oneℝ ≡ a
  inv+-ℝ : ∀ {a : ℝ} → a +ℝ (-ℝ a) ≡ zeroℝ
  inv×÷ℝ : ∀ {a : ℝ} → a ×ℝ (÷ℝ a) ≡ oneℝ
  dist×+ℝ : ∀ {a b c : ℝ} → a ×ℝ (b +ℝ c) ≡ (a ×ℝ b) +ℝ (a ×ℝ c)
```

In order, these postulates correspond to the field properties

- Associativity of addition
- Associativity of multiplication
- Commutativity of addition
- Commutativity of multiplication
- zeroℝ is the additive identity
- oneℝ is the multiplicative identity
- -ℝ is the additive inverse
- ÷ℝ is the multiplicative inverse
- Multiplication distributes over addition

We can now use any of these definitions in our Agda programs, and although the current implementation, upon compilation, still only approximates the reals (and hence complex numbers), it should only be necessary to change the definition of the reals and define these axioms using that definition to give us a fully formally verified implementation of QIO.

We would now like to use the definitions given above in defining the field of complex numbers, ℂ. The definition of ℂ itself is exactly as would be expected.

```
data ℂ : Set where
  _:+_ : ℝ → ℝ → ℂ
```

and the functions representing addition and multiplication are defined in the normal way.

```
_+ℂ_ : ℂ → ℂ → ℂ
(a :+ b) +ℂ (a' :+ b') = (a +ℝ a') :+ (b +ℝ b')
_×ℂ_ : ℂ → ℂ → ℂ
(a :+ b) ×ℂ (a' :+ b') = ((a ×ℝ a') +ℝ (-ℝ (b ×ℝ b')))
                        :+ ((a ×ℝ b') +ℝ (b ×ℝ a'))
```

As the reals are a subset of the complex numbers, we can define a function ℝ→ℂ that lifts elements of ℝ into elements of ℂ with no imaginary part. We can then use this function in defining the additive and multiplicative identities of ℂ (zeroℂ and oneℂ respectively).

```
ℝ→ℂ : ℝ → ℂ
ℝ→ℂ a = a :+ zeroℝ
zeroℂ : ℂ
zeroℂ = ℝ→ℂ zeroℝ
oneℂ : ℂ
oneℂ = ℝ→ℂ oneℝ
```

Continuing on in a similar manner, we are able to define the additive inverse, and multiplicative inverse in terms of their ℝ counter-parts. We can also define all the necessary field axioms for ℂ using the underlying axioms from ℝ. Most of these definitions are omitted from here for brevity, but can be found in the on-line code [Gre09]. As a taster, we shall give the proof that multiplication is commutative. The type of such a proof is given by

```
com×ℂ : ∀ {a b : ℂ} → a ×ℂ b ≡ b ×ℂ a
```

but Agda is able to reduce this so that we have the following goal.

```
((a ×ℝ b) +ℝ -ℝ (ai ×ℝ bi)) :+ ((a ×ℝ bi) +ℝ (ai ×ℝ b))
                              ≡
((b ×ℝ a) +ℝ -ℝ (bi ×ℝ ai)) :+ ((b ×ℝ ai) +ℝ (bi ×ℝ a))
```

To make the definition of this proof easier, it is useful to have proof constructor functions that distribute equivalence over the :+ constructor of ℂ, and also over the addition operator _+ℝ_ (or indeed any binary operator acting on ℝ).

```
_≡:+≡_ : ∀ {a a' b b' : ℝ} → (a ≡ b) → (a' ≡ b')
              → (a :+ a') ≡ (b :+ b')
refl ≡:+≡ refl = refl
_≡[_]≡_ : {A : Set} → ∀ {a b c d : A} → (a ≡ c)
               → (_op_ : A → A → A) → (b ≡ d)
               → (a op b) ≡ (c op d)
refl ≡[_]≡ refl = refl
```

The proof of the commutativity of ×ℂ can now be given in terms of these functions, and the field axioms of ℝ that corresponds to the commutativity of ×ℝ and +ℝ (com×ℝ and com+ℝ respectively).

```
com×ℂ : ∀ {a b : ℂ} → a ×ℂ b ≡ b ×ℂ a
com×ℂ {a :+ ai} {b :+ bi} = (com×ℝ ≡[_+ℝ_]≡ cong -ℝ com×ℝ)
                                    ≡:+≡ (trans com+ℝ (
                            (com×ℝ ≡[_+ℝ_]≡ com×ℝ)))
```

Now that we have a type of complex numbers that we can use in Agda, we can go on to look at the new implementation of our unitaries. Such a definition of the complex numbers does lead to some problems later, as proof terms must be constructed explicitly for an entire calculation. We shall discuss the draw backs of this approach later.

10.2 A formally verified semantics for unitary operations

In the quantum realm we must model the semantics of our unitary operators in terms of functions acting on our model of quantum state. For the purposes of this implementation, we can think of a quantum state as a list of base states, which are each given a corresponding complex amplitude. A base state in this formulation can just be a vector of Booleans, along with its amplitude.

```
data Base (n : ℕ) : Set where
  _○_ : Vec Bool n → ℂ → Base n
```

The semantics of a QIO computation can now be given in terms of functions that take a base state, and return a list of base states. When running our computations, we shall simply map such a function over the entire state. However, this formalism brings a few problems when we wish to add proofs to the semantics that ensure that we do indeed have unitary operations. For example, we wish that applying a unitary operator to a single base state, and then mapping the inverse of the unitary operator over the list of base states returned by the original function, to leave us in the base state in which we started. This is already going to have a type mismatch, so we must define what it means for a list of base states to be equal to a given single base state. The easiest solution would be to check that the list is only a singleton list containing a single base state that is equivalent to the given base state, but in practice this approach isn't realistic as it involves defining a map function that is able to remove states that have a zeroℂ amplitude. With our current definition of ℂ we are unable to construct such a function as the system can't evaluate equivalences on ℂ, and the user must provide proofs constructed from the field properties. As such, a better definition of what it means for a base state to be equal to a list of base states is to ensure that every base state with a non-zero amplitude in the list will sum up to give the overall base state with which we are comparing it. To do this, we still need a way of removing base states from an overall state that have a zero amplitude, and can postulate an equivalence that defines this when given a proof that the amplitude does equal zeroℂ.

```
postulate
  removeZero : ∀ {n} → {x : Vec Bool n} → {c : ℂ}
    → {xs : List (Base n)}
    → (c ≡ zeroℂ) → (x ∘ c) :: xs ≡ xs
```

Defining the equivalence between a list of base states and a single base state can now be defined using a **record** type.

```
record _≡S_ {n : ℕ} (xs : List (Base n)) (x : Base n) : Set where
  field
    xs' : List (Base n)
    p : xs ≡ xs'
    bs : map (extractVecBool) xs'
      ≡ map (extractVecBool) (repeat x (length xs'))
    cs : foldr _+ℂ_ zeroℂ (map extractℂ xs')
      ≡ extractℂ x
```

Having the xs' field means that we are able to give a list of states which is equivalent to the list of states we are given. This is ensured by having to give a proof of such an equivalence (p). Normally, the only proof term we can use is the refl construct, and Agda can infer from this that xs' must be the list we are given. However, having to define xs' explicitly, means that we are able to provide a proof in terms of the removeZero equivalence function from above, and only have to give the following proofs for such a normalised state. The bs field is a proof that the classical state stored in every base state is equivalent to the classical state from the given base state. Finally, the cs field is a proof that the sum of all the amplitudes in the list of base states is equivalent to the amplitude of the single base state. A better implementation of the complex numbers would simplify this definition, as we would be able to define a filter function that removes base states with a zero amplitude automatically, instead of having to give proofs for such states explicitly.

Now that we have a definition for an equivalence between a single base state, and a list of base states, we are able to define the new USem structure that will provide the functions and proofs as in the classical case.

```
record USem (n : ℕ) : Set where
  field
    f : Base n → List (Base n)
    f⁻¹ : Base n → List (Base n)
    p : {x : Base n} → (join (map f⁻¹ (f x))) ≡S x
    p⁻¹ : {x : Base n} → (join (map f (f⁻¹ x))) ≡S x
```

The four fields in this new USem structure correspond exactly to the four fields in the classical USem structure, although now we are having to map the inverse function over the result of the first function, and use the new equivalence we defined above.

The USem structure can be defined as a monoid in a very similar manner to in the classical version, and as such the definitions shall be omitted here. We go on now to look at the extra structure that is associated with defining the unitaries that we have in QIO in terms of this new semantics.

10.3 Unitaries in QIO Agda

The definition for many of our unitaries are similar to their classical counterparts as we are able to just give a singleton list as the result of applying the functions to a single base state. The only exception to this is in defining the new type of rotations that we are allowed in the quantum version. Constructing the proofs for this new semantic is also a little harder than for the classical case as we have to look at the behaviour of the map function when we are defining our proofs. As such we shall go over the redefinition of the Swap gate in order to see the new proofs that we have to define.

The new definition of the Swap gate makes use of the underlying behaviour that was defined for the swap gates in the classical version. That is, we have the following functions:

```
swapQ' : ∀ {n} → (x y : Qbit n) → Vec Bool n → Vec Bool n
pswapQ' : ∀ {n} → (x y : Qbit n) → (xs : Vec Bool n)
  → swapQ' x y (swapQ' x y xs) ≡ xs
```

Indeed, the behaviour of the Swap gate that we wish to define now is just these functions lifted onto the list representation of quantum state. Firstly, the new behaviour of the swap function just returns the singleton list containing the base state that corresponds to swapping the qubits in the given argument base state.

```
swapQ : ∀ {n} → Qbit n → Qbit n → Base n → List (Base n)
swapQ q1 q2 (bs ∘ c) = [(swapQ' q1 q2 bs ∘ c)]
```

As Swap is self-inverse, we only require one proof. In this instance, we don't need to, and indeed would be unable to remove any zero amplitude base states before giving the proofs. As such we can give the xs' as the wild card pattern _, and Agda is able to use the proof refl to automatically infer that xs' is indeed just the input to the equivalence relation.

```
pswapQ : ∀ {n} → (x y : Qbit n) → (xs : Base n)
```

```
    → join (map (swapQ x y) (swapQ x y xs)) ≡S xs
pswapQ x y (y' ∘ y0) = record {xs' = _
                              ; p = refl
                              ; bs = cong (\x → x :: [])
                                       (pswapQ' x y y')
                              ; cs = zero+ℂ
                              }
```

The proof term bs uses the underlying pswapQ′ function lifted over a singleton list, and because of the behaviour of the foldr function we must give a proof that zeroℂ is the additive identity as our cs proof term. With the definition of both the swapQ function, and the relevant proof (pswapQ) we are able to give the full member of the USem datatype for the Swap constructor.

```
Swap : ∀ {n : ℕ} → (Qbit n) → (Qbit n) → USem n → USem n
Swap x y u = record {f = \xs → swapQ x y xs
  ; f⁻¹ = \xs → swapQ x y xs
  ; p = λ {xs} → pswapQ x y xs
  ; p⁻¹ = λ {xs} → pswapQ x y xs
  } • u
```

Before we go on to look at the definition of rotations, it is useful to look at the new definition of the Sep requirement. We can now think of a qubit as being separable from a given unitary if for every base state in the quantum state produced from running the unitary, the given qubit is in the same state as it started, and that the sum of the amplitudes of all these states is equivalent to the amplitude of the state in which it started. Indeed, this a very similar form of equivalence as we used above, but now restricted to only having a single bit in the classical state having to be equal.

```
record ≡Ss {n : ℕ} (xs : List (Base n)) (x : Base n)
  (q : Qbit n) : Set where
  field
    xs' : List (Base n)
    p : xs ≡ xs'
    bs : map (lookupQ q) xs'
      ≡ map (lookupQ q) (repeat x (length xs'))
    cs : foldr _+ℂ_ zeroℂ (map extractℂ xs')
      ≡ extractℂ x
record Sep {n : ℕ} (i : Qbit n) (u : USem n) : Set where
  field
    s : ∀ {x} → ≡Ss (USem.f u x) x i
    s⁻¹ : ∀ {x} → ≡Ss (USem.f⁻¹ u x) x i
```

10.3.1 Rotations

To define a unitary that represents a rotation, we must be able to define a function that in essence maps a single base state to the two possible base states that such a rotation can create, with their amplitudes updated accordingly. To ensure that these rotations are indeed unitary, we can define them such that they contain a proof that when multiplied by their conjugate transpose we arrive at the identity rotation. Having these proofs built into the rotations themselves means that we are able to use them within the proofs required by the USem structure. We will come back to the proofs that we require later.

Before looking at the proofs, we can define a rotation in terms of a **record** type containing 4 fields that each represent one of the entries of the matrix representation of such a rotation.

```
record Rot : Set where
  field
    a : ℂ
    b : ℂ
    c : ℂ
    d : ℂ
```

Using this representation we can already start to define some of the standard rotations we would like to use, although there is nothing to ensure that they are unitary at this stage. An example that we shall use through-out the rest of this section is that of the X rotation, which we shall also show is indeed a unitary rotation.

```
x' : Rot
x' = record {a = zeroℂ
            ;b = oneℂ
            ;c = oneℂ
            ;d = zeroℂ
            }
```

We are also able to define some standard matrix operations acting on these rotations, such as matrix multiplication,

```
_⋆_ : Rot → Rot → Rot
r1 ⋆ r2 = record {a = ((Rot.a r1) ×ℂ (Rot.a r2))
                      +ℂ ((Rot.b r1) ×ℂ (Rot.c r2))
                 ;b = ((Rot.a r1) ×ℂ (Rot.b r2))
                      +ℂ ((Rot.b r1) ×ℂ (Rot.d r2))
                 ;c = ((Rot.c r1) ×ℂ (Rot.a r2))
                      +ℂ ((Rot.d r1) ×ℂ (Rot.c r2))
                 ;d = ((Rot.c r1) ×ℂ (Rot.b r2))
                      +ℂ ((Rot.d r1) ×ℂ (Rot.d r2))
                 }
```

and the conjugate transpose operation.

```
_* : Rot → Rot
r * = record {a = conjugate (Rot.a r)
             ;b = conjugate (Rot.c r)
             ;c = conjugate (Rot.b r)
             ;d = conjugate (Rot.d r)
             }
```

Now we have our basic type for rotations, we want to think how we are able to define that they are unitary. The definition of unitary in this sense is that multiplying a matrix by its conjugate transpose must equal the identity matrix. As such, we can define that an arbitrary rotation (a : Rot) is unitary if we can show that a ⋆ (a *) is equal to the identity rotation (id' : Rot). To help with this process it is useful to define equivalence between rotations in terms of a **record** type that contains a proof that each field of a rotation is equivalent to the corresponding field in the other rotation.

```
record _≡R_ (r1 r2 : Rot) : Set where
  field
    a : Rot.a r1 ≡ Rot.a r2
    b : Rot.b r1 ≡ Rot.b r2
    c : Rot.c r1 ≡ Rot.c r2
    d : Rot.d r1 ≡ Rot.d r2
```

With this equivalence, we are able to define a datatype which encodes the requirement for a rotation to be unitary

```
Unitary : (r : Rot) → Set
Unitary r = (r ⋆ (r *)) ≡R id'
```

and define the Rotation datatype which contains a proof that the embedded rotation is indeed unitary.

```
data Rotation : Set where
  Rotate : (Σ Rot Unitary) → Rotation
```

We can now extend our example of the X rotation to give a formally verified unitary version. The proof xp : Unitary x' is quite large, so is omitted from here, but is available on-line [Gre09]. The proofs we need to construct, in general, boil down to using the field properties of ℂ that we defined earlier. Using a better implementation of ℂ would mean that Agda was able to evaluate the proof terms further, and we would be left with simpler and more concise proofs to construct.

```
x : Rotation
x = Rotate (x', xp)
```

We can now use these rotations in defining a member of the USem datatype that applies a rotation to a given qubit. The definition of the function that applies a rotation is given as follows:

```
rotate : ∀ {n} → (Qbit n) → Rotation
             → Base n → List (Base n)
rotate qzero
   (Rotate (r, p))
   ((false :: xs) ∘ c)
    =        ((false :: xs)) ∘ (Rot.a r ×ℂ c)
          :: ((true :: xs)) ∘ (Rot.b r ×ℂ c)
          :: []
rotate qzero
   (Rotate (r, p))
   ((true :: xs) ∘ c)
    =  (false :: xs) ∘ (Rot.c r ×ℂ c)
      :: (true :: xs) ∘ (Rot.d r ×ℂ c)
      :: []
rotate (qsuc i) r ((x :: xs) ∘ c) =
   map (\xs' → x ::' xs') (rotate i r (xs ∘ c))
```

With the definition of this function using the _×ℂ_ function to calculate the amplitudes of the two base states in its output, and the use of the conjugate transpose of a rotation as its inverse, the proofs that we require can be constructed using the underlying unitarity proofs of the rotation. The proofs are left out here as they are long constructions, but we can note that they must also make use of the removeZero function given previously to remove base states with a zero amplitude from the list of states calculated from mapping the inverse rotation to the result of the first rotation. The full code, including these proofs, is available on-line [Gre09].

```
URot : ∀ {n : ℕ} → (Qbit n) → Rotation → USem n → USem n
URot {n} q r u = record {f = rotate q r
                        ;f⁻¹ = rotate q (r *')
                        ;p = λ {xs} → rotateP q r xs
                        ;p⁻¹ = λ {xs} → rotatePR q r xs
                        } • u
```

We are now able to give all the unitary operations that we have defined, a syntax which we can use when writing unitary operations for use in our QIO computations. Firstly we use the same syntax as in the classical case for the uswap, ucond, and ulet operations

```
uswap : ∀ {n} → (Qbit n) → (Qbit n) → USem n
uswap q1 q2 = Swap q1 q2 UReturn

ucond : ∀ {n} → (i : Qbit n) → (Bool → (Σ (USem n) (Sep i)))
  → USem n
ucond q fb = Cond q fb UReturn

ulet : ∀ {n} → Bool → ((i : (Qbit (suc n)))
  → Σ (USem (suc n)) (Sep i)) → USem n
ulet b fq = Ulet b fq UReturn
```

and we can now introduce a urot operations for defining rotations, along with some standard rotations we are able to use.

```
urot : ∀ {n : ℕ} → (Qbit n) → Rotation → USem n
urot qn r = URot qn r UReturn

uId : ∀ {n} → Qbit n → USem n
uId q = urot q id

uX : ∀ {n} → Qbit n → USem n
uX q = urot q x

uHad : ∀ {n} → Qbit n → USem n
uHad q = urot q had

uPhase : ∀ {n} → ℝ → Qbit n → USem n
uPhase r q = urot q (phase r)
```

Now that we have defined our unitary operations, we are able to use them with QIO computations. We go over the definition of the actual QIO monad in the following section, and then look at how we are able to simulate the running of QIO computations by compiling our programs so they are able to use the underlying floating point representation of $\mathbb{R}$ and $\mathbb{C}$.

10.4 Evaluating quantum QIO computations using Agda

The definition of the QIO monad is identical to the previous definition we have given for the classical implementation of QIO in Agda. So in this section we shall just introduce the new evaluation functions that can be used to simulate the running of a QIO computation. The main difference from the classical implementation is that we must now map our new unitary operators over the quantum states we have defined. The map function that we have been using in the proofs doesn't combine base states that are equal as this is useful for the way we define our proof structures. However, when we come to actually wanting to run our programs, we are more interested in having a normalised output, in the sense that equal base states are combined by adding their amplitudes. For this purpose, we are able to define a new map function that keeps the state normalised by using a Boolean equality function for the classical part of the base states. We omit the definition of this new map′ function as it is pretty standard and corresponds well to the definition of the normalising vectors we used in the Haskell implementation. The other difference is that the type returned by the evaluation function can no longer be a pure value, and as such we shall return a list of the possible values from the type embedded in the monad, paired with the current state that produced that value.

```
runQIO : ∀ {n m A} → QIO A n m → List (Base n)
    → List (A × List (Base m))
runQIO (QReturn a) v = [a, v]
runQIO {n} {m} (MkQbit b fq) v = runQIO (fq (newQbit n))
                                         (map′ (\x → x ::ʳ′ b) v)
runQIO (ApplyU u q) v = runQIO q
                            (map′ (USem.f u) v)
runQIO (MeasQbit q fb) v =
        (runQIO (fb false)
              (filter (\b → ¬ (lookupQ q b)) v))
    ++ (runQIO (fb true)
              (filter (\b → (lookupQ q b)) v))
```

From this definition, we can see that applying a unitary uses the new map′ function, and that it is during measurements that the states can split and different values of the underlying type can occur. The evaluation of an entire QIO computation can now be defined by running it with an initial state that contains no elements.

```
run′ : ∀ {n A} → QIO A zero n (Qbit n)
    → List (A × List (Base n))
run′ q = runQIO q [[] ∘ oneℂ]
```

We would like the actual run function we define to have the same behaviour as the sim function that was defined in the Haskell version, so we need some way of extracting the probabilities for each member of the embedded type (a : A) from the state associated with that element. This is a simple task as it just involves summing the absolute squares of the amplitudes left in that base state. We can define a sum function that achieves this task

```
sum : ∀ {n} → List (Base n) → ℝ
sum [] = zeroℝ
sum (y ∘ y' :: xs) = [| y' |²] +ℝ sum xs
```

and then use it in the definition of a function that combines all the probabilities, along with their respective elements.

```
combine : ∀ {n : ℕ} → ∀ {A : Set} → List (A × List (Base n))
    → List (A × ℝ)
combine [] = []
combine ((a, []) :: xs) = combine xs
combine ((a, x :: xs) :: xs') =
   ((a, sum (x :: xs))) :: ((combine xs'))
```

The overall run function can now be defined:

```
run : ∀ {n : ℕ} → ∀ {A : Set} → QIO A zero n (Qbit n)
    → List (A × ℝ)
run q = combine (run' q)
```

As it stands, QIO programs written in Agda can be evaluated using the run function given above. However, these evaluations within Agda will just give us very large constructs that Agda has evaluated the QIO programs to, consisting of many operations on the ℝ datatype which it is unable to evaluate any further. If we want to actually get a result from running our QIO computations, then we must use the compiled version of QIO Agda that gives the programs a corresponding semantics using the underlying floating point representation of $\mathbb{R}$ that can be evaluated fully. The on line code [Gre09], contains example programs written in QIO Agda that can be compiled and run. These examples include an implementation of the creation of a bell state, an implementation of quantum teleportation, and an implementation of Deutsch's algorithm. Running the compiled version of these programs gives the output as expected. For example, the output from compiling and running the QIO computation that measures a bell state is as follows:

```
Qio.run measBell = ( false , false ):0.50000006 ,
                   ( true , true ):0.50000006
```

Even for such a *simple* construct, we can already see that the underlying floating point representation doesn't faithfully simulate the real numbers as the probabilities add up to more than one.

10.5 Remarks on QIO in Agda

The implementation of the classical subset of QIO in Agda has given us an excellent start towards developing a formally verified version of QIO, in the sense that; by having to define proofs of unitarity we ensure that all the computations we can define are indeed realisable computations in the standard model of quantum computation. Extending the classical version of QIO to a fully quantum version has shown some major draw backs, mainly coming about due to the lack of an implementation of the real numbers in the dependently-typed setting. The main draw backs are in the size and complexity of the proof terms that we need to provide, which would be a lot simpler if Agda could evaluate the real numbers to a normal-form and not just have them defined by all the operations that are used upon them. This problem also means that currently it is nearly impossible to reason in general about the QIO computations we are able to define, in the sense of creating formally verified proofs about the behaviour of quantum algorithms. For example, we would like to be able to write a proof term that shows Deutsch's algorithm acting on a function f is indeed equivalent to showing that the function f is balanced. However, there are also lots of positive results that can be taken from this implementation of QIO in Agda, as it shows that a dependently-typed approach can give us a nice way of defining quantum computations that are verified to have the behaviour that they specify.

There is still further work that could be done on an implementation of QIO in Agda, which would benefit greatly from a better definition of the real numbers. However, even without a new definition for the reals, there is still more work that could be done. In the short term, developing these ideas using the implementation of the classical subset of QIO is a better prospect, although any ideas shown in the classical version should extend naturally to their quantum counterparts. The final chapter goes on to give a brief comparison of the two implementations of QIO that I have presented in this thesis.

Chapter 11

QIO and other Quantum programming languages

11.1 Related Languages

In the introduction (Section 1.2.1), we introduced work that is related to the work in this thesis. As well as talking about related work on categorical models, and effectful programming in the functional setting; we also briefly mentioned a few quantum programming languages that are closely related to *QIO*:

- [Sab03] gives a model of quantum computing in Haskell that is designed to give programmers an intuitive approach to quantum computation
- [VAS06] looks at how quantum effects can be modelled in Haskell using the idea of *arrows*.
- [AG05] and [Gra06] introduce the functional quantum programming language QML. The denotational semantics of QML can be thought of in terms of the arrows approach mentioned above.

This chapter aims to give a more in depth look at the three languages mentioned above, with direct comparisons to the *QIO* approach. In so doing, I hope to give the reader a better understanding of where *QIO* fits within the realm of functional quantum programming languages, as well as discuss the pros and cons related with each approach.

11.2 Modelling quantum computation in Haskell

In [Sab03], the author goes over a model of quantum computation in Haskell. Although not fully developed as a language, the model provides programmers with an approach for developing quantum computations.

The approach starts by extending the idea of classical data types, to corresponding quantum data types, whereby the values that a classical data type can take form a basis for the quantum data type. A member of the quantum data type then consists of assigning a complex amplitude to each of the base states (that form a basis). The paper briefly mentions that this model can include certain forms of infinite data type, but to extend the model so as to allow computations on these quantum data types, we are restricted to finite (or enumerated) types. A simple example of a quantum data type in this model is the quantum analogue of *Bool*, which is denoted $QV\ \mathit{Bool}$, and elements can simply be defined by the amplitudes $\alpha, \beta :: \mathbb{C}$ which describe the quantum state $\alpha\ \ |\mathit{False}\rangle + \beta\ \ |\mathit{True}\rangle$. The type $QV\ \mathit{Bool}$ is simply a representation of a qubit, but any finite type can be lifted to its quantum counterpart directly. This contrasts with *QIO* in which at the lowest level, we only have qubits as a quantum counter part of *Bool*, and any other quantum data type must be constructed from this (E.g. the underlying structure of the quantum integer data type in *QIO* is a fixed-length list of qubits).

The interesting aspects of quantum computation only appear when the author starts to look at the quantum analogue of pairs, remarking that there are two types of pair that can be defined. Either $(QV\ a, QV\ b)$, which is just an (unremarkable) classical pair of quantum data types, or $QV\ (a, b)$ which is a quantum pair, and introduces the notion of entanglement. As an example, the author defines a Bell state, $(p_1 :: QV\ (\mathit{Bool}, \mathit{Bool}))$ and notes that it cannot be described in terms of its two sub components.

With the definition of quantum data types in place, it is then possible to start thinking of operations on these data types. It is quite simple to define operations on whole data types (E.g. not the sub components of a pair type for example) in terms of a matrix like structure that defines the probabilities of each base state being taken to a different base state (possibly of a different quantum type). The author briefly mentions how pure functions on the classical data types can be lifted to versions on the quantum data type, but leaves the side condition for the programmer to ensure that the functions they *lift* to be of a reversible nature.

Measurements on whole quantum data types are also quite straight forward to model, with a measurement collapsing the state into a single base state, or value of the underlying classical data type, and in so doing, changing the amplitude of all other base states to zero. This just models the probabilistic aspects of measurement operations and the use of the *IO* monad is sufficient. This is the natural choice in this instance, as it is possible to make use of *IORef*s so that quantum values can only be accessed via a reference, and observations can update the reference cell with a new value. Using the *IO* monad also follows from the same reasoning as for why the *run* function of *QIO* embeds the results within the *IO* monad.

It is when we wish to model computations, and measurements, on only sub elements of a quantum data type that the given representation becomes less elegant. With the model as it stands, quantum data types are only modelled as the wave functions that describe the entire state of the quantum type. For example, we have the quantum pairs as described previously, but have no way of accessing a single element of that pair. A nice remark, is that this is akin to the principle of wave/particle duality. The state is described by a wave function, but each element of the pair is an individual particle, which we should be able to treat individually in terms of applying operations to it, and measuring it. The effect of operations to these individual *particles* may well have an overall effect on the *wave function*, and hence the other particles with which it is entangled.

The paper goes on to describe a way in which to overcome this problem, using what it calls *Virtual Values.* A virtual value is described as being *a value which although possibly embedded deep inside a structure and entangled with others can be operated on individually.* In order to use a virtual value, the user must provide the entire data structure in which it is embedded, and an *adaptor* which is a pair of functions that map from the entire structure to the structure in question, and vice versa. The definition of these *adaptor* functions is shown to be regular, and the author provides examples that give access to the separate *particles* within a (possibly) entangled pair.

This is where the main contrast between the model presented here and *QIO* lies. Computations in *QIO* are defined upon individual qubits, and the wave functions describing the state are only simulated when running the computations. We will see in a comparison example towards the end of this section, that the quantum computations written in the model presented here incur a coding overhead in which all the sub-structures that need to be operated on must be extracted from the overall state that the computation uses.

Quantum computations written in this model end up defining effectful programs in Haskell, this is also different from the aim of the Quantum IO Monad. In *QIO*, we are able to write quantum computations, and then embed them into effectful Haskell programs if we wish to simulate the running of them. The aim here was to present a model of quantum computation to functional programmers, but the aim of *QIO* is to provide a syntax for defining quantum computations. *QIO* uses a monadic structure to be explicit about the effects that occur in quantum computation, but here, the *IO* monad is used so that the probabilistic aspects of measurement can be modelled within Haskell.

The paper finishes off with a few examples of quantum computations written using the model described. One of the examples presented is of Deutsch's algorithm, and it is nice to compare this with the implementation of Deutsch's algorithm presented in *QIO*. The following implementation of Deutsch's algorithm is lifted directly from [Sab03].

```
deutsch :: (Bool → Bool) → IO ()
deutsch f = do inpr ← mkQR (qFalse &* qTrue)
               let both = virtFromR inpr
                   top = virtFromV both ad_pair1
                   bot = virtFromV both ad_pair2
                   uf = cop f qnot_op
               in app1 hadamard_op top
                  app1 hadamard_op bot
                  app1 uf both
                  app1 hadamard_op top
                  topV ← observeVV top
                  putStr (if topV then "Balanced" else "Constant")
```

The first line creates the entire quantum data structure over which the computation will take place (*inpr*). In this case, it is defined as the tensor product (&∗) of the two states $|0\rangle$ (*qFalse*) and $|1\rangle$ (*qTrue*). The first three **let** expressions must then define the *virtual values* over which each stage of the computation can take place. The first creates a virtual value that contains both qubits from the overall state (*both*), the second creates a virtual value that contains the first qubit from the pair just defined (*top*), and finally the third creates a virtual value that contains the second qubit from the pair previously defined (*bot*). The final **let** statement defines a controlled operation that applies a not operation to its second argument, depending on the value of applying the function *f* to its first argument. The rest of the code looks quite similar to the definition of Deutsch's algorithm in *QIO* given below. Hadamards are applied to the top and bottom qubits, the unitary *uf* is applied to both qubits, and then the top qubit has another Hadamard applied to it before being measured. The resultant Boolean value specifies exactly if the input function was balanced.

```
deutsch :: (Bool → Bool) → QIO Bool
deutsch f = do x ← qPlus
               y ← qMinus
               applyU (cond x (λb → if f b then unot y
                                           else mempty))
               applyU (uhad x)
               measQ x
```

In *QIO* we define Deutsch's algorithm as a quantum computation that returns a Boolean value. We could embed the computation into an effectful Haskell computation by running it, and outputting the same message as in the definition above.

```
runDeutsch :: (Bool → Bool) → IO ()
runDeutsch f = do result ← run (deutsch f)
                  putStr (if result then "Balanced" else "Constant")
```

The monadic **do** notation in someway makes the definitions look very similar, with the main difference being that in *QIO* the qubits are distinct particles by definition, and we don't have to have explicit views of an overall state (the *virtual values*) in order to operate on them. The *QIO* definition just defines an operation on qubits, and it isn't until the computation is run that an overall state is even created. It is in this way that *QIO* is a language, giving a syntax for quantum computations that is separate from the underlying model that we use to evaluate the running of a computation.

The use of a monad in this model is to model the probabilistic nature of measurements, and not to directly model the side-effects inherent with measurements to the global state. Indeed, the monadic structure of *QIO* is quite the opposite, and is defined to directly model the side-effects inherent with measurement, and it is only upon simulation that different monadic structures are used (the two instances of the *PMonad* for *sim* and *run*) to model the probabilistic nature of the computations. Indeed, the model presented here struggles to combine measurements in-line with quantum computations, such as is necessary to model quantum teleportation. The model is adequate in the sense that any quantum computation can be re-defined in terms of a computation in which all measurements are deferred until the end, but this takes away the elegance of the quantum teleportation protocol as it no longer models the sending of classical data from Alice to Bob.

11.3 Monads and Arrows

The work presented in [VAS06] introduces a model of quantum computation that uses the idea of *arrows* from the functional programming paradigm, to model the effects that can occur in the quantum realm. Arrows were first introduced as a generalisation of monads [Hug00], and Haskell now provides a special syntax for arrows, along with a preprocessor built into GHC.

The main differences between the arrows approach, and the monadic approach of *QIO* is the semantics which they both use to represent quantum computations. *QIO* is built upon the notion of unitary operations and measurements being distinctly different types of operations. With unitary operations being pure, but measurements being effectful, as described by *QIO*s monadic structure. The arrows approach doesn't treat measurements and unitary operations as different types of operations, as both can be defined in terms of superoperators. It is these superoperators that form the arrows in the approach described.

The work is presented as a way of extending the previous model of quantum computation given in Haskell (in section 11.2). In fact, the teleportation protocol is presented as a motivating factor behind this, trying to define a model of quantum computation in which measurements can be modelled nicely as intermediary parts of a quantum computation. It is shown that a change in the semantic model is necessary so that quantum states are modelled in terms of density matrices.

Previously, the global state of a quantum system was modelled in terms of a pure state, that is, a single state that described the probabilities of a measurement of a system. Or in other words, the probabilities of the system collapsing into each of its base states upon measurement. Measurements were modelled by irreversible operations on these states that collapsed the wave-function of the system. Density matrices allow us to model the probability distribution of these so called pure states, and give us a way of modelling the different outcomes that may occur after a measurement. That is, a measurement collapses a pure state into another pure state, with a certain probability. A density matrix allows us to model all the pure states that can result from a measurement, along with their probabilities.

For example, pure states can be given as a density matrix by taking their outer product (with themselves). The density matrix of the pure state $|+\rangle$ is given as:

$$|+\rangle\langle+| = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix} [\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}] = \begin{bmatrix} \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & \frac{1}{2} \end{bmatrix}$$

However, density matrices can also describe mixed states, such as the probability distribution of states after measuring the $|+\rangle$ state. The following density matrix describes exactly the fact that measuring the $|+\rangle$ state leaves you either in the $|0\rangle$ state, or the $|1\rangle$ state each with probability $\frac{1}{2}$. It is also simply (a renormalisation of) the sum of the two pure states corresponding to $|0\rangle$ and $|1\rangle$.

$$Measure \begin{bmatrix} \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & \frac{1}{2} \end{bmatrix} = \begin{bmatrix} \frac{1}{2} & 0 \\ 0 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 0 & \frac{1}{2} \end{bmatrix} = \begin{bmatrix} \frac{1}{2} & 0 \\ 0 & \frac{1}{2} \end{bmatrix}$$

Operations that take a density matrix to another density matrix are known as superoperators. It is quite simple to see that unitary operations acting on pure states can be lifted to superoperators on density matrices, in a similar manner as to pure states being lifted to density matrices. However, it also possible to model measurement, and traces, as superoperators.

In Haskell, the standard model of pure types is defined using a type of vector that is, in essence, a function from elements (of a basis), to their corresponding probability amplitudes. Linear operators are then defined as functions from elements of a basis to vectors, which are applied to vectors using a form of monadic bind.

type $PA = Complex\ Double$

type $Vec\ a = a \rightarrow PA$

type $Lin\ a\ b = a \rightarrow Vec\ b$

It is this monadic structure that can be thought of as modelling the probabilistic nature of quantum computations, but it means that it is not straight forward to add another level of abstraction that models the effectful part of quantum computation that arises from measurement.

It has already been noted, that the Quantum IO monad gets around this problem as it is only modelling the effectful part of quantum computation, and not the probabilistic nature of running the computations. *QIO* embeds the running of a quantum computation into a different monadic structure if and when a user decides to simulate running the computation. Indeed, the monadic structure defined to model the probabilistic aspects of running a quantum computation would be unnecessary if we had a quantum machine on which to run the code, whereas the monadic structure of the code itself would still remain.

This is where the authors of [VAS06] argue that a change in the semantics to the density matrix and superoperator model can also be used to overcome this problem. In Haskell, they are able to define the type of density matrices as a specific instance of the previously defined *Vec* type.

type $Dens\ a = Vec\ (a, a)$

and a definition of superoperators that would follow from the previous definition for linear operators. That is, modelling superoperators as functions from pairs of elements, to Density operators, and then trying to define something similar to a monadic bind that can apply the superoperator to an entire density matrix.

type $Super\ a\ b = (a, a) \rightarrow Dens\ b$

It is when we look at the type of the bind operation required that we can start to notice that a monadic structure is somehow not *enough* to model such an operation.

$\gg= :: Dens\ a \rightarrow ((a, a) \rightarrow Dens\ b) \rightarrow Dens\ b$

It is noted that

> This type does not however correspond to the required type as computations now *consume multiple input values*

and how it is reminiscent of the original motivation behind Hughes' generalisation of monads to arrows ([Hug00]).

With the definitions for density matrices and superoperators in place, it is now possible to define the construction and composition of superoperators in terms of the constructors of the *Arrow* type class. A superoperator can be constructed from a pure function using the *arr* constructor, superoperators can be composed using the >>> constructor, and the *first* constructor is able to apply a superoperator only to the first component, leaving the second component unchanged.

In keeping with what is now the De Facto notation for arrow computations in Haskell ([Pat01]), quantum computations can be written in terms of superoperators being applied to an overall quantum state. As it was one of the motivating examples, we shall use the definition of teleportation to compare this approach to computations written in *QIO*.

The following definition of Teleportation using the arrows approach is taken directly from [VAS06]. It is split into three parts, corresponding to Alice's operations, Bob's operations, and the combination of the two to describe the entire teleportation protocol. As such, we shall compare each part with its corresponding *QIO* code.

```
alice :: Super (Bool, Bool) (Bool, Bool)
alice = proc (eprL, q) → do
            (q1, q2) ← (lin2super (controlled qnot)) ≺ (q, erpL)
            q2 ← (lin2super hadamard) ≺ q1
            ((q3, e2), (m1, m2)) ← meas ≺ (q2, e1)
            (m1′, m2′) ← trL ≺ ((q3, e2), (m1, m2))
            returnA ≺ (m1′, m2′)
```

Alice's part is defined as a superoperator between quantum values of $(Bool, Bool)$, although the output values are in a single base state due to the measurements. In *QIO* this is explicit in the type of Alice's computation, as her input is two qubits, and the output is a quantum computation that returns two Boolean values. As can be seen, the arrows approach also means that the result of applying an operation has to be labelled explicitly as a new variable, so the programmer has to keep track of all the variables within the computation. This also follows from the arrows approach being an extension to the original Haskell model presented previously, as computations are defined by having a global state that is updated by any operations. The *QIO* code for Alice's part is given below.

```
alice :: Qbit → Qbit → QIO (Bool, Bool)
alice aq eq = do applyU (ifQ aq (unot eq))
                 applyU (uhad aq)
                 measQ (aq, eq)
```

As the global state is hidden (or indeed, not even defined until runtime), we treat qubits as particles, and computations consist of operations on these qubits that don't explicitly update any form of global state. The other overhead in the arrows approach comes from the use of density matrices, as after measurement, the now extraneous quantum information must be traced out of the overall state. That is, that a measurement creates a density matrix that contains the measurement results, as well as the remaining quantum states that such a measurement would leave. These quantum states correspond to the states left after the side-effects of measurement have occurred.

We can now look at Bob's part of the protocol.

```
bob :: Super (Bool, Bool, Bool) Bool
bob = proc (erpR, m1, m2) → do
        (m2′, e1) ← (lin2super (controlled qnot)) ≺ (m2, eprR)
        (m1′, e2) ← (lin2super (controlled z)) ≺ (m1, e1)
        q′ ← trL ≺ ((m1′, m2′), e2)
```

$returnA \prec q'$

Bob's part is a super-operator from a quantum value of type $(Bool, Bool, Bool)$ to a single quantum value of type $Bool$. Again, the type of the operator doesn't give us such a good intuition into what is occurring. One of Bob's inputs is indeed in a quantum state, but the other two values are the classical data from Alice. Compare this with the type of Bob's function written in QIO.

```
bob :: Qbit → (Bool, Bool) → QIO Qbit
bob eq (a, b) = do applyU (if b then (unot eq) else mempty)
                   applyU (if a then (uZ eq) else mempty)
                   return eq
```

The rest of the definitions are very similar, although again, an explicit trace must be made to leave only the single quantum value as required. So as to have a complete example, we also include the combination of Alice and Bob's parts in the overall teleportation protocol.

```
teleport :: Super (Bool, Bool, Bool) Bool
teleport = proc (erpL, eprR, q) → do
               (m1, m2) ← alice ≺ (eprL, q)
               q' ← bob ≺ (erpR, m1, m2)
               returnA ≺ q'
teleportation :: Qbit → QIO Qbit
teleportation iq = do (eq1, eq2) ← bell
                      cd ← alice iq eq1
                      tq ← bob eq2 cd
                      return tq
```

The arrows approach gives a nice way of extending the original model of quantum computation given in Haskell ([Sab03]) so that it explicitly models the side effects of measurement as well as the probabilistic nature of the values returned by a measurement. However, it is still modelling quantum computation in Haskell, and as such has quantum states explicit in the syntax. QIO is intended to be more of a language for defining quantum computations, that is separate from the underlying model of quantum computation. That is, computations written in *QIO* can be thought of as programs that can be interpreted into any model of quantum computation, whereas computations written using this arrows approach are hard-wired into the underlying model of quantum computation defined in Haskell.

The arrows approach presented here, also has a few draw backs. The main one of which is that it is possible to define arrows that don't give rise to physically realisable superoperators. This is because there is no way of controlling *weakening.* For an arrow to be a superoperator, it is necessary that every variable within the arrow is used. As an example, it is totally acceptable in Haskell to write Bob's operation as follows:

```
bob :: Super (Bool, Bool, Bool) Bool
bob = proc (erpR, m1, m2) → do
            (m2', e1) ← (lin2super (controlled qnot)) ≺ (m2, eprR)
            (m1', e2) ← (lin2super (controlled z)) ≺ (m1, e1)
            returnA ≺ e2
```

whereby we haven't explicitly traced out the other values, before returning *e2*. As such, this isn't a physically realisable superoperator, even though it may seem *logically* correct. It is noted therefore, that this model of quantum computation may be better off being treated as a model of quantum computation into which other languages can be compiled. Specifically languages in which weakening is dealt with explicitly. As such, we move on now to look at *QML*, a functional quantum programming language that deals with weakening explicitly.

11.4 QML

QML ([AG05, Gra06]), is presented as a functional quantum programming language that allows both quantum and classical control. Previous quantum programming languages had been presented that allowed classical control, whereby a quantum value is measured, and the corresponding classical value can be used to decide which path of a computation is executed. In introducing quantum control, QML allows a form of quantum if structure (denoted $\mathbf{if}^{\circ}$), whereby a superposition can be used as a first class term, allowing orthogonal branches of the computation to both contribute to the result. This form of quantum control lead to the design of the conditional operation available in *QIO*, in which the state of a qubit can be used as the control in a conditional. In fact, the orthogonality conditions that must hold for the branches of an $\mathbf{if}^{\circ}$ operation in QML, are satisfied in *QIO* by the constraint that the state of the control qubit must not be changed by either of the branches.

The syntax of QML is designed so as to be similar to other functional programming languages, and is based on strict linear logic so as to keep weakenings explicit. The choice of semantics for QML also means that reversible, and irreversible computations can be defined, and the semantics is able to embed the irreversible computations into a larger reversible structure. This is very similar to the techniques described in [GA08], and in fact, the operational semantics are based on the categorical model *FQC*. That is, that the compiler for QML translates functions written in QML into typed quantum circuits as defined in the category *FQC*. The compiler actually uses *FQC* as an intermediary structure, whereby a further denotational semantics can be chosen as the final output of a compilation. The choice of denotational semantics includes unitary matrices, isometries, or superoperators.

QML is developed as a fully independent language, with a compiler written in Haskell. This has the advantages that the language design doesn't depend on any underlying parent language, and also means that the language designer has complete control over the types, and constructs available in the language. There are some draw backs to this, in that there is also no extra structure provided for free by a parent language, such as all the classical types and functions that we can use in *QIO* because they form part of standard Haskell. However, anything that is essential can be added to the language, and it is argued in [Gra06] that a better model of classical types and structures in QML would be a nice feature. For example, the following code (taken from [Gra06]) forms a variant of Deutsch's algorithm written in QML.

```
$deutsch \in \mathcal{Q}_2 \multimap \mathcal{Q}_2 \multimap \mathcal{Q}_2$
$deutsch\ a\ b = \mathbf{let}\ (x, y) =$
                 $\mathbf{if}^\circ\ \mathbf{qfalse} + \mathbf{qtrue}$
                 $\mathbf{then}\ (\mathbf{qtrue}, \mathbf{if}^\circ\ a$
                         $\mathbf{then}\ (\mathbf{qfalse} + (-1) * \mathbf{qtrue}, (\mathbf{qtrue}, b))$
                         $\mathbf{else}\ \ ((-1) * \mathbf{qfalse} + \mathbf{qtrue}, (\mathbf{qfalse}, b)))$
                 $\mathbf{else}\ (\mathbf{qfalse},\ \mathbf{if}^\circ\ b$
                         $\mathbf{then}\ ((-1) * \mathbf{qfalse} + \mathbf{qtrue}, (a, \mathbf{qtrue}))$
                         $\mathbf{else}\ \ (\mathbf{qfalse} + (-1) * \mathbf{qtrue}, (a, \mathbf{qfalse})))$
               $\mathbf{in}\ had\ x$
```

There is a lot of extra structure involved that could be removed if there was a better model of classical data in QML, and indeed this variant of Deutsch's algorithm is a test on the equality of two qubits (which are both guaranteed to be in a computational base state) as the classical types available in the current implementation of QML aren't sufficient to define Deutsch's algorithm in its more traditional guise of testing whether a given function is balanced. In ([Gra06]), Grattage goes on to give a simplified version of this algorithm, that assumes a slightly better classical structure in QML. We shall give this here, and look at what is going on in more detail.

```
$deutsch \in Bool \multimap Bool \multimap \mathcal{Q}_2$
$deutsch\ a\ b = \mathbf{let}\ (x, y) = \mathbf{if}^\circ\ \mathbf{qfalse} + \mathbf{qtrue}$
                          $\mathbf{then}\ (\mathbf{qtrue}, \mathbf{if}\ a$
                                  $\mathbf{then}\ \mathbf{qfalse} + (-1) * \mathbf{qtrue}$
                                  $\mathbf{else}\ \ (-1) * \mathbf{qfalse} + \mathbf{qtrue})$
                          $\mathbf{else}\ (\mathbf{qfalse},\ \mathbf{if}\ b$
```

$$\begin{aligned}&\quad\quad\quad\quad\quad\quad\quad\quad \textbf{then } (-1) * \textbf{qfalse} + \textbf{qtrue}\\ &\quad\quad\quad\quad\quad\quad\quad\quad \textbf{else } \ \textbf{qfalse} + (-1) * \textbf{qtrue})\\ &\quad \textbf{in } \mathit{had}\ x\end{aligned}$$

The outer **if**$^\circ$ statement is using an equal superposition of **qfalse**, and **qtrue** to run both branches of the computation. The first branch adds a negative phase to the **qtrue** part of an equal superposition if the input a is *True*, and adds a negative phase to the **qfalse** part of an equal superposition if the input a is *False*. The second branch does the opposite depending on the input value b. If the input Booleans are equal, then the overall state is left in $\pm$ (**qfalse** $+ (-1) *$ **qtrue**, **qfalse** $+ (-1) *$ **qtrue**), whereas if the values aren't equal the overall state is left as $\pm$ (**qfalse**$+$**qtrue**, **qfalse**$+(-1)*$**qtrue**). The final Hadamard operation will therefore take either of the equal cases to $\pm$ **qtrue**, and the non-equal cases to $\pm$ **qfalse**.

Another nice example of QML is that of teleportation, taken from ([Gra08]). I have changed the names of some of the functions so that they are easier to compare with the *QIO* implementation of teleportation, given previously.

$$\begin{aligned}&\mathit{Had}, \mathit{Qnot}, \mathit{Meas} \ \in \ \mathcal{Q}_\mathbf{2} \multimap \mathcal{Q}_\mathbf{2}\\ &\mathit{Had}\ b = \textbf{if}^\circ\ b \textbf{ then qfalse} + -\textbf{qtrue}\\ &\quad\quad\quad\quad\quad \textbf{else qfalse} + \textbf{qtrue}\\ &\mathit{Qnot}\ b = \textbf{if}^\circ\ b \textbf{ then qfalse else qtrue}\\ &\mathit{Meas}\ b = \textbf{if } b \textbf{ then qtrue else qfalse}\\ &\mathit{CNot} \ \in \ \mathcal{Q}_\mathbf{2} \multimap \mathcal{Q}_\mathbf{2} \multimap \mathcal{Q}_\mathbf{2} \ \otimes \ \mathcal{Q}_\mathbf{2}\\ &\mathit{CNot}\ s\ t = \textbf{if}^\circ\ s \textbf{ then } (\textbf{qtrue}, \mathit{Qnot}\ t)\\ &\quad\quad\quad\quad\quad\quad \textbf{else } (\textbf{qfalse}, t)\\ &\mathit{Bell} \ \in \ \mathcal{Q}_\mathbf{2} \ \otimes \ \mathcal{Q}_\mathbf{2}\\ &\mathit{Bell} = (\textbf{qtrue}, \textbf{qtrue}) + (\textbf{qfalse}, \textbf{qfalse})\\ &\mathit{Alice} \ \in \ \mathcal{Q}_\mathbf{2} \multimap \mathcal{Q}_\mathbf{2} \multimap \mathcal{Q}_\mathbf{2} \ \otimes \ \mathcal{Q}_\mathbf{2}\\ &\mathit{Alice}\ x\ y = \textbf{let } (x', y') = \mathit{CNot}\ x\ y\end{aligned}$$

$$\textbf{in} \quad (\mathit{Meas}\ (\mathit{Had}\ x'), \mathit{Meas}\ y')$$

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

$$\mathit{Bob}\ q\ xy = \textbf{let}\ (x, y) = xy\ \textbf{in if}\ x\ \textbf{then}\ (\textbf{if}\ y\ \textbf{then}\ U_{11}\ q\ \textbf{else}\ U_{10}\ q)$$
$$\textbf{else}\ (\textbf{if}\ y\ \textbf{then}\ U_{01}\ q\ \textbf{else}\ q)$$

$$U_{01}, U_{10}, U_{11} \in \mathcal{Q}_{\mathbf{2}} \multimap \mathcal{Q}_{\mathbf{2}}$$

$$U_{01}\ x = \textbf{if}^{\circ}\ x\ \textbf{then qfalse else qtrue}$$

$$U_{10}\ x = \textbf{if}^{\circ}\ x\ \textbf{then} - \textbf{qtrue else qfalse}$$

$$U_{11}\ x = \textbf{if}^{\circ}\ x\ \textbf{then} - \textbf{qfalse else qtrue}$$

$$\mathit{Tele} \in \mathcal{Q}_{\mathbf{2}} \multimap \mathcal{Q}_{\mathbf{2}}$$

$$\mathit{Tele}\ q = \textbf{let}\ (a, b) = \mathit{Bell}$$
$$cd = \mathit{Alice}\ q\ a$$
$$\textbf{in}\ \mathit{Bob}\ b\ cd$$

Again, the lack of classical types means that the types of the teleportation functions are unable to tell us that it is indeed classical data that Alice is creating, although if we look at the procedure we do know that the pair of qubits that Alice creates are in a base state as they are measured. It is also noticeable how QML defines computations over quantum states, and not qubits. That is, QML treats qubits directly as a two-level quantum state, and not a particle which has that state as a property.

QML is a functional quantum programming language that is designed to model the higher-level aspects of quantum computation. This is also the case of the previous two models of quantum computation that have been presented in this chapter. As such, they aim to abstract away from the lowest level of computations as unitary operations occurring on individual qubits. Instead they are designed to model quantum computations at a higher level, and as operations on larger quantum structures. In contrast, *QIO* has been designed as a low level language, although you are able to define operations that act as an interface to the higher level constructs modelled in these languages.

11.5 Conclusions

Having given a more in depth introduction to some of the other works that are closely related to *QIO*, along with some comparisons to *QIO* itself, I would like to finish this chapter with some conclusions, and closing remarks on how *QIO* places in the current landscape of functional quantum programming languages.

QIO as it currently stands, is an embedded language for defining quantum computations within the parent language. In this case, the parent language is either Haskell or Agda. This gives us a greater opportunity to use the abstractions available in these well developed functional programming languages *in situ* with the extra constructions defined for *QIO*. A nice example that shows the benefits of such an embedding is the definition of the *Qdata* class in Haskell, where we actually use Haskell's class system to give extra structure to the quantum computations we can define in *QIO*.

Many of the functional quantum programming languages that currently exist are designed to model the higher-level aspects of quantum computation. That is, they are designed so as to abstract away from the low level aspects of quantum computations being defined as unitary operations occurring on qubits. In order to do this, many of them treat quantum states as first class entities, and computations are defined as unitary operations occurring on these states. However, *QIO* is designed from the opposite perspective, where at the lowest level, we must define everything in terms of operations on qubits. However, we are also able to use the language tools available to us (from the parent languages) to create functional abstractions that model the same high-level aspects as are inherent in the other languages.

One advantage of having *QIO* as a low-level language, is that it could be treated as the operational semantics for high-level languages. However, as we can already use functional abstractions with-in *QIO*, we can also think of *QIO* computations in terms of these higher-level structures directly. Another advantage of *QIO*'s low-level approach is that it makes reasoning about *QIO* programs an easier prospect. For example, in the Agda implementation of *QIO*, we are able to reason about *QIO* programs in the same way as any other Agda programs, in that the dependent type system allows us to embed proofs of certain specifications into our code.

Chapter 12

Discussion and Conclusions

12.1 Comparing the Haskell and Agda implementations

In this thesis, I have presented my work on the Quantum IO Monad, with a full implementation written in the functional programming language Haskell, and a further development of the QIO Monad written in the dependently-typed (functional) language Agda. The Quantum IO Monad was designed as an interface to quantum computation, with the view that it could be developed into a full quantum programming language. The choice of writing QIO in Haskell was originally based on Haskell's monadic approach to effectful programming, which leads to a very nice setting for a quantum language as the side-effects from measurement have to be dealt with explicitly in the design of the language. Using Haskell also meant that we could work with QIO as an embedded language, or more accurately as a Haskell library, and use all the language tools that Haskell has available for it. However, although the implementation defines a fully universal quantum programming language, the type system of Haskell isn't expressive enough to ensure the unitarity of all the operators defined within QIO. This lead to our quantum simulator functions having to throw runtime errors, which would not necessarily be possible on an actual quantum machine. The move to Agda meant that these checks for unitarity could be moved to the type-level, and thus would be caught at compile time, before the code could ever be run on a quantum machine. The work on QIO in Agda presented in this thesis goes some way towards a full reimplementation of QIO which contains formal proofs of the unitarity of the computations that can be defined, although the proofs that we must currently provide are overly long and complicated due to the lack of an implementation of the real numbers in Agda. Despite this drawback, the implementation provided does represent a fully universal library of quantum computation within Agda, in keeping with the monadic design of the original Haskell implementation. Indeed, we are able to write QIO computations in Agda in a very similar manner to the examples given in Haskell, along with the necessary proofs. These computations can then be compiled into executable code that simulates the running of the quantum computations using a floating point representation of the real numbers. The on-line code repository [Gre09] currently contains implementations of a few of the simplest quantum algorithms written in QIO Agda, along with an example program that can be used to compile QIO computations. It is expected that more examples will be added to this repository as they are implemented.

The proofs we define are indeed proofs of the unitarity of the operators in QIO Agda, despite the current implementation meaning they are overly long and complicated. This means that code can be written using the monadic behaviour of QIO to explicitly deal with any side-effects that may occur due to measurement, that is verified to represent a computation that is realisable in the standard model of quantum computation. This verification is likely to be a major issue when we actually have quantum machines that can run computations, as any un-unitary behaviour cannot be modelled physically. As such, QIO Agda can be thought of as a proof of concept that suitable quantum programming languages can be developed.

QIO Agda also goes someway towards showing that we can provide a language in which new quantum algorithms can be developed. This arises from being able to give proofs that any algorithms that are developed are indeed an implementation of their specification. These proofs may appear to have to include a classical simulation of the defined algorithm, but often the proofs can be defined for small, manageable sizes of input, and then extended automatically for arbitrarily large inputs.

12.2 Further Work

There is further work that can be done in many of the areas presented in this thesis. The three equivalence laws presented in Chapter 3 have not been shown to be complete, in the sense that all equivalences can be given just in terms of these three laws. Further work in this area has been suggested, such as looking at the functoriality of the circuits that we can define to give a better definition of equivalence [YY09].

The work on QIO in Haskell gives a nice syntax to quantum computations, despite the drawbacks that have been discussed. The semantics of these computations comes directly from the quantum gate model, and indeed the choice of constructs available in QIO is based heavily on a universal family of circuits presented in the categorical model $\mathbf{FxC}^{\simeq}$. However, recent work has shown that other models of quantum computation, such as the one-way model of quantum computation [BB06] could give rise to a more physically realisable implementation of a quantum computer. As such, a nice follow up to QIO in Haskell would be to re-implement it such that the semantics are based on the measurement calculus [DKP07], which is a calculus that describes the behaviour of one-way quantum computation.

The work in this thesis that could be extended on the most is the implementation of QIO in Agda. As it stands, it is more of a proof of concept than a usable system for defining formally verified quantum computations. The main area that needs to be tackled to give a better implementation of QIO in Agda is in defining a type of real numbers. The work in [GN02] develops a construction of the real numbers in Coq, which is a proof assistant that can be thought of as a dependently typed language. A reimplementation of this work in Agda could provide a better starting point for constructing the formal proofs needed in our implementation, although there is other work ongoing in developing an implementation of the real numbers in Agda. Even with a better definition of the real numbers in Agda, there is still more work that could be done to improve it. For instance, a few changes to the language syntax would enable us to reference qubits explicitly in our monadic construct that applies a unitary, which would allow Agda to automatically lift the qubits in scope in a similar manner to the weakening of references in [Swi08] chapter 6.

12.3 Conclusions

This thesis has presented my research into the field of quantum computation from the perspective of a functional programmer. Along with my research I have also tried to present introductions to each of the main subject areas that are covered. That is, I have given an introduction to quantum computation that covers the main aspects of qubits, superposition, entanglement, unitary operations, and measurement. In the measurement section I have focused specifically on the side-effects that can be caused when measuring entangled systems. I have also given an introduction to functional programming, and more specifically the language Haskell. It was my aim, in presenting an introduction to functional programming, to give an intuition into the use of monadic constructs in Haskell to explicitly deal with effectful computation. The last subject introduced is that of dependently-typed programming, specifically in the language Agda. The introduction covers the topic of indexed monads, and also looks at how programs in Agda can be thought of as formal proofs that are verified by the type-checker.

The bulk of this thesis presents my work on a monadic interface to quantum computation, known as the Quantum IO monad, or QIO. The monadic approach taken is unique amongst the languages currently available for quantum computation, as it keeps the behaviour of side-effects that may arise from measurements explicit within the language constructs. The work uses many ideas from previous quantum languages, such as the use of quantum control structures that were first developed for QML ([AG05, Gra06]). Many of these language constructs have been formalised in a categorical setting, such as the category of quantum circuits that has also been presented.

The development of QIO as a monadic structure followed directly from the use of monads in Haskell to define any form of effectful computation. Recent work on reasoning about monadic computations ([SA07, Swi08]) has been a big influence on the design of the monadic constructs for QIO, and indeed many of the ideas on reasoning about monadic computations in Agda, presented in [Swi08], have also been used when reimplementing QIO in a dependently-typed setting.

Using a monadic approach in modelling quantum computation gives us some advantages over other approaches. Firstly, the subject of monads is well developed, especially in the area of functional programming, where monads are used to model any form of effectful computation. Secondly, the whole monadic approach is designed to give a model of computation in which effects are explicit. Quantum computation is, by the very nature of measurements, an effectful form of computation, and it is therefore very natural to think of it in terms of a monadic structure. Having a model of quantum computation in which the side-effects of computation are explicitly modelled also gives us a frame work in which we can start to reason about our computations. This comes about, as we are able to use the monadic structure to give *meaning* to the side-effects that may occur.

Using a monadic approach to model the side-effects of measurement also means that we are able to move the development of unitary operators to a separate monoidal construct, and only worry about effectful parts of the computation when we introduce measurements. Indeed, many of the example programs given for QIO are first defined in terms of unitary operators, and then used in a specific computation in which initialisations and measurements can also occur. Having the monoidal structure of unitaries separate from the monadic structure of computations means that we are able to reason about our programs in terms of the behaviour of the unitary operators. In QIO Agda, we have formalised this idea, in the sense that every unitary structure that we are able to define comes with a corresponding proof that it is indeed unitary. These proofs can then be used within the definition of a monadic quantum computation to give a formal verification that the computation fulfils its specification.

As physical realisations of quantum computers develop further, it will be important that quantum programs can be shown to satisfy a given specification. Having a system that enables the verification of quantum programs, classically, may be a major benefit as the cost of quantum machines would mean that running quantum code is quite a commodity. Verifying code classically before it is ever run on a quantum system will help ensure that the resource of quantum machines isn't wasted on running incorrect programs.

12.4 Final remarks

The research I have done for this thesis has given me a great insight into the world of quantum computation, and has only helped to whet my appetite for the subject and surrounding areas. There is plenty more work that could be done with QIO, to develop it into a language that could be used for the quantum computers of the future. The subject area itself is full of many interesting people, and is a perfect avenue for greater collaboration between different subject areas that are working toward similar goals. It is this ongoing collaboration between physicists, computer scientists, and mathematicians etc. that may one day provide us with quantum systems capable of computation in this style. It is my hope, especially as a computer scientist, that work similar to the work presented in this thesis, shall help to contribute towards having languages for these systems free from all the bugs and subsequent crashes etc. that plague the world of classical computers in the current era. I would like to thank anyone who has taken the time to read this thesis, and hope that it has been a valuable insight into my research.

Bibliography

[AC03] Samson Abramsky and Bob Coecke. Physical traces: Quantum vs. classical information processing. In *Proceedings of the 9th Conference on Category Theory and Computer Science (CTCS 2002)*, volume 69 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 2003. Also arXiv:cs.CG/0207057.

[AC04] Samson Abramsky and Bob Coecke. A categorical semantics of quantum protocols. In *Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science (LICS)*. IEEE Computer Society, 2004. Also arXiv:quant-ph/0402130.

[AD08] P. Arrighi and G. Dowek. Linear-Algebraic Λ-Calculus: Higher-Order, Encodings, and Confluence. In *Rewriting Techniques and Applications: 19th International Conference, RTA 2008 Hagenberg, Austria, July 15-17, 2008, Proceedings*, page 17. Springer, 2008.

[AG05] Thorsten Altenkirch and Jonathan Grattage. A functional quantum programming language. In *Proceedings of the 20th Annual IEEE Symposium on Logic in Computer Science (LICS)*. IEEE Computer Society, 2005. Also arXiv:quant-ph/0409065.

[AG10] Thorsten Altenkirch and Alexander Green. The Quantum IO Monad. In Simon Gay and Ian Mackie, editors, *Semantic Techniques in Quantum Computation*. Cambridge University Press, 2010. ISBN13-9780521513746.

[BB06] Dan E. Browne and Hans J. Briegel. One-way quantum computation - a tutorial introduction, 2006.

[BCDP96] David Beckman, Amalavoyal N. Chari, Srikrishna Devabhaktuni, and John Preskill. Efficient networks for quantum factoring. *Phys. Rev. A*, 54(2):1034–1063, Aug 1996.

[Bel64] John S. Bell. On the Einstein–Podolsky–Rosen paradox. *Physics*, 1(??):195–200, 1964.

[CDKM04] Steven A. Cuccaro, Thomas G. Draper, Samuel A. Kutin, and David Petrie Moulton. A new quantum ripple-carry addition circuit, 2004.

[Coe05] Bob Coecke. Kindergarten quantum mechanics, 2005.

[Deu85] D. Deutsch. Quantum Theory, the Church-Turing Principle and the Universal Quantum Computer. *Proceedings of the Royal Society of London. Series A, Mathematical and Physical Sciences*, 400(1818):97–117, 1985.

[Dir82] P. A. M. Dirac. *The Principles of Quantum Mechanics (International Series of Monographs on Physics).* Oxford University Press, USA, February 1982.

[DJ92] D Deutsch and R Jozsa. Rapid solution of problems by quantum computation. *Proc Roy Soc Lond A*, 439:553–558, October 1992.

[DKP07] Vincent Danos, Elham Kashefi, and Prakash Panangaden. The measurement calculus. *Journal of the ACM*, 54(2), 2007. Preliminary version in arXiv:quant-ph/0412135.

[Dra00] Thomas G. Draper. Addition on a quantum computer, 2000.

[EPR35] A. Einstein, B. Podolsky, and N. Rosen. Can quantum-mechanical description of physical reality be considered complete? *Physical Review*, 47:777–780, May 1935.

[Eve57] H. Everett. 'Relative State' formulation of quantum mechanics. *Reviews of Modern Physics*, 29:454–462, 1957.

[Fey82] Richard Feynman. Simulating physics with computers. *International Journal of Theoretical Physics*, 21:467–488, 1982.

[GA08] Alexander S. Green and Thorsten Altenkirch. From reversible to irreversible computations. *Electronic Notes in Theoretical Computer Science*, 210:65 – 74, 2008. Proceedings of the 4th International Workshop on Quantum Programming Languages (QPL 2006).

[Gay06] Simon J. Gay. Quantum programming languages: Survey and bibliography. *Mathematical Structures in Computer Science*, 16(4), 2006.

[GN02] Herman Geuvers and Milad Niqui. Constructive reals in coq: Axioms and categoricity. In *TYPES '00: Selected papers from the International Workshop on Types for Proofs and Programs*, pages 79–95, London, UK, 2002. Springer-Verlag.

[Gos98] Phil Gossett. Quantum carry-save arithmetic, 1998.

[Gra06] J. J. Grattage. *QML: A functional quantum programming language.* PhD thesis, The University of Nottingham, 2006.

[Gra08] Jonathan Grattage. An overview of qml with a concrete implementation in haskell, 2008.

[Gre09] Alexander S. Green. The Quantum IO Monad, source code and example computations (for both haskell and agda implementations). http://www.cs.nott.ac.uk/~asg/QIO/, 2009.

[Gro96] Lov K. Grover. A fast quantum mechanical algorithm for database search. In *STOC '96: Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pages 212–219, New York, NY, USA, 1996. ACM.

[HRP$^+$06] P A Hiskett, D Rosenberg, C G Peterson, R J Hughes, S Nam, A E Lita, A J Miller, and J E Nordholt. Long-distance quantum key distribution in optical fibre. *New Journal of Physics*, 8(9):193, 2006.

[Hug00] John Hughes. Generalising monads to arrows. *Sci. Comput. Program.*, 37(1-3):67–111, 2000.

[Hut07] Graham Hutton. *Programming in Haskell.* Cambridge University Press, January 2007.

[Jon03] Simon P. Jones. *Haskell 98 Language and Libraries: The Revised Report.* Cambridge University Press, May 2003.

[Kar03] Jerzy Karczmarczuk. Structure and interpretation of quantum mechanics — a functional framework. In *Proceedings of the ACM SIGPLAN Workshop on Haskell.* ACM Press, 2003.

[Lan00] R. Landauer. Irreversibility and heat generation in the computing process. *IBM Journal of Research and Development*, 44(1):261–269, 2000.

[Mac71] Saunders MacLane. *Categories for the working mathematician.* Springer-Verlag, New York, 1971. Graduate Texts in Mathematics, Vol. 5.

[MB01] Shin-Cheng Mu and Richard Bird. Functional quantum programming. In *Proceedings of the 2nd Asian Workshop on Programming Languages and Systems*, 2001.

[McB99] Conor McBride. *Dependently Typed Functional Programs and their Proofs.* PhD thesis, University of Edinburgh, 1999.

[McB02] Conor McBride. Faking It (Simulating Dependent Types in Haskell). *Journal of Functional Programming*, 12(4& 5):375–392, 2002. Special Issue on Haskell.

[Mer85] David N. Mermin. Is the moon there when nobody looks? reality and the quantum theory. *Physics Today*, 38(4):38–47, 1985.

[ML84] Per Martin-Lf. *Intuitionistic type theory.* Bibliopolis, Napoli, 1984.

[Mog88] Eugenio Moggi. Computational lambda-calculus and monads. pages 14–23. IEEE Computer Society Press, 1988.

[NC00] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information.* Cambridge University Press, October 2000.

[Nor07] Ulf Norell. *Towards a practical programming language based on dependent type theory.* PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

[Pat01] Ross Paterson. A new notation for arrows. In *International Conference on Functional Programming*, pages 229–240. ACM Press, September 2001.

[Pre04] John Preskill. Lectures on quantum computation. http://www.theory.caltech.edu/people/preskill/ph229/, 1997 to 2004.

[RSA77] R. L. Rivest, A. Shamir, and L. M. Adelman. A method for obtaining digital signatures and public-key cryptosystems. Technical Report MIT/LCS/TM-82, 1977.

[Rud07] Roland Rudiger. Quantum Programming Languages: An Introductory Overview. *The Computer Journal*, 50(2):134–150, 2007.

[SA07] Wouter Swierstra and Thorsten Altenkirch. Beauty in the beast: A functional semantics of the awkward squad. In *Haskell '07: Proceedings of the ACM SIGPLAN workshop on Haskell*, 2007.

[Sab03] Amr Sabry. Modelling quantum computing in Haskell. In *Proceedings of the ACM SIGPLAN Workshop on Haskell*. ACM Press, 2003.

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

[Sel07] Peter Selinger. Dagger compact closed categories and completely positive maps: (extended abstract). In *Proceedings of the 3rd International Workshop on Quantum Programming Languages (QPL 2005)*, volume 170 of *Electronic Notes in Theoretical Computer Science*, pages 139–163, 2007.

[Sho94] P Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings, 35th Annual Symposium on Foundations of Computer Science*. CA: IEEE Press, 1994.

[Sim94] David R. Simon. On the power of quantum computation. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*, pages 116–123, Los Alamitos, CA, 1994. Institute of Electrical and Electronic Engineers Computer Society Press.

[Sit08] Ganesh Sittampalam. Restricted monads in Haskell, live journal entry. http://hsenag.livejournal.com/11803.html, March 2008.

[SV09] Peter Selinger and Benot Valiron. Quantum lambda calculus. In Simon Gay and Ian Mackie, editors, *Semantic Techniques in Quantum Computation*. Cambridge University Press, 2009+. to appear.

[Swi08] Wouter Swierstra. *A Functional Specification of Effects*. PhD thesis, University of Nottingham, 2008.

[Val08] Benot Valiron. *Semantics for a Higher Order Functional Programming Language for Quantum Computation*. PhD thesis, University of Ottawa, 2008.

[VAS06] Juliana Kaizer Vizzotto, Thorsten Altenkirch, and Amr Sabry. Structuring quantum effects: Superoperators as arrows. *Mathematical Structures in Computer Science*, 16(3), 2006. Also arXiv:quant-ph/0501151.

[VBE95] V. Vedral, A. Barenco, and A. Ekert. Quantum networks for elementary arithmetic operations, 1995.

[VSB$^+$01] Lieven M. K. Vandersypen, Matthias Steffen, Gregory Breyta, Costantino S. Yannoni, Mark H. Sherwood, and Isaac L. Chuang. Experimental realization of shor's quantum factoring algorithm using nuclear magnetic resonance. *Nature*, 414:883, 2001.

[YY09] Tomoo Yokoyama and Tetsuo Yokoyama. Functoriality in reversible circuits (work in progress). In *Preliminary Proceedings of the Workshop on Reversible Computation*, pages 68–72, 2009.