



CHALMERS
UNIVERSITY OF TECHNOLOGY



UNIVERSITY OF GOTHENBURG



A Functional Quantum Programming Language

Matilda Blomqvist
Nicklas Botö
Beata Burreau
Fabian Forslund
Marcus Jörgensson
Joel Rudsberg

A Functional Quantum Programming Language

Bachelor Thesis

Matilda Blomqvist Nicklas Botö Beata Burreau
Fabian Forslund Marcus Jörgensson
Joel Rudsberg

Supervisor: Robin Adams

Department of Computer Science and Engineering
Chalmers University of Technology
Gothenburg, Sweden

May 14, 2021



CHALMERS
UNIVERSITY OF TECHNOLOGY

A Functional Quantum Programming Language

MATILDA BLOMQVIST

NICKLAS BOTÖ

BEATA BURREAU

FABIAN FORSLUND

MARCUS JÖRGENSSON

JOEL RUDSBERG

© MATILDA BLOMQVIST, NICKLAS BOTÖ, BEATA BURREAU,
FABIAN FORSLUND, MARCUS JÖRGENSSON, JOEL RUDSBERG,
2021

Examiner: Ana Bove

Department of Computer Science and Engineering

Chalmers University of Technology

SE-412 96 Göteborg

Sweden

Telephone +46 (0)31-772 1000

Cover page image ©IBM RESEARCH

Acknowledgments

First and foremost, we would like to thank our supervisor Robin Adams for the invaluable help and insights. We would also like to thank Peter Dybjer for acting as our supervisor for a few weeks with great enthusiasm and valuable ideas for the project.

Abstract

This thesis presents a functional quantum programming language, *funQ*, modeled after a typed quantum lambda calculus [1]. The language *funQ* is implemented as an external domain-specific language in Haskell with a complementing parser, type checker, and interpreter. The type checker is defined with a linear type system – meaning that data cannot be copied – to prevent the physically impossible event of duplication of quantum data. The interpreter uses call-by-value semantics and is connected to a quantum computation library developed to perform the quantum operations with a built-in quantum computer simulator. The language is complemented with a command line tool that can execute *funQ* programs interactively or run *funQ* program files. The project successfully implemented a functional quantum programming language with slight modifications in regards to the modeled language.

Keywords — Functional programming, quantum computation, programming language, linear types.

Sammanfattning

Detta arbete presenterar ett funktionellt kvantprogramsspråk, *funQ*, som modellerats efter en typad kvantlambdakalkyl [1]. Språket *funQ* är implementerat som ett externt domänspecifikt språk i Haskell, med tillhörande tolk, typkontrollerare och interpretator. Typkontrollen är definierad med ett linjärt typsystem – ett system som inte tillåter kopiering av data – för att förhindra den fysiskt omöjliga händelsen av duplicerad kvantdata. Interpretatorn använder 'call-by-value' som evalueringsstrategi och är sammankopplad med ett kvantbibliotek som utvecklats för att utföra kvantberäkningar, med en inbyggd kvantdatorsimulator. Språket har kompletterats med ett program för att köra *funQ* i en kommandotolk, antingen interaktivt i tolken eller genom att köra en *funQ*-programfil. Projektet resulterade i en framgångsrik implementation av ett funktionellt kvantprogrammeringsspråk med några modifieringar av det modellerade språket.

Nyckelord — Funktionell programmering, kvantberäkning, programspråk, linjära typer.

Contents

1	Introduction	1
1.1	The Potential of Quantum Computers	1
1.2	Quantum Programming Languages	1
1.3	Purpose	2
1.4	Delimitations	2
1.5	Outline	2
2	Theory	3
2.1	Fundamentals of Quantum Computation	3
2.1.1	Qubits	3
2.1.2	Quantum Gates	4
2.1.3	Quantum Circuits	5
2.2	The Lambda Calculus	6
2.2.1	Syntax	6
2.2.2	Operations	6
2.3	Programming Language Theory	7
2.3.1	Syntax, Grammars and Parsing	7
2.3.2	Type Checking	8
2.3.3	Interpreting	8
2.4	The Language funQ	9
2.4.1	Syntax	9
2.4.2	Types	9
2.4.3	Typing Rules	10
2.4.4	Additional Relations for Typing	12
2.4.5	Evaluation Rules	13
3	Method	16
3.1	To Implement a Domain-Specific Language	16
3.2	Host Language	16
3.3	Testing	17
4	Implementation of funQ	18
4.1	Syntax and Grammar	18
4.2	Semantic Analysis	20
4.3	Conversion to an Intermediate Abstract Syntax	21
4.4	Type Checking	23
4.4.1	The <i>Check</i> Monad	23
4.4.2	Type Checking Algorithm	23
4.5	Interpreting	26
4.5.1	The <i>Eval</i> Monad	26
4.5.2	Environment	26
4.5.3	Evaluation Algorithm	27
4.6	Library for Quantum Computations	30
4.6.1	The Quantum State	30
4.6.2	Qubits and Gates	30
4.6.3	New and Measure	31

5	Results	32
5.1	The funQ Language	32
5.2	Quantum Algorithms in funQ	33
5.2.1	Coin Flip	33
5.2.2	Quantum Teleportation	34
5.2.3	Deutsch-Jozsa Algorithm	35
5.2.4	Grover's Algorithm	36
5.2.5	Shor's Algorithm	37
5.3	Command Line Tool	38
5.4	Test Results	39
6	Discussion	40
6.1	Ease of Use	40
6.1.1	Simple Syntax	40
6.1.2	Abstraction, Reusability, and Extensibility	40
6.1.3	Value of High-Level Language Features	41
6.2	Implementing a Theoretical Language	41
6.2.1	Effects of Deviations from QLambda	41
6.2.2	Insights from the Implementation	42
6.3	Future Work	42
6.3.1	Improve the Computation Speed	42
6.3.2	Hindley-Milner Type Inference for Linear Types	43
6.3.3	Adding Language Features	43
7	Conclusion	44
	References	45
A	Quantum gates	i
B	Grammar	ii
C	Error types	iv
D	Order-finding algorithm	v

1

Introduction

Quantum computing takes advantage of quantum phenomena to compute and solve problems that may take significantly more time to solve on classical computers. Quantum computers operate on quantum bits (qubits) which in many ways differ from traditional bits. These differences must be taken into consideration when designing a programming language for quantum computers. Many quantum programming languages today either operate on low-level constructs such as quantum circuits or are purely theoretical. Languages with higher abstraction levels and expressiveness exist, but there is a need for more work in this area. Therefore, the goal of this project is to implement a high-level functional programming language for quantum computation.

1.1 The Potential of Quantum Computers

In the second half of the 20th century, the concept of computers based on the principles of quantum mechanics was beginning to form. The idea was to base computation on quantum-specific phenomena such as superposition and entanglement. Some quantum mechanical models of Turing machines were developed in the early 1980's [2]. Around this time, Richard Feynman believed that simulating physical quantum systems would be infeasible on classical computers but could be possible on quantum computers [3].

Later it was found that there is a substantial difference in time complexity between quantum and classical computers for certain types of problems. One such instance is the subset of NP-problems (non-deterministic polynomial time) that require exponential time to solve on classical computers but are easily verifiable [4]. Factorization of integers is a famous example of this type of NP-problem that no known classical deterministic algorithm can solve in polynomial time [5]. However, for quantum computers, Peter Shor developed an algorithm that can factorize integers in polynomial time [6]. He also found a set of other problems where quantum computers outperform classical computers in this way – the class of BQP-problems (bounded-error quantum polynomial time) [7].

In 2019, a group of researchers at Google successfully demonstrated quantum supremacy for a particular task in practice, solving a problem infeasible for a classical computer. They completed a task on a quantum computer in 200 seconds, claimed to need up to 10,000 years on the most powerful modern supercomputer [8]. This concrete example and the set of BQP problems Shor discovered highlight the potential of quantum computers.

1.2 Quantum Programming Languages

Quantum programming languages can be used to write quantum programs by constructing quantum circuits of qubits and quantum gates. Most of today's quantum programming languages, such as QML [9] and Silq [10], include some high-level features on top of the quantum constructs. High-level features, such as loops, if and let statements, more closely resemble natural language than quantum circuits do. High-level languages may therefore make it easier to write quantum programs.

In the paper "A lambda calculus for quantum computation with classical control" [1], Peter Selinger and Benoît Valiron propose a high-level functional quantum programming language (hereafter referred to as QLambda). QLambda combines higher-level language constructs such as if statements with quantum operations. The language is based on the quantum random access machine (QRAM) model, a proposed model of a quantum computer architecture that uses a classical computer connected to a quantum device [11]. The classical computer sends instruction sequences to the quantum device that performs quantum operations and then transfers the result back to the classical computer.

A closely related language is the QML language [9] by Grattage. Like QLambda, it is a functional quantum programming language but with a stricter type system than QLambda. Another difference between QLambda and QML is that QML allows classical and quantum control, whereas QLambda is defined for classical control exclusively. Additionally, QML is implemented in Haskell, whereas QLambda is purely theoretical.

1.3 Purpose

This project aims to implement a user-friendly functional quantum programming language able to run well-known quantum algorithms. The language – funQ – is modeled after the theoretical quantum lambda calculus QLambda and intends to follow the syntax, type- and reduction rules of QLambda as closely as possible.

Implementing a programming language based on a theoretical language involves finding sensible ways to represent the defined types, terms, syntax, and semantics. Running programs in funQ can verify that QLambda works in practice. Hopefully, funQ will provide insights for future work on quantum programming languages.

1.4 Delimitations

Running a quantum program requires either a quantum computer or a simulator. In this project, a simulator is implemented and used to run programs. To connect funQ to an actual quantum computer is not part of this project.

The aim is to implement QLambda as it is, without any major extensions or modifications. However, linear polymorphic type inference with subtyping of Curry-style lambdas is an open problem. Solving this problem is out of the scope of this thesis, and is the reason Church-style lambdas are used as opposed to the Curry-style lambdas in QLambda.

1.5 Outline

This thesis is outlined as follows; first, a presentation of the underlying theory of the project is given in chapter 2. The method used for implementing funQ is described and motivated in chapter 3. Following the method, chapter 4 gives a detailed description of the implementation of funQ, such as the syntax and parsing, type checker, and interpreter. Chapter 5 includes quantum algorithms written in funQ, test results, and sample usage of the command line tool. Finally, the implementation and methodology are discussed in chapter 6, and conclusions are drawn about how well the result of the project meets its purpose in chapter 7.

2

Theory

This chapter covers the fundamental theory behind funQ, which is quantum computation, lambda calculus, and programming language theory.

2.1 Fundamentals of Quantum Computation

In this section, a basic introduction to quantum computation is given. First, some of the properties of qubits and quantum gates are described. These are the building blocks of quantum computers. This is followed by a description of how quantum circuits can be created by combining these two elements. Much of the theory in this section on quantum computation is from a set of lecture notes given by John Preskill in 1998 at the California Institute of Technology [5].

2.1.1 Qubits

The smallest unit of information in classical computers is the bit; the quantum computing counterpart is a quantum bit, or *qubit*. A classical bit has a discrete value 0 or 1, whereas a qubit does not have a discrete value. Instead, it represents 0 or 1 with some probability. More formally, a qubit is in a superposition of the states 0 and 1 simultaneously.

The qubit describes a state in the simplest possible quantum system. Two examples of such systems in real quantum computers are the vertical and horizontal polarizations of light and the up and down spin of electrons. For each case, the principle is the same: there are two levels to the system between which the quantum state can be measured.

The state of a qubit can be mathematically represented by a vector in a two-dimensional Hilbert space, denoted \mathbb{C}^2 , where the superscript denotes the number of dimensions. The Hilbert space most importantly provides the definition of orthogonality between vectors. The orthonormal basis for a two-dimensional space may be denoted by the set $\{|0\rangle, |1\rangle\}$. Any general qubit, $|\psi\rangle$, describes a state in this space and is written as

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

where α and β are complex numbers and denote the qubit's amplitudes, with the constraint $|\alpha|^2 + |\beta|^2 = 1$. The square of the absolute value of α and β correspond to the probability of a measurement outcome. As can be seen in the constraint, these probabilities add up to one.

The two operations that can be performed on qubits are gate transformations and measurement. Applying a gate to a qubit transforms the qubits' state, changing the probabilities of the qubit to collapse to either $|0\rangle$ or $|1\rangle$. The measurement operation is done by projecting the mathematical representation of a qubit onto one of the basis vectors. The qubit, that is, the vector that describes the quantum state, then collapses to its projection onto either of the basis vectors depending on its associated probability. Measuring the qubit $|\psi\rangle$ of equation 1, the result will be $|0\rangle$ with probability $|\alpha|^2$ and $|1\rangle$ with probability $|\beta|^2$. When measuring the qubit, the result is a discrete binary value that is mapped to a bit in a classical computer.

The measurement operation physically entails checking the value of some property of a quantum system. Take the direction of a particle's spin as an example. The quantum state is in a superposition of the spin-up and spin-down states before measurement, each with some corresponding amplitude. Upon measurement, the state collapses to either up or down-spin with the probability according to its amplitudes.

An important aspect of qubits is that they cannot be duplicated, giving rise to the *no-cloning theorem*. This theorem states that there is no quantum gate acting on two arbitrary qubits such that the state of the first qubit is copied onto the other, which in essence means that qubits cannot be duplicated. For instance, the operation $|\psi\rangle |0\rangle \mapsto |\psi\rangle |\psi\rangle$ is not possible [12].

A multiple-qubit system is represented by the tensor product between qubits. For example, a two-qubit system is represented as the tensor product of the vector representations of the two individual qubits. If they are both in \mathbb{C}^2 , the result is a vector in \mathbb{C}^4 . An instance of this is the tensor product between the zero and one vector, calculated as in equation (2).

$$|01\rangle = |0\rangle \otimes |1\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \otimes \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \cdot 0 \\ 1 \cdot 1 \\ 0 \cdot 0 \\ 0 \cdot 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \quad (2)$$

In general, an n -qubit system will be represented as a vector of the form $\otimes_{i=0}^{n-1} \mathbb{C}^2 = \mathbb{C}^{2^n}$.

Another important aspect of qubits is entanglement. Consider the two-qubit state given by $\psi_{AB} = [0, 1, 0, 0]^T$ in equation (2). As can be seen from the example above, it is simple to separate (factor) the two qubits into their corresponding one-qubit states $[1, 0]^T \otimes [0, 1]^T$. They are uncorrelated as each qubit does not depend on the other. In other words, they are unentangled. Qubits in an entangled state cannot be factored. They are correlated; thus, if one qubit of an entangled pair is affected somehow, this also affects the other. An example of a simple entangled state is one of the Bell states, which is $|\Phi^+\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$. Here the state is in a superposition of $|00\rangle$ and $|11\rangle$, which tells us that both the first and second qubits are either 0 or 1. Measuring either of the qubits will give information about the state of the other qubit. If the first qubit is measured and collapses to $|0\rangle$, then the second qubit also collapses to $|0\rangle$, and vice versa. The two qubits are thus connected in *some* way and cannot be factored.

2.1.2 Quantum Gates

Quantum gates act on some number of qubits and can change the state of the qubits. Quantum gates are analog to logical gates in classical computers, such as AND, XOR, and NOR. Quantum gates have the additional property of unitarity, meaning they are reversible and preserve the norm of the quantum state to one. For a quantum gate U and state $|\psi\rangle$, unitarity gives that $\|U|\psi\rangle\| = \||\psi\rangle\|$ where $\|v\|$ denotes the norm of a vector v . Reversibility means that each quantum gate can be reversed by an inverse gate. Some gates, such as the Hadamard gate, are their own inverse. This means that applying the Hadamard gate twice will output the same state as before. In contrast, classical gates such as AND are not reversible. If the output of an AND gate is zero, it is impossible to know which of the input bits was zero.

Unitary quantum gates are mathematically represented as matrices and act on qubits through matrix multiplication. Two examples are given to make the idea of quantum gates clear. First, consider the matrix representations of two common quantum gates displayed in figure 2.1.

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

(a) Pauli-X/NOT matrix.

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

(b) Hadamard matrix.

Figure 2.1: Matrix representations of two quantum gates acting on one respective two qubits. The matrices' sizes correspond to the number of qubits they act upon; a gate acting on a quantum state of size n has a matrix size of $2^n \times 2^n$.

The Pauli-X gate (also called NOT) corresponds to a state inversion operation. In a similar manner to the familiar NOT gate of classical computers, it acts on a single qubit, inverts its state, and maps $[\alpha, \beta]^T$ to $[\beta, \alpha]^T$. The Hadamard matrix represents the Hadamard gate operating on two qubits. Applying the Hadamard gate to the two qubits changes their state so that upon measurement of the qubits, the result is one of the four possible combinations of two bits, i.e., 00, 01, 10, 11. The probabilities of the outcomes correspond to the amplitudes of qubits before measurement. If the quantum state before application of the Hadamard gate is $\psi = [0, 0, 1, 0]^T$, applying the gate will change the state to $H^{\otimes 2}\psi = [\frac{1}{2}, \frac{1}{2}, -\frac{1}{2}, -\frac{1}{2}]^T$. Measuring the state gives $\psi_{00} = |00\rangle$, $\psi_{01} = |01\rangle$, $\psi_{10} = |10\rangle$ or $\psi_{11} = |11\rangle$, each with probability 25%.

An essential type of quantum gate is the controlled gates. These take in two or more qubits where some number of qubits are called control qubits, which control the operation of the gate. The state of the control qubits decides whether an operation should be performed on the remaining qubits. For example, the two-qubit controlled NOT gate (CNOT) performs a NOT operation on the second qubit if the state of the control qubit is $|1\rangle$. It does nothing if the state of the control qubit is $|0\rangle$.

2.1.3 Quantum Circuits

Quantum circuits are constructed using a sequence of quantum gates to perform computation and are one way of representing quantum programs. Two examples of quantum circuits are given in figure 2.2.

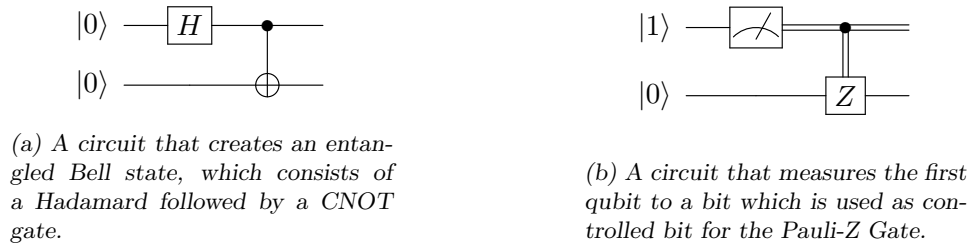


Figure 2.2: Two circuits that include the essentials of quantum circuits; that is, inputs, gate operations, measurement, and outputs.

Circuit diagrams are read left to right. The leftmost part is the input to the circuit. The above examples either have the zero-qubit $|0\rangle$ or the one-qubit $|1\rangle$ as input. The right end of the circuit denotes the output. The horizontal lines are called the wires. These connect the operations performed on a certain qubit. The boxes on the wires represent gates, where H stands for the Hadamard gate and Z for the Pauli-Z gate. The circle containing a cross together with the little black dot represents the CNOT gate. Controlled gates, such as CNOT, can be distinguished by the vertical line connecting the gate to the control bit, represented as a black dot. As can be seen, controlled gates can be controlled by either a qubit (figure 2.2a) or a bit (figure 2.2b). If the control bit is 1 or the qubit is in state $|1\rangle$, the controlled operation is performed. Finally, the box in figure 2.2b represents a measurement. Since measurement outputs classical bits, the wire after the measurement is represented by two parallel lines to distinguish it from qubit wires.

2.2 The Lambda Calculus

The lambda calculus is a language for expressing computation, which has the same computational power as Turing Machines. The lambda calculus has very few constructs, reflected in its syntax. It also has a few simple operations.

2.2.1 Syntax

The lambda calculus consists of three constructs: a variable, lambda abstraction, and application. The syntax of the lambda calculus is shown in figure 2.3.

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

Figure 2.3: The syntax of the lambda calculus. The terms represent a variable, lambda abstraction, and application.

The first term, x is a variable. The second term is an anonymous function called a lambda abstraction, where x is the argument and M the function body. Lambda abstractions introduce and bind a variable x in the term M . In this way, the term $\lambda x.M$ is similar to the mathematical expression $f(x) = M$, where x may or may not appear in M . If a lambda abstraction has introduced a variable in a term, the variable is said to be *bound*. Otherwise, it is called *free*. For example, the variable x is bound in $\lambda x.x$, and free in $\lambda y.x$. The set of free variables in a term M is denoted by $FV(M)$. The last term describes an application, that is, applying a function to its argument. Analogous to mathematical notation, the term $(\lambda x.M)N$ is similar to $f(N)$, where $f(x) = M$.

An extension of the lambda calculus is the typed lambda calculus, where all terms have a type. Lambda abstractions where the argument is explicitly typed, such as $(\lambda x : \tau.M)$, are called Church-style lambdas, opposed to Curry-style lambdas $(\lambda x.M)$.

2.2.2 Operations

Lambda calculus includes three operations known as β -reduction, α -conversion, and η -reduction. The process of replacing the lambda-bound variable in the body of a lambda abstraction with the argument is β -reduction. The term $(\lambda x.M)N$ is β -reduced to $M[x := N]$, where N replaces all free variables x in M . When performing β -reductions, α -conversion can be used to avoid name collisions. That is, renaming the bound variables in a lambda abstraction. For instance, consider the term $\lambda x.(\lambda y.\lambda x.y)x$ that is β -reduced to $\lambda x.(\lambda x.x)$, where the variable x incorrectly gets bound to the inner lambda, when it should be referring to the outer lambda. This situation is avoided using α -conversion. Performing α -conversion on the term $\lambda x.(\lambda y.\lambda x.y)x$ in the example above, renaming the variable x in the inner lambda to z , would lead to the term $\lambda x.(\lambda y.\lambda z.y)x$ which evaluates to $\lambda x.\lambda z.x$ (known as the K-combinator). The third operation, η -reduction, states that $\lambda x.Mx$ can be reduced to M if x is not a free variable in M .

The need for α -conversion can be avoided by using De Bruijn indexing. A De Bruijn index corresponds to how many abstractions away the variable's binding abstraction is [13]. For instance, the constant function is $\lambda(\lambda 1)$ where 1 refers to the outer lambda, as the lambda is one abstraction level away from the variable. The example term from above would be converted to $\lambda(\lambda \lambda 1)0$, which evaluates to $\lambda(\lambda 1)$ as intended. De Bruijn indices simplify both evaluation and type checking since no α -conversions is needed.

2.3 Programming Language Theory

The funQ language is built on fundamental concepts from programming language theory, including syntax and parsing, type checking and type inference, and evaluation of programs.

2.3.1 Syntax, Grammars and Parsing

Programming languages are often defined by a formal grammar. The grammar describes the syntax of a language and is a collection of production rules. Production rules describe how expressions can be formed, similar to how sentences can be formed in natural languages. It is also possible to define how expressions can be combined into larger constructs like programs in the grammar [14]. Often when defining a new programming language, the grammar is expressed in Backus-Naur form (BNF) [14] [15]. For instance, consider figure 2.4, a small grammar written in BNF notation. This grammar can be used to recursively derive expressions of integer addition or simply integers or digits.

$$\begin{aligned}\langle Exp \rangle &::= \langle Exp \rangle '+' \langle Exp \rangle \mid \langle Integer \rangle \\ \langle Integer \rangle &::= \langle digit \rangle \mid \langle digit \rangle \langle Integer \rangle \\ \langle Digit \rangle &::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9\end{aligned}$$

Figure 2.4: A small grammar written in Backus-Naur form. Derivable expressions from this grammar are expressions of integer additions, digits, and integers.

Between the angled brackets is the syntactic category, and after "::<=" comes the definition of this category. To exemplify, the syntactic categories of the grammar in figure 2.4 are Exp, Integer, and Digit. "|" can be read as "or". For example, the production rule for digits in figure 2.4 semantically means "a digit is defined to be 0 or 1 or 2 or ... or 9". A concrete example of a derivable expression from this grammar would be "2 + 2 + 5".

With the syntax of a language described by a grammar, it is possible to parse a program to generate an abstract syntax tree (AST) representation of the program. A program is parsed in two steps. First, the lexer analyses the program string, character by character, and splits it into tokens, or "words". Then, the parser analyses the tokens by mapping each token with the grammar, building up the AST. An AST is a tree representation of a program where nodes and leaves denote syntactic constructs defined in the grammar. For instance, an AST for the language described in figure 2.4 would represent integers and the addition of expressions as nodes and single digits as leaves. The derivable expression "2 + 2 + 5" could be represented by the AST in figure 2.5, depending on the associativity of the "+" operator.

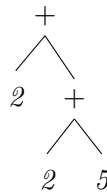


Figure 2.5: An abstract syntax tree representation of the expression "2 + 2 + 5" in the language described in figure 2.4.

2.3.2 Type Checking

The AST can be checked and analyzed to make sure that the program is valid. The AST is often run through a type checker, where the types of the program are statically checked to be valid according to the type system. Common type checking procedures need two kinds of operations: type inference and type checking. Type inference is performed by finding the type T of a given expression e [14]. Type checking is performed by checking that a given expression e has a given type T .

The type system is commonly described by typing rules. Typing rules consists of judgments, and a basic judgment has the form $\Gamma \vdash x : \tau$ which reads as "in context Γ , x has type τ ". Γ denotes the typing context, which is a map from variables to their types. As an example, the typing rule for the equal operator in figure 2.6 can be described as "in context Γ , x has type Int , and y has type Int , then $x == y$ has type Bool ".

$$\frac{\Gamma \vdash x : \text{Int} \quad \Gamma \vdash y : \text{Int}}{\Gamma \vdash x == y : \text{Bool}}$$

Figure 2.6: A typing rule for the expression $x == y$.

2.3.3 Interpreting

After type checking, the AST is evaluated by the interpreter. The interpreter executes the program; for example, the execution includes printing out print statements, evaluating arithmetic expressions, etc. Similar to the typing rules for typing, the *operational semantics* is a rule system for interpreters [14]. A basic judgment has the form $\gamma \vdash e \Downarrow \langle v, \gamma' \rangle$ which reads as "in environment γ , the expression e evaluates to value v and the new environment γ' " and can be used to build up more complex evaluation rules. The environment γ is a map from variables to their corresponding values. The environment is updated by adding a new binding from variable x to value v by writing $\gamma(x := v)$. If the binding already exists, the value is updated. The updated environment is denoted γ' . An evaluation rule states that a judgment holds, potentially under some condition, expressed as other judgments. In the example shown in 2.7, the judgment for an if then else statement holds under the condition of the two judgments above the horizontal line.

$$\frac{\gamma \vdash x \Downarrow \langle 1, \gamma' \rangle \quad \gamma' \vdash y \Downarrow \langle v, \gamma'' \rangle}{\gamma \vdash \text{if } x \text{ then } y \text{ else } z \Downarrow \langle v, \gamma'' \rangle}$$

Figure 2.7: An evaluation rule for an if then else statement. If the condition is evaluated to true, the whole expression is evaluated to the value generated from the 'then' branch. The environment is possibly updated, indicated by the γ'' . The 'else' branch is left unevaluated.

2.4 The Language funQ

Modeled after the theoretical quantum lambda calculus QLambda, funQ is an implemented quantum language defined with classical control, call-by-value operational semantics, and a linear type system. This section covers the syntax, typing rules, and reduction rules of funQ.

2.4.1 Syntax

The syntax of funQ is defined with regards to a lambda term, which is named *Term* and from which expressions and programs can be derived. The syntax is described by the production rules in figure 2.8.

$$\begin{aligned}
 \text{Term } M, N, P ::= & x \mid (MN) \mid \lambda x : \sigma . M \mid \langle M, N \rangle \mid \text{let } \langle x, y \rangle = M \text{ in } N \\
 & \mid \text{if } P \text{ then } M \text{ else } N \mid 0 \mid 1 \mid \text{new} \mid \text{meas} \mid * \mid M \$ N \mid U \\
 \\
 \text{Gate } U ::= & H \mid X \mid Y \mid Z \mid I \mid S \mid T \mid \text{CNOT} \mid \text{TOF} \mid \text{SWP} \mid \text{FRDK} \\
 & \mid \text{QFT}n \mid \text{QFT}In \mid \text{CR}n \mid \text{CRI}n \mid \text{CCR}n \mid \text{CCRI}n
 \end{aligned}$$

Figure 2.8: The syntax of funQ. M , N and P are terms. U is any gate. To specialize a gate, integer n is used.

The first production rule of *Term* is x , which is a string representing a variable. The second and third rule – function application and lambda abstraction – has the same meaning as in the regular lambda calculus described previously in section 2.2, except that the lambda is Church-style. The next rule defines the tuple, or product term, containing two terms. The following term, *let*, is used to split product terms into their individual components. The *if* term represents regular conditional branching. Terms 0 and 1 represents bit values, while *new* and *meas* are two predefined functions. The function *new* creates a new qubit while *meas* represents the measurement operation. The star (*) is the unit term which is the only value of the unit type \top . The unit term is similar to the null or none values in other programming languages. The dollar (\$) is the precedence operator that binds an application with right-associativity and is equivalent to the \$ operator in Haskell.

The syntactic category denoted by Gate U is the set of all available unitary gates. Some gates take an argument n that changes the behavior of the gate. For instance, the gates $\text{CR}n$ and $\text{CRI}n$ take an angle that specifies the rotation to be performed. The full names of each gate are given in appendix A.

This definition of terms is mainly equivalent to the one given by Selinger and Valiron [1]. The differences are the predefined set of gates, the \$-operator, and that Curry-style lambdas have been replaced with Church-style lambdas.

2.4.2 Types

Linear type theory is based on the linear logic proposed by Girard [16] and restricts the use of a value. Fundamentally, a value with a linear type must be used exactly once – no values can be duplicated, or discarded [17]. Linear types are therefore especially well-suited for quantum programming languages due to the no-cloning theorem of qubits.

The type system of funQ is an affine type system as defined by Selinger and Valiron in [1]. Affine types can be seen as a version of linear types with the difference that discarding values is allowed, and an

affine value can be used *at most* once. Throughout the rest of this thesis, the use of the word linear will practically mean affine. The types of funQ are defined as in figure 2.9 below.

$$\text{Type } \sigma, \tau ::= \text{bit} \mid \text{qbit} \mid !\sigma \mid \sigma \multimap \tau \mid \top \mid \sigma \otimes \tau.$$

Figure 2.9: The definition of types in funQ. The type of bit, qubit, and \top are the base types while σ and τ is any type.

The base types are bit, qbit, and the unit type \top . The function type is represented by $\sigma \multimap \tau$ and the product type by $\sigma \otimes \tau$, both right-associative. The n-fold product type $\sigma \otimes \sigma \otimes \dots \otimes \sigma$ is also denoted σ^n , for example $\text{bit}^2 = \text{bit} \otimes \text{bit}$. Furthermore, a type σ is assumed to be non-duplicable, that is, a *linear* type. Prepending a $!$, to a type σ , denotes the duplicable type $!\sigma$. A type with n repeated $!$ can be denoted $!^n\sigma$ or $! \dots !\sigma$. $n \geq 1$ gives a duplicable type and $n = 0$ gives a linear type. Repeated $!$ in a type is in practice equivalent to a single $!$, since in both cases the type is duplicable.

2.4.3 Typing Rules

The type system for funQ is defined by the typing rules. Before showing the typing rules, the subtype relation and specific terminology for type environments need to be defined. The subtyping and typing rules are based on the definitions in [1] but are presented in a slightly different way.

Subtypes are used in the type system to make linear types more flexible. For example, a lambda expecting a linear bit should also be callable with a duplicable bit since the duplicable bit can be downgraded to a linear bit. The rules for subtyping are shown in figure 2.10. In comparison with QLambda, the additional rule $(! \otimes)$ is added, which is needed in the implementation.

$$\begin{array}{c} \frac{}{\sigma <: \sigma} \text{ (AX)} \quad \frac{\sigma <: \tau}{!\sigma <: \tau} \text{ (D)} \quad \frac{!\sigma <: \tau}{!\sigma <: !\tau} \text{ (!)} \quad \frac{\sigma_1 <: \tau_1 \quad \sigma_2 <: \tau_2}{\sigma_1 \otimes \sigma_2 <: \tau_1 \otimes \tau_2} \text{ (}\otimes\text{)} \\[10pt] \frac{!\sigma_1 <: \tau_1 \quad !\sigma_2 <: \tau_2}{!(\sigma_1 \otimes \sigma_2) <: \tau_1 \otimes \tau_2} \text{ (!}\otimes\text{)} \quad \frac{\tau_1 <: \sigma_1 \quad \sigma_2 <: \tau_2}{\sigma_1 \multimap \sigma_2 <: \tau_1 \multimap \tau_2} \text{ (}\multimap\text{)} \end{array}$$

Figure 2.10: The rules for subtyping in funQ. The relation $\sigma <: \tau$ reads as " σ is a subtype of τ " or " τ is a supertype of σ ".

The typing rules make use of a typing environment that maps all free and bound variables to their types, denoted by Γ . The domain of the typing environment is denoted $|\Gamma|$, that is, the name of all variables $x \in \Gamma$. In addition, $!\Gamma$ denotes the environment of all non-linear variables and $\mathfrak{l}\Gamma$ denotes the linear environment, such that $!\Gamma \cup \mathfrak{l}\Gamma = \Gamma$. Finally, $|U|$ denotes the input size of a unitary gate U . If the gate U operates on two qubits, then $|U| = 2$. The typing rules are displayed in figure 2.11.

The typing rules labeled with *ax* cover the basic terms, the built-in functions, and the unitary gates. One thing to note is that they are all given the most general type. For example, a 0 or 1 is typed as $!\text{bit}$ since it is possible to restrict it to a linear bit if required while the opposite violates the subtyping rules. The term *new* is inferred to the type $!(\text{bit} \multimap \text{qbit})$ by (ax_n) , where the outer $!$ means that the whole function can be reused. The type of a unitary gate is a function from n qubits to n qubits. The type depends on the size of the gate, that is, how many qubits the gate operates on.

Typing a variable x follows the (*var*) rule. If a variable exists and has a type σ in the typing environment, the variable's type could be any supertype of σ .

$$\begin{array}{c}
\frac{}{\Gamma \vdash * : !\top} (ax_*) \quad \frac{}{\Gamma \vdash 0 : !\text{bit}} (ax_0) \quad \frac{}{\Gamma \vdash 1 : !\text{bit}} (ax_1) \\
\\
\frac{}{\Gamma \vdash \text{new} : !(\text{bit} \multimap \text{qbit})} (ax_n) \quad \frac{}{\Gamma \vdash \text{meas} : !(\text{qbit} \multimap !\text{bit})} (ax_m) \\
\\
\frac{x : \sigma \in \Gamma \quad \sigma <: \tau}{\Gamma \vdash x : \tau} (var) \quad \frac{|U| = n}{\Gamma \vdash U : !(\text{qbit}^n \multimap \text{qbit}^n)} (ax_U) \\
\\
\frac{x : \sigma, \Gamma \vdash M : \tau}{\Gamma \vdash (\lambda x : \sigma. M) : \sigma \multimap \tau} (\lambda_1) \\
\\
\frac{\text{If } FV(M) \cap |\mathfrak{i}\Gamma| = \emptyset \quad \Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash (\lambda x : \sigma. M) : !(\sigma \multimap \tau)} (\lambda_2) \\
\\
\frac{!\Gamma, \mathfrak{i}\Gamma_1 \vdash M : \sigma \multimap \tau \quad !\Gamma, \mathfrak{i}\Gamma_2 \vdash N : \sigma}{!\Gamma, \mathfrak{i}\Gamma_1, \mathfrak{i}\Gamma_2 \vdash MN : \tau} (app) \\
\\
\frac{!\Gamma, \mathfrak{i}\Gamma_1 \vdash P : \text{bit} \quad !\Gamma, \mathfrak{i}\Gamma_2 \vdash M : \sigma \quad !\Gamma, \mathfrak{i}\Gamma_2 \vdash N : \sigma}{!\Gamma, \mathfrak{i}\Gamma_1, \mathfrak{i}\Gamma_2 \vdash \text{if } P \text{ then } M \text{ else } N : \sigma} (if) \\
\\
\frac{!\Gamma, \mathfrak{i}\Gamma_1 \vdash M : !^n \sigma \quad !\Gamma, \mathfrak{i}\Gamma_2 \vdash N : !^n \tau}{!\Gamma, \mathfrak{i}\Gamma_1, \mathfrak{i}\Gamma_2 \vdash \langle M, N \rangle : !^n(\sigma \otimes \tau)} (\otimes) \\
\\
\frac{!\Gamma, \mathfrak{i}\Gamma_1 \vdash M : !^n(\sigma \otimes \tau) \quad !\Gamma, \mathfrak{i}\Gamma_2, x : !^n \sigma, y : !^n \tau \vdash N : \psi}{!\Gamma, \mathfrak{i}\Gamma_1, \mathfrak{i}\Gamma_2 \vdash \text{let } \langle x, y \rangle = M \text{ in } N : \psi} (let)
\end{array}$$

Figure 2.11: The typing rules for *funQ* which are similar to the typing rules given in [1].

There are two rules for lambda abstractions to capture different cases. The first case, (λ_1) , is the trivial case where the function type is linear. In the other case, (λ_2) , the function type is duplicable, assuming the body does not use any free variables that exist in the linear environment.

The *(app)* rule says that the argument must match the function parameter in an application. The final type of an application is the type of the function body.

The *(if)* rule treats the linear environments a bit differently than the other rules. The then and else terms have the same linear environment, resulting in that linear variables could be used in both branches. However, linear variables cannot be used in both the condition and one of the then/else terms. The type of the condition must be a bit, and the resulting type is a common type of the branches.

The product rule (\otimes) explains how $!$ is treated for product types. All common $!$ between its components are moved outside the product.

Finally, the *(let)* rule implies that all $!$ outside a product is appended to its constituent components when deriving the type of the N term.

2.4.4 Additional Relations for Typing

Three additional functions need to be defined to implement the typing rules. Two functions serve the goal of finding the least common supertype of two types. The third function checks how many times a variable is used.

When inferring the type of an if expression, functions are used to find the least common supertype, called supremum, and the greatest common subtype, called infimum. The then and else branches of an if expression may have different types and still be typeable, as long as they have a common supertype. Hence, their least common supertype is the inferred type of an if expression. Supremum is denoted by \vee and infimum by \wedge , and the definition is given in figure 2.12.

$$\begin{array}{ll}
\sigma \wedge \sigma = \sigma & \sigma \vee \sigma = \sigma \\
!\sigma \wedge !\tau = !(\sigma \wedge \tau) & !\sigma \vee !\tau = !(\sigma \vee \tau) \\
!\sigma \wedge \tau = !(\sigma \wedge \tau) & !\sigma \vee \tau = \sigma \vee \tau \\
\sigma \wedge !\tau = !(\sigma \wedge \tau) & \sigma \vee !\tau = \sigma \vee \tau \\
(\sigma_1 \otimes \tau_1) \wedge (\sigma_2 \otimes \tau_2) = (\sigma_1 \wedge \sigma_2) \otimes (\tau_1 \wedge \tau_2) & (\sigma_1 \otimes \tau_1) \vee (\sigma_2 \otimes \tau_2) = (\sigma_1 \vee \sigma_2) \otimes (\tau_1 \vee \tau_2) \\
!(\sigma_1 \otimes \tau_1) \wedge (\sigma_2 \otimes \tau_2) = (!\sigma_1 \wedge \sigma_2) \otimes (!\tau_1 \wedge \tau_2) & !(\sigma_1 \otimes \tau_1) \vee (\sigma_2 \otimes \tau_2) = (!\sigma_1 \vee \sigma_2) \otimes (!\tau_1 \vee \tau_2) \\
(\sigma_1 \otimes \tau_1) \wedge !(\sigma_2 \otimes \tau_2) = (\sigma_1 \wedge !\sigma_2) \otimes (\tau_1 \wedge !\tau_2) & (\sigma_1 \otimes \tau_1) \vee !(\sigma_2 \otimes \tau_2) = (\sigma_1 \vee !\sigma_2) \otimes (\tau_1 \vee !\tau_2) \\
(\sigma_1 \multimap \tau_1) \wedge (\sigma_2 \multimap \tau_2) = (\sigma_1 \vee \sigma_2) \multimap (\tau_1 \wedge \tau_2) & (\sigma_1 \multimap \tau_1) \vee (\sigma_2 \multimap \tau_2) = (\sigma_1 \wedge \sigma_2) \multimap (\tau_1 \vee \tau_2)
\end{array}$$

Figure 2.12: The definition of \wedge and \vee for any type σ and τ .

To ensure linearity, counting how many times a variable is used in a term is needed every time a new variable is bound. For example, the lambda argument x cannot have a linear type if it is used more than once in the body. $\Omega_x(M)$ is the number of occurrences of the free variable x in a term M and is defined in figure 2.13.

$$\begin{array}{l}
\Omega_x(x) = 1 \\
\Omega_x(y) = 0 \\
\Omega_x(\lambda y.M) = \Omega_x(M) \quad \text{where } y \neq x \\
\Omega_x(MN) = \Omega_x(M) + \Omega_x(N) \\
\Omega_x(\text{if } P \text{ then } M \text{ else } N) = \Omega_x(P) + \max(\Omega_x(M), \Omega_x(N)) \\
\Omega_x(\langle M, N \rangle) = \Omega_x(M) + \Omega_x(N) \\
\Omega_x(\text{let } \langle a, b \rangle = M \text{ in } N) = \Omega_x(M) + \Omega_x(N) \quad \text{where } a \neq x \text{ and } b \neq x
\end{array}$$

Figure 2.13: The definition of the function $\Omega_x(M)$ which counts the number of occurrences of the free variable x in the term M .

2.4.5 Evaluation Rules

The evaluation of a funQ term is defined by the evaluation rules. The notation and evaluation rules defined in this section are specific for funQ and thus original work. However, with funQ being modeled after QLambda, inspiration has been taken from the probabilistic call-by-value operational semantics defined in [1]. Before introducing the rules, the notation and overall structure of the evaluation is explained.

Evaluation is performed by evaluating a funQ term t to a value term v . The set of value terms is defined in figure 2.14.

$$\text{Value } v, w ::= q \mid \lambda x.M \mid 0 \mid 1 \mid \text{meas} \mid \text{new} \mid U \mid * \mid \langle v, w \rangle$$

Figure 2.14: The value term in funQ. A value term may be a qubit, lambda abstraction, bit of value 0 or 1, new or measurement operation, unitary gate, unit, or tuple of values.

A program state is continuously updated throughout the evaluation. The state consists of the quantum state Q and the environment γ and is denoted Q, γ . The contents of γ are three maps:

- values of bound variables, $x_1 := v_1, \dots, x_m := v_m$
- function terms, $f_1 := t_1, \dots, f_n := t_n$
- values of evaluated function terms, $f_1 := v_1, \dots, f_k := v_k$.

Assignment and existence checks in γ are made in the appropriate map depending on the letters used. For instance, $\gamma(f := v)$ assigns a value v to function f , and $f := t \in \gamma$ checks function f and its associated term t exists.

The other part of the state, Q , is the quantum state. It is a normalized vector, $\otimes_{i=0}^{n-1} \mathbb{C}^2$ for some $n > 0$, that describes the physical state of all qubits. A single qubit is referred to as q , which is an integer pointer that links to the qubit in Q .

The evaluation of a term is probabilistic, which means that given the environment Q, γ , a term t evaluates to a particular value v with the probability p . This is described by the judgment $Q, \gamma \vdash t \Downarrow_p \langle v, Q, \gamma \rangle$. If the probability $p = 1$, the evaluation always yields the same result.

$$\begin{array}{c}
\frac{}{Q, \gamma \vdash \text{bit } 0 \Downarrow_1 \langle 0, Q, \gamma \rangle}^{(b_0)} \quad \frac{}{Q, \gamma \vdash \text{bit } 1 \Downarrow_1 \langle 1, Q, \gamma \rangle}^{(b_1)} \\
\\
\frac{}{Q, \gamma \vdash \text{unit } * \Downarrow_1 \langle *, Q, \gamma \rangle}^{(unit)} \\
\\
\frac{}{Q, \gamma \vdash \text{new } 0 \Downarrow_1 \langle q, Q \otimes |0\rangle, \gamma \rangle}^{(new_0)} \\
\\
\frac{}{Q, \gamma \vdash \text{new } 1 \Downarrow_1 \langle q, Q \otimes |1\rangle, \gamma \rangle}^{(new_1)} \\
\\
\frac{}{\alpha |Q_0\rangle + \beta |Q_1\rangle, \gamma \vdash \text{meas } q \Downarrow_{|\alpha|^2} \langle 0, |Q_0\rangle, \gamma \rangle}^{(meas_0)} \\
\\
\frac{}{\alpha |Q_0\rangle + \beta |Q_1\rangle, \gamma \vdash \text{meas } q \Downarrow_{|\beta|^2} \langle 1, |Q_1\rangle, \gamma \rangle}^{(meas_1)} \\
\\
\frac{f := t \in \gamma \quad f := v \notin \gamma \quad Q, \gamma \vdash t \Downarrow_p \langle v, Q', \gamma \rangle}{Q, \gamma \vdash f \Downarrow_p \langle v, Q', \gamma(f := v) \rangle}^{(f_1)} \quad \frac{f := t \in \gamma \quad f := v \in \gamma}{Q, \gamma \vdash f \Downarrow_1 \langle v, Q, \gamma \rangle}^{(f_2)} \\
\\
\frac{}{Q, \gamma \vdash U \langle q_1, \dots, q_n \rangle \Downarrow_1 \langle \langle q_1, \dots, q_n \rangle, Q', \gamma \rangle}^{(U_n)} \\
\\
\frac{Q, \gamma \vdash t \Downarrow_{p_1} \langle v, Q', \gamma' \rangle \quad Q', \gamma'(x := v) \vdash M \Downarrow_{p_2} \langle w, Q'', \gamma'' \rangle}{Q, \gamma \vdash (\lambda x. M) t \Downarrow_{p_1 p_2} \langle w, Q'', \gamma'' \rangle}^{(app)} \\
\\
\frac{Q, \gamma \vdash t_1 \Downarrow_{p_1} \langle v_1, Q', \gamma' \rangle \quad Q', \gamma' \vdash t_2 \Downarrow_{p_2} \langle v_2, Q'', \gamma'' \rangle}{Q, \gamma \vdash \langle t_1, t_2 \rangle \Downarrow_{p_1 p_2} \langle \langle v_1, v_2 \rangle, Q'', \gamma'' \rangle}^{(tup)} \\
\\
\frac{Q, \gamma \vdash t_1 \Downarrow_{p_1} \langle \langle v_1, v_2 \rangle, Q', \gamma' \rangle \quad Q', \gamma'(x_1 := v_1, x_2 := v_2) \vdash t_2 \Downarrow_{p_2} \langle v, Q'', \gamma'' \rangle}{Q, \gamma \vdash \text{let } \langle x_1, x_2 \rangle = t_1 \text{ in } t_2 \Downarrow_{p_1 p_2} \langle v, Q'', \gamma'' \rangle}^{(let)} \\
\\
\frac{Q, \gamma \vdash t \Downarrow_{p_1} \langle 1, Q', \gamma' \rangle \quad Q', \gamma' \vdash t_1 \Downarrow_{p_2} \langle v, Q'', \gamma'' \rangle}{Q, \gamma \vdash \text{if } t \text{ then } t_1 \text{ else } t_2 \Downarrow_{p_1 p_2} \langle v, Q'', \gamma'' \rangle}^{(if_1)} \\
\\
\frac{Q, \gamma \vdash t \Downarrow_{p_1} \langle 0, Q', \gamma' \rangle \quad Q', \gamma' \vdash t_2 \Downarrow_{p_2} \langle v, Q'', \gamma'' \rangle}{\gamma \vdash \text{if } t \text{ then } t_1 \text{ else } t_2 \Downarrow_{p_1 p_2} \langle v, Q'', \gamma'' \rangle}^{(if_0)}
\end{array}$$

Figure 2.15: The evaluation rules for funQ.

The evaluation rules (b_0) and (b_1) in figure 2.15 show that a bit term evaluates to the corresponding value term. The evaluation is deterministic; that is, bit 0 will always evaluate to 0, and bit 1 to 1, leaving the state unchanged. Similarly, a unit term evaluates to the corresponding value term as shown in rule $(unit)$.

The evaluation of the new operation is deterministic and represented in rules (new_0) and (new_1) . Evaluation of new 0 yields an updated quantum state Q by appending a qubit $|0\rangle$ to it and a pointer q to the qubit.

The measurement operation gives rise to the probabilistic elements in the evaluation scheme. The measurement of a qubit evaluates to 0 with a probability of $|\alpha|^2$ and 1 with a probability of $|\beta|^2$, which is represented in the rules $(meas_0)$ and $(meas_1)$, respectively. Upon measurement, the qubit collapses, which alters its amplitudes, and therefore the quantum state Q changes. Here, the quantum state Q is represented as $\alpha|Q_0\rangle + \beta|Q_1\rangle$. $|Q_0\rangle$ is the state with q collapsed to 0, and $|Q_1\rangle$, the state with q , collapsed to 1.

Evaluation of a function has two possible outcomes depending on whether the function has been evaluated before or not. If there is no value term associated with f in the environment γ , as in rule (f_1) , the term associated with the function is evaluated, and the resulting value term is returned. The value is then added to γ ; thus, reevaluating the function will follow the rule (f_2) , retrieving the value from the environment instead of evaluating the function term. The rules (f_1) and (f_2) ensure that linear functions are not incorrectly evaluated more than once. If the function term instead was evaluated every time, linear functions could be incorrectly be called more than once through a non-linear one.

The rule (U_n) shows the evaluation of gate application on n qubits with $n \geq 1$. The evaluation is deterministic and affects Q since a gate application entails a unitary transformation of the quantum state. The state change of Q depends on the applied gate and is not described by the evaluation rules.

The application rule (app) shows the call-by-value semantics since the argument is first evaluated to v , and then through β -reduction, the v is assigned to the variable x in γ . The updated state, γ' and Q' , are passed to evaluate M , which produces the final value w with updated states.

Tuples (tup) are evaluated left to right with the updated state passed on from t_1 to t_2 . For (let) , the term t_2 evaluates to v in an environment where the values v_1 and v_2 are assigned to x_1 and x_2 in γ' .

An if statement is evaluated under one of the two rules (if_1) and (if_0) . The former applies when t evaluates to 1, meaning that the if statement returns the value term obtained from evaluating t_1 . The latter rule applies when t evaluates to 0, returning the value obtained from t_2 .

3

Method

This chapter describes the overarching design choices that were made to realize funQ. The primary design choice was to create a stand-alone language with specified syntax, type checker, and interpreter with a separate library for quantum computation and simulation. The implementation of the components needed for a stand-alone language is explained in detail in chapter 4.

3.1 To Implement a Domain-Specific Language

Since funQ is a language for the quantum computation domain, it is considered a domain-specific language (DSL). As described by Fowler in [18], DSLs are built upon a general-purpose host language and come in two forms: external and internal. An external DSL is a language with a syntax that is parsed independently of the host language. An internal DSL is, on the other hand, represented within the syntax of the host language and is, as Fowler explains, "a stylized use of that [the host] language for a domain-specific purpose" [18].

The two approaches differ primarily in difficulty for a user to learn the language and the design freedom. Since an internal DSL is represented within the syntax of the host language, a user familiar with the host language should, in general, find an internal DSL more approachable than an external [18]. Imitating the syntactic conventions of a common programming language, such as the host language, can make the DSL more accessible for users of the host language. The effort of building a DSL is, according to Fowler, similar for external and internal once familiar with techniques for doing so. External DSLs offer greater syntactic freedom than internal and also allows, for example, the evaluation strategy to be manually configured.

An external DSL was chosen for funQ since it allowed the language to resemble QLambda more closely than an internal DSL would. It allowed defining a custom syntax and building a type checker and interpreter to follow the type and reduction rules of QLambda. For instance, linearity can be enforced in the type checker, and call-by-value can be adopted in the interpreter. These benefits made the option clear-cut to opt for an external DSL. An internal DSL would not allow for these custom behaviors due to the limitations imposed by the host language.

A tool named BNFC (BNF Converter) [19] was used to generate a lexer, parser, and an AST from a labeled BNF grammar (LBNF). This saved development time, and all effort could be put into the grammar that defines funQ. This tool also provided other utilities such as pretty-printing.

3.2 Host Language

Haskell was chosen as a host language for the external DSL. A functional language was self-evident since a functional quantum programming language was the intended goal of the project. Other than being functional, there are several reasons why Haskell was selected: other quantum programming languages studied were also implemented in Haskell (see section 1.2); Haskell is compatible with the BNFC tool; and finally, the authors' interest and knowledge about Haskell.

3.3 Testing

Tests were written continuously during the implementation and consists of small functional tests, unit tests, and property-based tests. The library and type checker were tested in isolation, while the functional tests used for the interpreter tested the whole chain from parsing to evaluation. Testing the code was done in parallel with the implementation, continuously ensuring that the codebase was working.

The quantum library was tested for correct norms in vector states and unitarity in gates. Property-based testing was utilized to ensure gate operations on qubits behaved correctly.

The type checker was tested with functional tests. The main focus was ensuring that linearity holds. Top- and term-level linearity were tested separately as they work on separate mechanisms. The tests use both well- and ill-typed programs to test that the type checker inferred the correct types and threw the expected type errors. Additionally, mathematical properties of the subtyping relation, supremum, and infimum operators were ensured through property tests.

The interpreter mainly used functional tests as well. A test suite of funQ programs was developed to catch erroneous programs and check expected results. Since the interpreter tests used the entire chain, this served as testing for all parts of the program.

Finally, the whole system was tested by implementing quantum algorithms and manually inspecting the outcome.

4

Implementation of funQ

This chapter describes the implementation of funQ in Haskell with concrete code examples along with explanations and design justifications. First, the syntax of funQ is discussed: how it is parsed, analyzed, type-checked and interpreted. Then the quantum library used to perform and simulate quantum computations is described. Figure 4.1 gives an overview of the implementation.

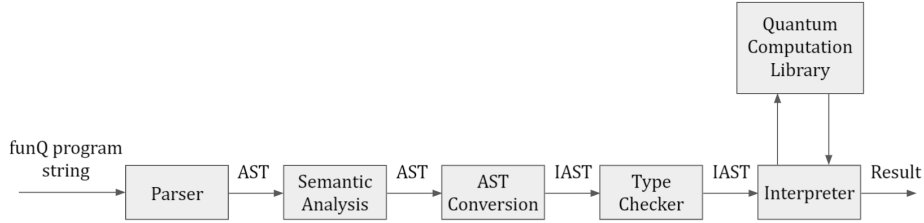


Figure 4.1: A high-level diagram of the implementation of funQ. The input is a source code string written in funQ. The output is the resulting value from the interpreter. AST refers to the abstract syntax tree created by the parser, and IAST refers to the intermediate abstract syntax tree.

4.1 Syntax and Grammar

The syntax of funQ is similar to Haskell’s and exemplified with the function in figure 4.2. Like Haskell, there is a function signature above the function definition. The function definition is given by a name, arguments, and followed by the body of the function. Other common constructs include let expressions and tuples.

```
qmap : (QBit -o QBit) -o (QBit >< QBit) -o (QBit >< QBit)
qmap f a = let (x,y) = a in (f x, y)
```

Figure 4.2: The source code of a function in funQ.

The syntax of funQ is described by a formal grammar written in LBNF and is used to create a parser. A language specification based on the grammar is given in appendix B. The LBNF grammar does not capture the exact syntax of funQ, and is intentionally made less strict. As a consequence, some terms with invalid funQ syntax are parsed successfully to enable more careful error handling. The additional restrictions needed are instead enforced in the semantic analysis, described in section 4.2.

A program in funQ is defined as a list of functions. Functions consist of a declaration with a name and a type signature, followed by a definition of the actual function (see figure 4.3). Therefore, all functions in funQ must be given a static type, which is helpful when type checking a program. This style is similar to how functions are declared in Haskell.

```

PDef. Program ::= [FunDec];

FDecl. FunDec ::= Var ":" Type Function;

FDef. Function ::= Var [Arg] "=" Term;

FArg. Arg ::= Var ;

```

Figure 4.3: The grammar for programs and function declarations in LBNF.

The label (the word before the dot) in the grammar becomes a constructor of the datatype and constitutes a part of the AST. For instance, the rule PDef in the grammar is translated to the abstract syntax data type `data Program = PDef [FunDec]` where PDef is the data constructor for a Program.

The definition of Term is shown in figure 4.4, and is semantically equivalent to the definition of terms given previously in section 2.4.1 in the theory chapter.

```

TVar . Term3 ::= Var ;
TBit . Term3 ::= Integer ;
TGate . Term3 ::= Gate ;
TTup . Term3 ::= "(" Term "," [Term] ")" ;
TUnit . Term3 ::= "*" ;
TApp . Term2 ::= Term2 Term3 ;
TIfEl . Term1 ::= "if" Term "then" Term "else" Term ;
TLet . Term1 ::= "let" "(" Var "," [Var] ")" "=" Term "in" Term ;
TLamb . Term1 ::= Lambda Var ":" Type "." Term ;
TDolr . Term1 ::= Term2 "$" Term1 ;

```

Figure 4.4: The grammar for terms. The suffix of Term determines its precedence when parsing. The second element of TTup is a list of terms to allow for tuples of any size. This is similarly done for the let binding.

All capital tokens are parsed as gates, where the ones not matching a concrete rule are parsed as a GateIdent with the GGate rule (see figure 4.5).

```

GH . Gate ::= "H" ;
GX . Gate ::= "X" ;
...
GCNOT . Gate ::= "CNOT" ;
GGate . Gate ::= GateIdent ;

```

Figure 4.5: An excerpt of the grammar for gates. The GGate rule parses all capital tokens as gates if it does not match another production rule.

The grammar for the types in funQ is shown in figure 4.6. The first three types corresponds to the constant types bit, qbit, and \top . TypeDup is the abstract syntax for $!$ and TypeTens is the constructor for the product type where $><$ is the syntax for the tensor product \otimes . Finally, TypeFunc is the abstract syntax for function types, where $-o$ is the syntax for the function arrow \rightarrow . Note that \otimes and \rightarrow are right-associative.

```

TypeBit . Type2 ::= "Bit"          ;
TypeQbit . Type2 ::= "QBit"        ;
TypeUnit . Type2 ::= "T"           ;
TypeDup . Type2 ::= "!" Type2      ;
TypeTens . Type1 ::= Type2 "><" Type1 ;
TypeFunc . Type1 ::= Type2 "-o" Type1 ;

```

Figure 4.6: The grammar for types.

4.2 Semantic Analysis

The AST generated by the parser is further analyzed for correctness in the semantic analysis. The semantic analysis checks that:

- function names are unique
- no duplicate function declaration exists
- function signatures and definitions have matching names
- the number of function arguments do not exceed the arguments in the signature
- integers can only be zero or one
- custom gates parsed as GGate are supported.

To show how semantic errors are handled, the checks of too many function arguments and supported gates serve as good examples. Starting with the former, only function definitions with equally many or fewer arguments than the signature are semantically correct. Fewer arguments are valid due to η -reduction. Consider figure 4.7, where the function takes an argument where none are expected. The error message describes the type of error and that the expected number of arguments is zero.

```

one : Bit
one a = 1

*** Exception, semantic error:
TooManyArguments: function one has 1 argument but it expects 0

```

Figure 4.7: The function `one` takes the argument `a`, but its function signature takes no arguments, which produces the semantic error `TooManyArguments`.

Regarding the gates, the semantic analysis verifies that gates parsed as GGate are supported by the language. If, for instance, a user tries to use an unsupported gate, `ZZ`, then the semantic error shown in figure 4.8 is produced.

```

λ: ZZ (new 0) (new 0)

*** Exception, semantic error:
UnknownGate: Gate not recognized: ZZ

```

Figure 4.8: An application of the gate '`ZZ`' which is not supported by `funQ`, and therefore produces the semantic error `UnknownGate`.

The purpose of the semantic analysis is twofold: to discover erroneous programs that would cause type checking or evaluation to fail and provide the user with informative error messages.

Having a semantic analyzer enables more informative error messages than parse errors, which is shown by comparing figures 4.8 and 4.9. Both figures show the same program. However, the semantic error in figure 4.8 gives more information about *why* the error occurred in contrast to the error in 4.9 that only points out *where* the error occurred.

```
λ: ZZ (new 0) (new 0)

*** Exception, syntax error:
syntax error at line 1, column 17 before `ZZ'
```

Figure 4.9: An application of the unsupported gate ZZ and the error that the parser would produce if not the more allowing grammar and semantic analysis was implemented.

Another benefit of this approach of having an allowing parser followed by a semantic analysis is that generic gates can be used. For instance, the controlled phase shift gate (CR) performs a phase shift with θ radians. This gate can generically be used by writing CR followed by an integer to represent the phase shift. Moreover, by appending an I to it, by writing CRI, the inverse operation can be performed. Generic gates are handled in the conversion to the intermediate AST, covered in the next section.

4.3 Conversion to an Intermediate Abstract Syntax

An intermediate step was introduced before the type checker and interpreter to simplify those parts. This step converts the BNFC-generated AST to an intermediate abstract syntax tree (IAST). The conversion includes: converting all functions into lambda abstractions, converting AST terms into IAST terms, and introducing De Bruijn indices for bound variables.

First, function terms are converted into lambda terms by replacing expressions of the form $f x_1 x_2 \dots x_n = term$ into $\lambda x_1 : \tau_1. \lambda x_2 : \tau_2. \dots \lambda x_n : \tau_n. term$. The type of each lambda is taken from the type signature. For example, in figure 4.10, the type of the argument x of the function `not` is found to be a bit by mapping the first argument x to the first type in the function signature. The outer `!` is ignored as it is irrelevant for the function's argument types.

```
not : !(Bit -o !Bit)
not x = if x then 0 else 1
```

Figure 4.10: A function `not`, for which the type of the argument x can be derived to `Bit` from the type signature.

The next step is to convert all AST terms into IAST terms. The intermediate abstract syntax for terms is shown in figure 4.11.

```
data Term = Idx Integer | Fun String | Bit Bit | Gate Gate
          | Tup Term Term | App Term Term | IfEl Term Term Term
          | Let Term Term | Abs Type Term | New | Meas | Unit
```

Figure 4.11: The terms in the IAST. `Var` from the generated AST is replaced by `Meas`, `New`, `Idx`, or `Fun`. The reduced number of parameters simplifies the structure of `Tup`, `Abs`, and `Let`.

The definition of Types from the generated AST is changed solely in notation. The type constructor for

product types, `TypeTens`, is replaced with the infix constructor `:><`, and the function type constructor, `TypeFunc`, is similarly replaced by `:=>`.

The conversion to IAST terms is done recursively in the function

`makeImTerm :: Env -> P.Term -> Term`. `Env`, short for environment, is a map from variable names to integers representing indices, used to keep track of the variables' De Bruijn indices during the conversion. This function makes the following conversions:

- The terms `Bit`, `App`, `IfEl`, and `Unit` are converted to their corresponding IAST terms.
- Custom gates are pattern matched and converted into terms.
- `Tuple Term [Term]` is converted into `Tup Term Term`, where the second term is converted into another tuple if the list of terms from the AST includes more than one term.
- `Var String` terms where the variable name matches "new", "meas" or "measure" are converted into the terms `New` and `Meas` respectively.
- For other `Var` terms, the variable name is looked up in the environment. If found, it is a bound variable and converted to `Idx i` where `i` is the De Bruijn index found in the environment. Otherwise, it is a free variable which, in practice, is a function call. Therefore, it is converted to the term `Fun String`, which is the data type for function calls in the IAST.
- `TDolr Term Term` is converted into `App Term Term`.
- `TLamb Lambda FunVar Type Term` is converted to `Abs Type Term`. `Lambda ("\"`) is simply removed. `Type` is converted to the corresponding IAST type. The lambda binds a variable that is added to the environment with 0 as the De Bruijn index. All other indices in the environment are incremented by one since they are now at one higher abstraction level. Finally, the body term is converted through a recursive call of `makeImTerm`.
- The cumbersome let term from the AST `TLet LetVar [LetVar] Term Term` is simplified to `Let Term Term`. A regular let expression written on the form "let (a,b) = X in M" has two bound variables, `a` and `b`, that are converted to De Bruijn indices 0 and 1 respectively and added to the environment. All other indices are incremented by two. If the list of variables contains more than one element, the let expression binds more than two variables, e.g., "let (a, b, c) = X in M". Since let terms only contain two terms, the second term becomes a nested let term.

The data type representing a funQ program is also converted to the IAST representation. A program in the AST is a list of function declarations (`FunDec`) that contains both the type signature and function definition. A comparison of a program in the AST and IAST is shown in figure 4.12 where a program is simplified to be a list of more easily used IAST functions.

<code>data Program = PDef [FunDec]</code>	<code>type Program = [Function]</code>
<code>data FunDec = FDecl FunVar Type Function</code>	<code>data Function = Func String Type Term</code>
<code>data Function = FDef Var [Arg] Term</code>	

Figure 4.12: The program definition in the AST (left) and in the IAST (right). A program given in the IAST version is simplified — the function type is included in the function instead of the function signature, the arguments are given in the term as De Bruijn indices, and the name duplication is removed (`FunVar` and `Var`).

The IAST is generally structured more conveniently than the generated AST, for example, by having right nested tuples instead of a term and a list of terms. Also, unnecessary syntax information needed in the parser is removed, such as the `"\"` in the lambda term. Functions are converted to lambda abstractions so that the lambda calculus can be used, which the typing rules and evaluation rules are based upon. The introduction of De Bruijn indices allows removing the lambda variable name, this avoiding name collisions. The conversion also enabled the simplified way of representing a program as a list of functions.

4.4 Type Checking

The type checker implements the typing rules from figure 2.11 explained in the theory section. It uses a linear type system that allows subtyping. The type checker algorithm works by inferring types of all program terms and check those against the type given in the function signature.

4.4.1 The *Check* Monad

All type checking occurs in the so-called Check monad, an except monad transformer consisting of a reader monad, state monad, and an except monad. The Check monad is defined as

`type Check = ExceptT TypeError (ReaderT TopEnv (State LinEnv))`. The except monad is used to throw exceptions when a type error is found (the complete list of type errors is found in appendix C). The internal reader monad contains a map of top-level functions and their types. Finally, the state monad contains a set of top-level linear functions that has been used thus far in the type checking process, referred to as the linear environment.

4.4.2 Type Checking Algorithm

The type checking algorithm is implemented as follows:

- Create the top-level typing context.
- Infer the type of each top-level function recursively.
- Check that the inferred type of a function is a subtype of the signature type.

Type checking a program is done by inferring the type of each function following the typing rules in section 2.4.3 (hereafter denoted by their names). The type of each function is inferred by recursively inferring the type of the function term. The function with the following type signature `inferTerm :: [Type] -> Term -> Check Type` is used to infer the type of a term given a typing context (a list of types). The typing context is used to keep track of the types of all variables that are bound by either lambda or let terms. The function operates within the Check monad to throw a type error if found and have access to the global environments. Each term is inferred somewhat differently, and the different cases in `inferTerm` are explained below and mapped to their corresponding typing rules.

Base terms are mapped to their corresponding base types, shown in figure 4.13, following the typing rules prefixed with *ax* in figure 2.11. Gate types are handled in a special way since they depend on the size of a gate. For instance, a gate operating on two qubits will return the type $!((qbit \otimes qbit) \multimap qbit \otimes qbit)$.

```
inferTerm _ Unit      = return $ TypeDup TypeUnit
inferTerm _ (Bit _)   = return $ TypeDup TypeBit
inferTerm _ New       = return $ TypeDup (TypeBit :=> TypeQBit)
inferTerm _ Meas      = return $ TypeDup (TypeQBit :=> TypeDup TypeBit)
inferTerm _ (Gate g)  = return $ inferGate g
```

Figure 4.13: Type inference for base cases for which the typing context is not needed. Inferring gate types are separated into a function `inferGate` that takes the size of the gate into account.

The type of bound variables is looked up in the typing context, corresponding to the type rule (*var*). For example, the variable with De Bruijn index 0 has the type most recently added to the context; hence the inferred type is the type on index 0 in the context (see figure 4.14).

```
inferTerm ctx (Idx i) = return $ ctx !! fromIntegral i
```

Figure 4.14: Type inference of bound variables. The type is found by its index in the typing context.

The type of a free variable, which is a function call in practice, is looked up in the top-level typing context. Function calls also correspond to typing rule (*var*). If the function has a linear type, it is added to the linear environment. If the function is already in the set, a linearity error is thrown (see figure 4.15).

```
inferTerm _ (Fun fun) = do
  top <- ask
  lin <- gets linenv
  case M.lookup fun top of
    Nothing -> throwError $ NotInScope fun
    Just t | isLinear t -> if S.member fun lin
      then throwError $ NotLinearTop fun
      else modify (\s -> s {linenv = S.insert fun lin})
    >> return t
  | otherwise -> return t
```

Figure 4.15: Type inference of a function call. The function name is looked up in the top environment, and the type is checked for linearity. If the function is not found or if it is a linear function that has already been used, a type error is thrown. Otherwise, the type is returned.

When inferring the type of an if term, the condition term and the two branches must be considered (see figure 4.16). The condition is checked to be a subtype of bit. The type of the then and else branches is then inferred in parallel by the function `parallelCheck` to allow free linear variables to be used in both branches since only one of the two will be evaluated. The type of the whole term is the least common supertype between the types of the then and else branches, found with the supremum function (\vee). This corresponds with the typing rule (*if*).

```
inferTerm ctx (IfEl c t f) = do
  tc <- inferTerm ctx c
  (tt, tf) <- parallelCheck (inferTerm ctx t) (inferTerm ctx f)
  if tc <: TypeBit
    then supremum tt tf
    else throwError $ Mismatch TypeBit t
```

Figure 4.16: Type inference of an if term. The `parallelCheck` assures that a linear top-level function can be used in both the then and the else branch. A type error is thrown if the condition is not a subtype (\leq) of bit.

The case for lambda terms is shown in figure 4.17. If the lambda argument has a linear type, it is checked to be used at most once in the body using the function `checkLinear`. If the argument is linear and used more than once, $\Omega_x > 1$, an error is thrown. If the body does not use any free linear variables, `boundLin` and `freeLin` are false, and the function type is made duplicable according to the (λ_2) rule. Otherwise, the function is given a linear type by (λ_1).

```

inferTerm ctx (Abs t e) = do
  top <- ask
  checkLinear e t
  et <- inferTerm (t:ctx) e
  let boundLin = any (isLinear . (ctx !!)) . fromIntegral) (freeVars (Abs t e))
      freeLin = any isLinear $ mapMaybe (`M.lookup` top) (names e)
  if boundLin || freeLin
    then return (t :=> et)
    else return $ TypeDup (t :=> et)

```

Figure 4.17: Type inference of a lambda term. The type of the argument is added to the type context when inferring the body.

Inferring let terms is one of the more complex cases. For a let expression `let (a,b) = x in M`, `x` corresponds with the `eq` term and `M` corresponds with the `inn` term in figure 4.18. The type of `eq` is inferred first, and is expected to be a product type. All prepended `!` outside the product type is moved inside to the component types of the product type, according to the (*let*) rule. Similarly to lambdas, the linear components are checked to be used at most once. The resulting type is the type of the `inn` term.

```

inferTerm ctx (Let eq inn) = do
  teq <- inferTerm ctx eq
  let nBangs = numBangs teq
  case debangg teq of
    (a1 :>< a2) -> do
      let a1t = addBangs nBangs a1
          a2t = addBangs nBangs a2
      checkLinear inn a2t
      checkLinear (Abs a2t inn) a1t
      inferTerm (a2t : a1t : ctx) inn
    _ -> throwError $ NotProduct teq

```

Figure 4.18: Type inference of a let term. All bangs are moved into the product components. The product types `a2t` and `a1t` are added to the context when inferring the `inn` term. The term `inn` is nested in another lambda when checking that the second variable is linear, so it counts usages of the correct variable.

The application case is shown in figure 4.19. The function term is checked to be a function type, and the argument is checked to be a subtype of the function's argument type. The result is the function's return type which follows from the (*app*) rule.

```

inferTerm ctx (App f arg) = do
  tf <- inferTerm ctx f
  argT <- inferTerm ctx arg
  case debangg tf of
    (fArg :=> fRet) | argT <: fArg -> return fRet
    | otherwise -> throwError $ Mismatch fArg argT
    _ -> throwError $ NotFunction tf

```

Figure 4.19: Type inference of function application. Any `!` is removed before pattern matching a function.

The last case, type inference of tuples, can be seen in figure 4.20. The inferred type is the product type of the inferred type of the left and right term, respectively, where any common ! is factored out, as stated by the (\otimes) rule.

```
inferTerm ctx (Tup l r) = do
  lt <- inferTerm ctx l
  rt <- inferTerm ctx r
  return $ shiftBang (lt :>< rt)
```

Figure 4.20: Type inference of tuples. If both *lt* and *rt* are duplicable, the tuple will be made duplicable by *shiftBang*.

4.5 Interpreting

Evaluation of a funQ program entails both classical and quantum operations. The interpreter primarily handles the classical control, implements the call-by-value semantics, and holds the environment. The interpreter delegates all quantum responsibilities to the quantum library. The library handles the quantum state and the operations that modify it and is described in section 4.6. The interplay between the interpreter and the quantum library is similar to the QRAM model with its classical computer connected to a quantum device. The interpreter and the quantum library together implement the evaluation rules for funQ, described in section 2.4.5. Recall the program state consisting of the environment γ and the quantum state Q . The interpreter handles γ and any updates of it; the quantum library does the same for Q .

The entry point for evaluation of any funQ program is the *main* function, which can call other methods defined in the program. The interpreter takes a funQ program as input and, after having reduced the main term into a value term, returns that value.

4.5.1 The *Eval* Monad

Interpreting is done within the *Eval* monad, defined as `type Eval = StateT FunctionValues QM`. To allow connection to the quantum library, *Eval* contains the quantum monad *QM*, which is defined in the quantum library and contains the quantum state. The quantum state in *QM* corresponds with quantum state Q in the evaluation rules in section 2.4.5. *Eval* also contains a map called *FunctionValues*, which is the values of evaluated function terms. This map constitutes one of the three parts of the environment γ from section 2.4.5.

4.5.2 Environment

The environment contains the bound variables and function terms (see figure 4.21). The bound variables are stored as a list, where the list indices map to the respective De Bruijn index that was introduced in the IAST. This environment together with the *FunctionValues* constitutes the entire environment of γ .

```
data Env = Env
  { values      :: [Value]
  , functions  :: Map String Term
  }
```

Figure 4.21: Definition of the environment. The environment holds a list of values of bound De Bruijn variables and a map from function names to terms.

4.5.3 Evaluation Algorithm

The evaluation algorithm is implemented by first adding all functions to the environment and then evaluating the main function term using the function `eval :: Env -> Term -> Eval Value`. The function takes an environment and a term as argument, the term is evaluated to a value and then returned wrapped in the Eval monad. Case analysis is performed on the term since each term is evaluated differently and the different cases are explained below. Since the interpreter implements the call-by-value strategy, arguments are always evaluated through recursive calls of `eval` before the body is evaluated. Evaluation of a term proceeds until a value term is returned.

The simplest cases of evaluation are shown in figure 4.22, where base terms evaluate to their corresponding value terms. The first three cases correspond to evaluation rules (b_0) , (b_1) and $(unit)$. The purpose of evaluation of unapplied new, measurement, and gate terms is to support η -reduction. The term `e` in `Abs` may contain variables bound by previous lambda abstractions. The values from the environment are included in the value term `VAbs` to preserve the equality between the index of a value in values and a De Bruijn index in `e`. Tuples are evaluated by first evaluating the two terms individually, then wrapping the results in the value term `VTup` which corresponds to the evaluation rule (tup) .

```
eval env (Bit BZero) = return $ VBit 0
eval env (Bit BOne)  = return $ VBit 1
eval env (Unit)      = return VUnit
eval env (Gate g)    = return $ VGate g
eval env (New)       = return VNew
eval env (Meas)      = return VMeas
eval env (Abs t e)   = return $ VAbs(values env) t e
eval env (Tup t1 t2) = do
  v1 <- eval env t1
  v2 <- eval env t2
  return $ VTup v1 v2
```

Figure 4.22: Evaluation of the trivial cases. Terms are mapped to their corresponding value term.

The evaluation for `App`, that is, the application of one term (`t1`) to another (`t2`) can be seen in figure 4.23. Depending on the kind of `t1`, the evaluation is performed differently. Therefore, it begins with a case expression with cases for `New`, `Meas`, `Gate`, while any other term falls through to the last wildcard case. The first two cases are evaluated similarly by first evaluating `t2`, obtaining a value term, and then calling the appropriate quantum library function with that value term. The `Gate` case includes a case expression with pattern matching on all defined gates. Due to limited space, figure 4.23 only shows two cases, but all the gates that the language provides are evaluated similarly. The `runGate` function handles application of a given gate to a single qubit and returns a `VQBit` wrapped in the `Eval` monad. The interpreter includes similar functions for applying gates that operate on more than one qubit.

In the last case of evaluating `App`, both constituent terms are evaluated before pattern matching on the value term of the first term. If `v1` is a `VAbs` (a lambda value term), then the value term `v2` and the values from the `VAbs` term are added to the list of values in the environment. Adding the values in front of the list gives them indices equal to their De Bruijn indices in the body, which will be evaluated in the updated environment under the evaluation rule (app) . The value terms `VNew`, `VMeas` and `VGate` are evaluated similarly by adding `v2` to the values in the environment, which gives `v2` index 0 in the list. With the updated environment, the evaluation proceeds with an application of the corresponding term to `Idx 0`.

```

eval env (App t1 t2) = case t1 of
  New -> do
    VBit b' <- eval env t2
    lift $ VQBit <$> new b'
  Meas -> do
    VQBit q' <- eval env t2
    lift $ VBit <$> measure q'
  Gate g -> case g of
    GH      -> runGate hadamard t2 env
    GX      -> runGate pauliX t2 env
    ...
  _ -> do
    v2 <- eval env t2
    v1 <- eval env t1
    case v1 of
      VAbs vs _ a -> eval env{ values = v2 : vs ++ values env } a
      VNew -> eval env{ values = v2 : values env }
              (App New (Idx 0))
      VMeas -> eval env{ values = v2 : values env }
              (App Meas (Idx 0))
      (VGate g) -> eval env{ values = v2 : values env }
              (App (Gate g) (Idx 0))

```

Figure 4.23: Evaluation of the application term. *New*, *Meas*, and *Gate* must be handled separately and are cased out.

Evaluation of **Idx** is the β -reduction of lambda terms, where a De Bruijn index is substituted with the corresponding value. As shown in figure 4.24, an index j evaluates to the value at index j in the list of values in the environment.

```

eval env (Idx j) = return $ values env !! fromIntegral j

```

Figure 4.24: The evaluation of an index term, which is the β -reduction of lambda terms.

Figure 4.25 shows that the evaluation of **Fun**, a function, depends on whether the function has been called before or not. The first time a function is called, the *Nothing* case is executed and its corresponding term is evaluated. The resulting value term is stored in **FunctionValues** and returned in accordance with evaluation rule (f_1). When evaluating a function that has been evaluated before, the lookup is successful and the saved value term is returned, thus not re-evaluating the function term. This corresponds to the rule (f_2).


```

eval env (Fun s) = do
  let (Just t) = getTerm env s
  fs <- get
  case M.lookup s fs of
    (Just v) -> return v
    Nothing -> do
      v <- eval env t
      modify (M.insert s v)
      return v

```

Figure 4.25: Evaluation of a function call. If the function is already evaluated, the value is returned. Otherwise, the function term is evaluated and the environment updated.

Evaluation of a let term is done in two steps and corresponds to evaluation rule (*let*). The tuple eq is evaluated first and the resulting values are added to the list of values in the environment. Adding the newly evaluated value terms in front of the list gives them indices equivalent to their De Bruijn indices in the term inn. Then, the inn term is evaluated with the updated environment.

```

eval env (Let eq inn) = do
  VTup v1 v2 <- eval env eq
  eval env{ values = v2 : v1 : values env } inn

```

Figure 4.26: Evaluation of a let term. The value of the eq term is first evaluated as a tuple and then added to the environment before evaluating the inn term.

If terms are evaluated by first evaluating the term t and then, based on the resulting bit value, evaluating either t1 or t2. The evaluation corresponds to evaluation rules (*if*₀) and (*if*₁).

```

eval env (IfEl t t1 t2) = do
  VBit b <- eval env t
  eval env $ if b == 1 then t1 else t2

```

Figure 4.27: Evaluation of an if term. Depending on the value of b, either t1 or t2 is evaluated.

4.6 Library for Quantum Computations

The library for quantum computations holds the quantum state and performs all quantum operations which operate on that state. Thus, the library implements the evaluation rules for new, measurement, and gate application, defined in section 2.4.5, and gives rise to the probabilistic element of the rules. Documentation of the library can be found on the documentation page¹.

4.6.1 The Quantum State

The quantum state is implemented as a complex vector, `QState`, that corresponds to `Q` in the evaluation rules. It is accessed through the `QM` monad, in which all computations of the quantum library are performed. See figure 4.28 for the definition of `QM` and `QState`. The `IO` monad is included in `QM` to allow pseudo random number generation for the probabilistic measurement operation.

```
newtype QState = QState { state :: Vector C }
newtype QM a = QM { runQM :: QState -> IO (a, QState) }
```

Figure 4.28: The quantum state `QState` and the quantum monad (`QM`), which all quantum operations are done within.

4.6.2 Qubits and Gates

Qubits are implemented by the `QBit` type that is a unique integer pointing to a qubit in the `QState`, which is defined as `newtype QBit = Ptr { link :: Int }`. The `QBit` pointer allows access to single qubits, which is needed to enable the application of gates and measurement of specific qubits.

The library includes functions for all gates, `U`, defined in the syntax for `funQ` in section 2.4.1. The gate operations follows the (U_n) rule in section 2.4.5 in the theory chapter. The rule is obeyed by performing a deterministic operation that returns the same amount of qubits as its input.

The gates are represented by matrices and used as functions. All gate functions have in common that they take one or more qubits as input, apply its gate transformation, and then return the updated quantum state along with the input qubits. The gate matrix is deterministically applied through vector multiplication to the specific qubit(s) it should act upon.

For example, consider the function `hadamard` representing the Hadamard gate in figure 4.29. The input qubit is passed to the `runGate` function, which applies the Hadamard matrix to this specific qubit in `Q`. This is done by combining identity matrices for all other qubits in the state that the gate should not operate on and putting the Hadamard matrix in the correct position to exclusively apply the gate to the desired qubit. In other words, the Hadamard gate is only applied to the input qubit and leaves all other qubits unchanged. The other gates are implemented similarly.

```
hadamard :: QBit -> QM QBit
hadamard = runGate hmat

hmat :: Matrix C
hmat = scale (sqrt 0.5) $ (2 >< 2)
  [ 1 , 1
    , 1 , -1 ]
```

Figure 4.29: The library function `hadamard`. It uses the `runGate` helper function that takes a qubit as input to apply the gate to a specific qubit. The function `hmat` is the matrix representation of the Hadamard gate.

¹Documentation of the library for quantum computation: <https://qfunc.nicbot.xyz/qfunc-0.1.0.0/FunQ.html>

4.6.3 New and Measure

The creation and measurement of qubits are implemented as two separate functions. The creation of qubits is done with function `new` (see figure 4.30), which is involved in realizing the rules (new_0) and (new_1) . The function `new` maps a classical bit to a qubit and updates the quantum state. It works by creating the bit's corresponding $|0\rangle$ or $|1\rangle$ qubit vector and adding it to `Q`. The qubit is added to the quantum state `Q` by taking the tensor product of `Q` and the new qubit vector, which produces a new and extended state vector. The size of the state vector that represents the quantum state is doubled each time `new` is invoked. Finally, it returns a pointer to the newly added qubit.

```
new :: Bit -> QM QBit
new x = do
  (_,size) <- getState
  modify $ appendState (newVector x)
  return $ Ptr size
```

Figure 4.30: The library function `new`, which adds a qubit to the quantum state and returns a pointer to that qubit.

The dual function, `measure`, is involved in realizing the rules $(meas_0)$ and $(meas_1)$. It collapses a qubit to a bit, and can be seen in figure 4.31. The `measure` function works by first calculating the probability of the qubit measuring to a 1, then using a pseudo random number generator to simulate whether it collapses to 1 or 0, and finally updating the quantum state according to the observation and returning the collapsed value. The probability of a qubit collapsing to 1 is the sum of the probabilities of all outcomes where it collapses to 1, which would be half of the outcomes of measuring the whole system. As measurement consumes the qubit being measured (and such its pointer to the quantum state), it is impossible to re-measure that qubit. Therefore, the state is updated by setting all outcomes that contradict the result to have a probability of zero. The quantum state vector is then normalized, the sum of all probabilities in the vector is one.

```
measure :: QBit -> QM Bit
measure qbit = do
  state <- get
  let p1 = findQbitProb1 qbit state
  bit <- io $ rngQbit p1
  let newState = remImpossibleStates state qbit bit
  put newState
  return bit
```

Figure 4.31: The library function `measure`, which collapses the qubit and returns a bit value.

5

Results

This project has resulted in an implementation of a functional quantum programming language that can be used to write quantum programs with the help of a quantum computer simulator. Some well-known quantum algorithms have been implemented in funQ, which are presented in this section. Users can interface with the language through the command line tool that reads and evaluates program files or expressions written inside the terminal.

5.1 The funQ Language

A short overview of the functionality of funQ is given here, although most constituent parts of the language have already been explained. Further details can be found on the project's Github repository with usage instructions ². Which also contains information on how funQ can be installed and used. The functionality of funQ includes:

- if expressions for conditional logic
- product types to group data and let expressions to pick apart product types
- linear types
- call-by-value evaluation
- new and measure functions to create and measure qubits
- built-in gates to operate on qubits
- comments
- named functions with type signatures
- recursion.

A feature of funQ not yet explained is recursion. The language allows users to define named functions in a Haskell-like manner. Immediately storing these functions in a global environment (as explained in chapter 4) enables function calls to be made recursively. An example of a funQ program using recursion is shown in figure 5.1.

```
constZero : !(Bit -o Bit)
constZero x = if x then 0 else constZero (not x)

not : !(Bit -o Bit)
not x = if x then 0 else 1
```

Figure 5.1: A recursive function `constZero`. The function calls itself recursively in the else clause.

²The project repository <https://github.com/NicklasBoto/funQ> A short guide with programming tasks is found at <https://github.com/NicklasBoto/funQ-tasks>

5.2 Quantum Algorithms in funQ

Several quantum algorithms were written in funQ, as a way of evaluating both the correctness and usability of the language. The corresponding quantum circuit is included for the coin flip, quantum teleportation, and Deutsch-Jozsa algorithm to see how it compares to a funQ implementation.

5.2.1 Coin Flip

One of the simplest quantum programs implemented is the "coin flip", randomly generating a one or zero with 50 % probability, similarly to a fair coin flip. A quantum circuit for this program is shown in figure 5.2.



Figure 5.2: The "coin flip" circuit. A Hadamard gate is applied to a qubit representing one of the basis vectors, which changes the probability of the qubit collapsing to zero or one equal upon measuring. The qubit is then measured, which will result in a zero or one.

Writing this program in funQ is straightforward, as seen in figure 5.3.

```
coinFlip : Bit
coinFlip = measure (H (new 0))
```

Figure 5.3: The funQ implementation of a simulated fair coin flip. The function measures a qubit that has been applied by the Hadamard gate.

First, new 0 creates the zero qubit, then the Hadamard gate and measurement operation is applied as in the circuit. Running the `coinFlip` function many times in the funQ command line tool (explained more in section 5.3) results in a distribution that approximately gives zero or one with a 50% probability (see figure 5.4).

```
$ funq coinflip.fq --runs=1000
0:      49.6%   496
1:      50.4%   504
```

Figure 5.4: Output from running a `coinFlip` simulation repeatedly, resulting in an approximately equal distribution between 0 and 1.

This example also serves as an indicator that the probability logic when measuring qubits works well.

5.2.2 Quantum Teleportation

Another example of a quantum algorithm implemented in funQ is quantum teleportation. Quantum teleportation is a way to use quantum entanglement and classical communication to teleport a qubit between two locations [20]. It is possible to use entanglement to transfer one quantum state to another, even when large distances separate the qubits. This is done using a quantum teleportation protocol, represented by the quantum circuit in figure 5.5.

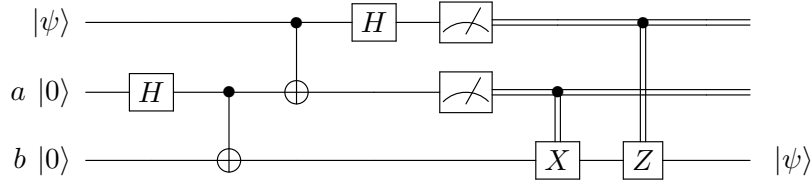


Figure 5.5: The teleportation circuit. The following is needed to perform quantum teleportation: three qubits, six gates, and two measurement operations.

The funQ code for quantum teleportation is shown in figure 5.6. Note that the qubits in the circuit in figure 5.5 correspond to `psi`, `a`, and `b` in the code. The zero-qubits `a` and `b` become entangled and forms a Bell state by the `epr` function. After that, the Bell measurement determines the Bell state of the qubits. This measurement produces a pair of bits, which along with the qubit to be teleported, is sent into the `correction` function that then outputs qubit `b` with ψ 's state teleported to it.

```
teleport : !(QBit -o QBit)
teleport psi = let (a,b) = epr * in correction b $ bellMeasure a psi

bellMeasure : !(QBit -o QBit -o (Bit >< Bit))
bellMeasure a b = let (x,y) = CNOT (a,b) in (measure (H x), measure y)

epr : !(T -o QBit >< QBit)
epr x = CNOT (H (new 0), new 0)

control : !((QBit -o QBit) -o Bit -o QBit -o QBit)
control g b = if b then g else I

correction : !(QBit -o (Bit >< Bit) -o QBit)
correction q bits = let (a,b) = bits in control Z a $ control X b q
```

Figure 5.6: The funQ implementation of the quantum teleportation algorithm. The circuit operators (gates and measurement) do not have a one-to-one mapping with the functions written in funQ.

5.2.3 Deutsch-Jozsa Algorithm

Deutsch-Jozsa is another well-known quantum algorithm that can determine if a function, known as the oracle, is balanced or constant. A balanced function will return 0 and 1 equally many times, while a constant function returns the same output for all inputs. It is one of the earliest discovered examples of a quantum algorithm that solves a problem exponentially faster than any deterministic classical algorithm [21]. The algorithm determines whether a given function is balanced or constant in a single call to the function, which would require at most $2^{n-1} + 1$ calls in the classical solution. The quantum circuit diagram of the algorithm is shown in figure 5.7.

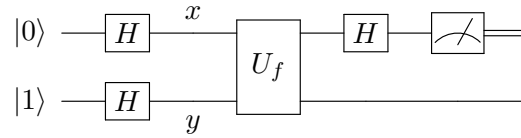


Figure 5.7: The Deutsch-Jozsa circuit. Where U_f is the oracle containing f , mapping $|x\rangle \mapsto |x\rangle$ and $|y\rangle \mapsto |y \oplus f(x)\rangle$ where \oplus is addition modulo 2.

The implementation is shown in figure 5.8, where the function `deutsch` will return a 0 if given a constant oracle and a 1 if given a balanced one.

```
balanced : (QBit >< QBit) -o QBit
balanced qs = let (x,y) = qs in
               let (x,y) = CNOT (X x, y)
               in (X x)

deutsch : ((QBit >< QBit) -o QBit) -o !Bit
deutsch oracle = measure $ H $ oracle (H (new 0), H (new 1))
```

Figure 5.8: The funQ implementation of the Deutsch-Jozsa algorithm.

5.2.4 Grover's Algorithm

Grover's algorithm is a quantum search algorithm that can find an element in an unsorted list of size N in $\mathcal{O}(\sqrt{N})$ steps [22]. The quantum circuit diagram of the algorithm is represented in figure 5.10.

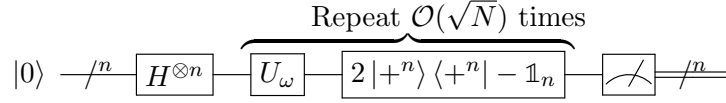


Figure 5.9: The Grover circuit for n qubits. $H^{\otimes n}$ is n parallel Hadamard gates and $2|+^n\rangle\langle +^n| - \mathbb{1}_n$ is the Grover diffusion operator [23].

To search a list of length N , $n = \lceil \log_2 N \rceil$ qubits are needed. To find the desired element at index ω the oracle U_ω is constructed according to equation 3.

$$U_\omega|x\rangle = \begin{cases} |x\rangle & \text{if } x \neq \omega \\ -|x\rangle & \text{if } x = \omega \end{cases} \quad (3)$$

The three qubit implementation shown in figure 5.10 can be used to search a list, of maximum length eight. The function `oracle` implements the action $|abc\rangle \mapsto |a\rangle \otimes CZ|bc\rangle$ which is the oracle $U_{3,7}$ [22]. As this oracle encodes more than one correct index, the program will return a uniform distribution of the correct indices.

```
diffuser : !((QBit >< QBit >< QBit) -o QBit >< QBit >< QBit)
diffuser qs = map H $ CCR1 $ map Z $ map H qs

oracle : !((QBit >< QBit >< QBit) -o QBit >< QBit >< QBit)
oracle qs = let (a,bs) = qs in (a, CR1 bs)

grover : !(QBit -o QBit -o QBit -o QBit >< QBit >< QBit)
grover q0 q1 q2 = diffuser $ oracle $ map H (q0,q1,q2)
```

Figure 5.10: The funQ implementation of Grover's algorithm. The function `map` applies a gate to every element in a 3-tuple.

Running the program gives the results shown in figure 5.11.

```
$ funq grover.fq --runs=1024
011:    49.31%  505
111:    50.68%  519
```

Figure 5.11: Output from running Grover's algorithm, resulting in 3 and 7.

5.2.5 Shor's Algorithm

Shor's algorithm is a quantum algorithm for factorizing integers [5]. The general algorithm entails converting the factoring problem into an order-finding problem. Here, the order denotes the smallest number r such that $a^r \equiv 1 \pmod{N}$, for a number to be factored N , and a factor guess a such that $\gcd(N, a) = 1$. If r is even, the above equivalence gives that $(a^r - 1) = (a^{r/2} - 1)(a^{r/2} + 1) \equiv 0 \pmod{N}$. Thus, both $(a^{r/2} - 1)$ and $(a^{r/2} + 1)$ divide N . Then, $\gcd(N, a^{r/2} - 1)$ and $\gcd(N, a^{r/2} + 1)$ are calculated classically and checked to see if they are factors of N . If they are factors of N , the algorithm is complete, otherwise input a different a and start over.

Below, two different implementations of Shor's algorithm are presented. First, a specialized implementation of Shor's algorithm that can factor the number 15, then a more general version of the order-finding part of Shor's algorithm is shown.

```
init : !((QBit >< QBit >< QBit >< QBit >< QBit)
      -o QBit >< QBit >< QBit >< QBit >< QBit)
init qs = let (a,b,c,d,e) = qs in
          let (a,b,c) = QFT3 (a,b,c) in (a,b,c,d,e)

shor : !((QBit >< QBit >< QBit >< QBit >< QBit)
      -o QBit >< QBit >< QBit >< QBit >< QBit)
shor qs = let (a,b,c,d,e) = init qs in
          let (b,c,d)      = (H b, CNOT (c,d)) in
          let (c,e)        = CNOT (c,e) in
          let (b,a)         = CR2 (b,a) in
          let (a,b,c)       = (H a, CR4 (b,c)) in
          let (a,c)         = CR2 (a,c) in (a,b,c,d,e)
```

Figure 5.12: The funQ implementation of Shor's algorithm for $N = 15$ with an initial guess of 11.

The program in figure 5.12 factors the number 15, with a guess of 11 [24]. Running the program gives the resulting numbers 4 and 0, as shown in figure 5.13. They are called periods and period 0 is discarded since it is trivial. Following [25] it can be derived from the period of 4 that the factors of 15 are 3 and 5, using efficient classical computations.

```
$ funq examples/shors.fq --runs=1024
00000: 47.94% 491
00100: 52.05% 533
```

Figure 5.13: Output from running Shor's algorithm factoring 15, resulting in the periods 0 and 4.

The order-finding part of Shor's algorithm is given by the function QOF (Quantum Order Finding) in figure 5.14. It makes use of the one-qubit control trick, described in [26]. The measured period is then returned as a number of bits, which can be used to find the factors of the number to be factored. The arguments of the function QOF are, going from left to right: the register used by the sub-components of QOF; the factor guess; the number to be factored; multiplication control qubits. Finally, the output is a tuple containing the input qubit arguments and the measured results of the order-finding algorithm.

$$QOF : !(qbit^6 \multimap !bit^3 \multimap !bit^3 \multimap qbit^3 \multimap qbit^9 \otimes !bit^6)$$

Figure 5.14: Type signature for the quantum order finding algorithm (QOF). The full function definition is in appendix D

5.3 Command Line Tool

A command line tool was developed to help reach the goal of funQ being easy to use. The command line tool can be used to run funQ program files or, interactively, by writing expressions directly in a read-eval-print loop (REPL). It is also possible to load funQ files into the interactive mode by the `-i` flag or with the `:load` command while in interactive mode.

Consider the example below where the function `coinFlip` is created interactively:

```
funQ 0.9.1
:? for help
λ coinFlip = meas $ H $ new 0
```

Typing a single expression results in a call to evaluate this expression. What actually happens is that a function `main = coinFlip` is created and added to the environment before the program is checked and evaluated. The result from the evaluation is then printed together with the type of the expression:

```
λ coinFlip
1 : !Bit
```

In interactive mode, the environment (a set of functions) is updated for every new function typed by the user. For example, a product of two qubits `q` is created:

```
λ q = (new 0, new 1)
```

It is then possible to view the type of the function with the `:t` command equivalent to the `:t` command in the Haskell compiler `GHCI`, which prints the type of the function:































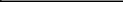
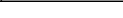
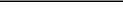
```
λ :t q
QBit ⊗ QBit
```

5.4 Test Results

All tests for the quantum library, type checker, and interpreter passes. Although not certain, this suggests that all aspects the tests-cases cover are correctly implemented. There are no known bugs.

A test coverage report has been generated, which can be seen below in table 5.1. Top level definitions, alternatives, and expressions have code coverage of 59%, 60%, and 55%, respectively. The coverage for each module is shown for each of these different categories.

Table 5.1: Test coverage from running the test programs with coverage. Top level definitions, being all top level functions, has 59% coverage. Alternatives, being case branches, has 60% coverage. Among all expressions in the project, 55% are tested.

module	Top Level Definitions			Alternatives			Expressions		
	%	covered / total		%	covered / total		%	covered / total	
module Lib.Internal.Core	100%	11/11		100%	8/8		100%	107/107	
module Type.TypeChecker	94%	32/34		83%	88/105		82%	571/693	
module SemanticAnalysis.SemanticAnalysis	90%	9/10		65%	25/38		65%	221/336	
module Lib.Internal.Gates	85%	17/20		83%	5/6		73%	220/299	
module Interpreter.Interpreter	75%	12/16		74%	47/63		76%	338/440	
module Lib.Gates	66%	12/18		33%	2/6		51%	124/240	
module Lib.QM	60%	17/28		100%	2/2		56%	89/157	
module Lib.Core	33%	2/6		0%	0/2		30%	28/92	
module AST.AST	30%	14/46		48%	39/80		41%	206/495	
module Interpreter.Run	21%	8/37		1%	1/54		8%	58/709	
Program Coverage Total	59%	134/226		60%	217/364		55%	1962/3568	

6

Discussion

This chapter covers a discussion on certain aspects of funQ including its ease of use and syntax as well as an elaboration of the value of high-level language features. Further, the effects of deviating from QLambda and the insights gained from doing so are discussed. Finally, suggestions for future work on funQ is given.

6.1 Ease of Use

Ease of use was one of the purposes of this project and has permeated all aspects when implementing funQ. The aim was to make the language's syntax as easy as possible, provide good documentation, feedback to the user, and an interactive command line tool to allow the user to explore the language. The discussion about ease of use is extended to the practical use of high-level quantum languages.

6.1.1 Simple Syntax

The syntax of funQ was intended to follow QLambda and be as simple as possible. In cases where no syntactic conventions from QLambda were available, the syntax was modeled after Haskell. One such instance is for function declarations. The reason for imitating the syntactic conventions of a particular programming language is to make the external DSL more accessible to those familiar with that language (as described in section 3.1). The programs showed in the result section are a good indication of the level of complexity and can give the reader an own impression of how easy or difficult it is. The syntax is reasonably simple and should be easy to learn and understand, especially for Haskell programmers.

The Church-styled lambda term is the most significant syntactic difference between funQ and Haskell, which complicates the syntax of funQ a little. In funQ program files, Church-style lambdas and their complexity can be avoided using top-level functions instead of lambda terms. In the REPL, lambda terms are the only way to introduce new functions (except constant functions), making the Church-style more visible. However, this is primarily a change in notation and implementation of the type-checking algorithm and does not tremendously affect the language's syntax.

6.1.2 Abstraction, Reusability, and Extensibility

In funQ, complexity can be abstracted away in functions that allow code-reuse and often better readability. Furthermore, funQ can be used to write modules that may be shared and reused by other developers. The `bellMeasure` function in the teleport example of figure 5.6 exemplifies the increased abstraction level and the possibility of code reuse. The function replaces the application of a single CNOT gate, Hadamard gate, and two measurement operations. All these separate operations must still be performed, but if the user wants to reuse `bellMeasure`, only one function needs to be applied instead of four. Moreover, common operations like these could be packaged in standard modules, like Prelude for Haskell, which would likely speed up development for quantum programmers by avoiding re-creating utility functions.

The reuse of functions is constrained within a single file and not within an entire project. It is not possible to import funQ files into another file, only in the interactive mode of the command line tool. The inability to import files has not impacted this project since only programs in a single file have been implemented. Extending the language with the ability to import files would allow for increased code reuse. Moreover, improved separations of concerns would be possible by having multiple files, each with functions performing a specific task, rather than including all functions of a project in one file.

Introducing new gates in funQ is not possible without modifying the source code in several places, limiting the extensibility. With that said, it is unclear if such freedom to users is desirable since most quantum computers nevertheless come with a set of predefined gates. Therefore, the set of predefined gates in funQ should be sufficient, and restricting users to predefined gates may even be desirable.

6.1.3 Value of High-Level Language Features

The difference between funQ and some quantum languages is higher-level constructs, such as *let* and *if*. Implementing established quantum algorithms in funQ assessed the usefulness of higher-level languages for larger quantum programs.

The *if* expression in funQ is essential for classical control and allows replacing computationally expensive qubits with classical bits and control the program flow. For programs defined in higher-level languages, the *if* construct can be quite helpful, as classical control may be used instead of gates, thus, simplifying the code. In circuit-oriented languages, controlled gates are instead used for conditional control flow. If these gates cannot be classically controlled by a control bit, expensive qubits must be used instead. With this in mind, one could allow for classically controlled gates in a circuit-oriented language, which would mimic some of the constructs available with the *if* expression. However, the *if* expression in funQ is more powerful than those classically controlled gates would be. For instance, the *if* expression allows a compact way to think about and represent paths in an execution. Rather than something either happening or not, as in circuits, *if* expressions allow the programmer to execute separate expressions depending on a condition.

Most papers found describe quantum algorithms in terms of circuits (notably Shor’s) and are not implemented for higher-level languages [26]–[29]. However, the *let* construct made them somewhat easier to translate into funQ but was quite cumbersome for larger algorithms. Using *let* statements for each element of the circuit essentially simulates, quite inefficiently, an imperative language in funQ. A more functionally oriented way of implementing quantum algorithms in a high-level language is possible but would require more specialized constructs than funQ currently offers, primarily lists and custom datatypes. Recursion combined with lists or recursive datatypes could also provide a concise way of presenting sequenced or repetitive circuit elements, further strengthening the use cases of a higher-level language. Perhaps in the future, there will be higher-level algorithmic descriptions of implementations of quantum algorithms. These would be simpler to implement in funQ.

6.2 Implementing a Theoretical Language

The purpose of this project involved implementing funQ as closely as possible to the syntax, reduction- and typing rules of QLambda. Naturally, when implementing a theoretical language, some deviations will occur in the translation process. This section discusses the implications of the deviations from QLambda and the insights gained from these deviations.

6.2.1 Effects of Deviations from QLambda

The translation process of the typing rules and operational semantics from QLambda to funQ resulted in changes that can be seen as both a strength and a weakness. The evaluation rules were created to reflect the probabilistic, call-by-value approach of QLambda’s reduction rules. These modifications and additions is a strength in that it exactly defines how the funQ language is implemented. More transparency is also achieved by specifying funQ since the differences to QLambda become visible, allowing for nuanced discussions. However, the deviations could be seen as a weakness since more changes naturally weaken the relation of funQ and QLambda, effectively decreasing the possibility to extend claims and insights between funQ and QLambda. For example, consider the subtyping rule (! \otimes) defined in section 2.4.2. It is needed in the implementation to allow certain valid programs to typecheck. The fact that this rule is not stated in QLambda but was needed in the implementation may suggest a misinterpretation of QLambda or a fault in the implementation of the type checker.

6.2.2 Insights from the Implementation

A deviation from QLambda regards the definition of a program, which in QLambda is a lambda term. In funQ, however, a program is defined as a list of functions. The successful implementation of, for example, quantum teleportation and Grover's algorithms shows the benefits of including functions in quantum programming languages. These benefits include increased readability, abstraction, and code reuse. Only allowing the creation of a lambda term to define a whole program is limiting since it would make the creation of larger programs complicated, which suggests that future functional quantum programming languages should include the ability to define functions.

Another insight gained when implementing the interpreter for funQ was that separation of classical control from quantum computation was suitable. The evaluation rules in section 2.15 reflect this distinction in the program state, with its environment γ and quantum state Q . However, the operational semantics of QLambda make no such distinction. The separation simplified the implementation of an interpreter and is recommended for a future implementation of a similar quantum programming language with classical control.

6.3 Future Work

There is some future work that could improve funQ. This work includes connecting it to an external quantum computer simulator or a real quantum computer, inferring linear types without type annotations, and extending the syntax.

6.3.1 Improve the Computation Speed

The efficiency of a quantum computer simulator can vary, and the simulator used in funQ could be improved in this regard. A simple improvement of the simulator could be to remove measured qubits from the quantum state vector. Similar to QLambda, measured qubits remain in the quantum state vector, though not affecting the state other than using up memory. Instead, collapsed qubits can be factored out and removed from the state vector making the simulation more efficient. Another improvement of the simulator could be to use matrix libraries from more fast-performing languages such as C, which might also support running matrix operations on the GPU (graphics processing unit).

Another possibility to improve the performance of funQ is to connect it to a, possibly more efficient, external quantum computer simulator. With an approved capacity and speed of running quantum programs, more qubits can be used simultaneously.

Taking it one step further and connecting funQ with an actual quantum device would be interesting to see if the language works in practice and what problems may arise. This could be done in the interpreter by communicating with a quantum web service that supports quantum actions to be made via an application programming interface but would require some work.

6.3.2 Hindley-Milner Type Inference for Linear Types

As discussed, funQ uses Church-style lambdas while QLambda uses Curry-style lambdas. Curry-style lambdas were originally preferred to keep the syntax of funQ more similar to QLambda. A sophisticated type inference algorithm is needed to infer types of Curry-styled lambdas, such as the algorithm W by Roger Hindley and Robin Milner [30] [31]. Quite some effort was put into extending algorithm W that could handle type inference with linear types. However, after a promising but failed attempt to implement this algorithm with linear types, it was discovered that the Hindley-Milner algorithm for a type system with subtypes is still an open problem. Therefore, it was decided that this problem was probably not going to be solved within the time frame of this thesis and instead could be possible future work.

A successful implementation of the Hindley-Milner algorithm with linear types would allow funQ to:

- omit type signatures for functions
- have generic functions with type variables in the signature
- include Curry-style lambdas in the syntax.

6.3.3 Adding Language Features

Language features and syntax sugar that exist in many other languages could be added to funQ. This includes custom data types, importing functions from other files, natural numbers, and enhanced error messages with specific lines of the error. Also, by adding more recursive data types, such as natural numbers, recursion might be more useful. Another helpful feature that could be added is a list data type for a more flexible container type than tuples.

7

Conclusion

The functional quantum programming language funQ has been successfully implemented and can be seen as an implemented version of QLambda. Following the conventions of QLambda, it is implemented with call-by-value semantics and a linear type system. Additionally, a command line interface was created to make the language more accessible. Several well-known quantum algorithms have been implemented in funQ with successful results, such as quantum teleportation, Deutsch-Jozsa, Grover's, and a specialized version of Shor's algorithm. The implementation of these algorithms showed that it is straightforward to write quantum programs in funQ.

The high-level features of the language increase the abstraction level of a quantum program from quantum circuits and give more expressiveness to the programmer. For example, if expressions are useful for conditional logic without using extra qubits needed in pure quantum circuits. Also, functions allow abstraction of concrete operations and reuse common pieces of logic. However, implementing large quantum algorithms, such as Shor's, showed to be somewhat cumbersome. All available descriptions of the algorithm were in terms of circuits, and the high-level features available in funQ were not generally helpful when writing this algorithm. Adding more data types to funQ to make more use of recursion should aid in constructing these larger circuits, perhaps making funQ more useful than circuit languages.

References

- [1] P. Selinger and B. Valiron, “A Lambda Calculus for Quantum Computation with Classical Control,” in *Typed Lambda Calculi and Applications*, D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, and P. Urzyczyn, Eds., vol. 3461, Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 354–368, ISBN: 9783540255932 9783540320142. DOI: 10.1007/11417170_26. [Online]. Available: http://link.springer.com/10.1007/11417170_26 (visited on 02/05/2021).
- [2] P. Benioff, “The computer as a physical system: A microscopic quantum mechanical Hamiltonian model of computers as represented by Turing machines,” *Journal of Statistical Physics*, vol. 22, pp. 563–591, 1980.
- [3] R. P. Feynman, “Simulating physics with computers,” *International Journal of Theoretical Physics*, vol. 21, pp. 467–488, 1982. DOI: <https://doi.org/10.1007/BF02650179>.
- [4] A. Wigderson, “P, N P and mathematics – a computational complexity perspective,” in *Proceedings of the International Congress of Mathematicians 2006 (Madrid)*, EMS Publishing House, 2007, pp. 667–712.
- [5] J. Preskill, *Lecture notes for Physics 229: Quantum Information and Computation*. California Institute of Technology, Sep. 1998. [Online]. Available: https://www.lorentz.leidenuniv.nl/quantumcomputers/literature/preskill_1_to_6.pdf (visited on 02/11/2021).
- [6] P. W. Shor, “Algorithms for quantum computation: Discrete logarithms and factoring,” in *Proceedings 35th Annual Symposium on Foundations of Computer Science*, 1994, pp. 124–134.
- [7] E. Bernstein and U. Vazirani, “Quantum Complexity Theory,” *SIAM Journal of Computing*, vol. 26, no. 5, pp. 1411–1473, Oct. 1997. DOI: <https://doi.org/10.1137/S0097539796300921>.
- [8] F. e. a. Arute, “Quantum supremacy using a programmable superconducting processor,” *Nature*, vol. 574, no. 7779, pp. 505–510, Oct. 2019, ISSN: 1476-4687. DOI: 10.1038/s41586-019-1666-5. [Online]. Available: <https://www.nature.com/articles/s41586-019-1666-5> (visited on 04/27/2021).
- [9] J. Grattage, “An Overview of QML With a Concrete Implementation in Haskell,” en, *Electronic Notes in Theoretical Computer Science*, Proceedings of the Joint 5th International Workshop on Quantum Physics and Logic and 4th Workshop on Developments in Computational Models (QPL/DCM 2008), vol. 270, no. 1, pp. 165–174, Feb. 2011, ISSN: 1571-0661. DOI: 10.1016/j.entcs.2011.01.015. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S1571066111000168> (visited on 02/19/2021).
- [10] B. Bichsel, M. Baader, T. Gehr, and M. Vechev, “Silq: A high-level quantum language with safe uncomputation and intuitive semantics,” *PLDI 2020: Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020. DOI: 10.1145/3385412.3386007. [Online]. Available: <https://files.sri.inf.ethz.ch/website/papers/pldi20-silq.pdf>.
- [11] S. Bettelli, T. Calarco, and L. Serafini, “Toward an architecture for quantum programming,” en, *The European Physical Journal D - Atomic, Molecular, Optical and Plasma Physics*, vol. 25, no. 2, pp. 181–200, Aug. 2003, ISSN: 1434-6079. DOI: 10.1140/epjd/

- e2003-00242-2. [Online]. Available: <https://doi.org/10.1140/epjd/e2003-00242-2> (visited on 02/11/2021).
- [12] D. J. Griffiths, *Introduction to quantum mechanics*, 2nd ed. Upper Saddle River, NJ: Pearson Prentice Hall, 2005, ch. "12.3 The no-clone theorem", ISBN: 9780131118928.
 - [13] B. C. Pierce, *Types and programming languages*. Cambridge, Mass: MIT Press, 2002, 623 pp., ISBN: 9780262162098.
 - [14] A. Ranta and M. Forsberg, *Implementing programming languages: an introduction to compilers and interpreters*, eng, ser. Texts in computing 16. London: College Publ, 2012, OCLC: 820917145, ISBN: 9781848900646.
 - [15] J. Daintith and E. Wright, *Backus normal form*, 2008. DOI: 10.1093/acref/9780199234004.013.0306. [Online]. Available: <https://www.oxfordreference.com/view/10.1093/acref/9780199234004.001.0001/acref-9780199234004-e-306>.
 - [16] J.-Y. Girard, *Linear logic*, 1987. [Online]. Available: [https://doi.org/10.1016/0304-3975\(87\)90045-4](https://doi.org/10.1016/0304-3975(87)90045-4).
 - [17] P. Wadler, "Linear types can change the world!" In *Programming Concepts and Methods*, 1990.
 - [18] M. Fowler, *Domain-Specific Languages*. Sep. 2010, ISBN: 9780131392809.
 - [19] A. Abel, J. Duregård, and K. Angelov, *The bnf converter*, version 2.8.3, Feb. 10, 2020. [Online]. Available: <https://bnfc.digitalgrammars.com/>.
 - [20] T. J. B. Community, *Quantum Protocols and Quantum Algorithms*, en. [Online]. Available: <https://community.qiskit.org/textbook/ch-algorithms/> (visited on 02/11/2021).
 - [21] R. Cleve, A. Ekert, C. Macchiavello, and M. Mosca, "Quantum algorithms revisited," *Proceedings of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences*, vol. 454, no. 1969, pp. 339–354, Jan. 1998, ISSN: 1471-2946. DOI: 10.1098/rspa.1998.0164. [Online]. Available: <http://dx.doi.org/10.1098/rspa.1998.0164>.
 - [22] C. Figgatt, D. Maslov, K. A. Landsman, N. M. Linke, S. Debnath, and C. Monroe, "Complete 3-qubit grover search on a programmable quantum computer," *Nature Communications*, vol. 8, no. 1, Dec. 2017. DOI: 10.1038/s41467-017-01904-7. [Online]. Available: <https://doi.org/10.1038/s41467-017-01904-7>.
 - [23] A. Asfaw, A. Corcoles, L. Bello, Y. Ben-Haim, M. Bozzo-Rey, S. Bravyi, N. Bronn, L. Capelluto, A. C. Vazquez, J. Ceroni, R. Chen, A. Frisch, J. Gambetta, S. Garion, L. Gil, S. D. L. P. Gonzalez, F. Harkins, T. Imamichi, H. Kang, A. h. Karamlou, R. Lored, D. McKay, A. Mezzacapo, Z. Mineev, R. Movassagh, G. Nannicini, P. Nation, A. Phan, M. Pistoia, A. Rattew, J. Schaefer, J. Shabani, J. Smolin, J. Stenger, K. Temme, M. Tod, S. Wood, and J. Wootton. (2020). "Learn quantum computation using qiskit," [Online]. Available: <http://community.qiskit.org/textbook>.
 - [24] L. M. K. Vandersypen, M. Steffen, G. Breyta, C. S. Yannoni, M. H. Sherwood, and I. L. Chuang, "Experimental realization of shor's quantum factoring algorithm using nuclear magnetic resonance," *Nature*, vol. 414, no. 6866, pp. 883–887, Dec. 2001, ISSN: 1476-4687. DOI: 10.1038/414883a. [Online]. Available: <http://dx.doi.org/10.1038/414883a>.
 - [25] P. J. Coles, S. J. Eidenbenz, S. Pakin, A. Adedoyin, J. Ambrosiano, P. M. Anisimov, W. Casper, G. Chennupati, C. Coffrin, H. Djidjev, D. Gunter, S. Karra, N. Lemons, S. Lin, A. Y. Lokhov, A. Malyzhenkov, D. D. L. Mascarenas, S. M. Mniszewski, B. Nadiga, D. O'Malley, D. Oyen, L. Prasad, R. Roberts, P. Romero, N. Santhi, N. Sinitsyn, P. Swart, M. Vuffray, J. Wendelberger, B. Yoon, R. J. Zamora, and W. Zhu, "Quantum algorithm

- implementations for beginners,” *CoRR*, vol. abs/1804.03719, 2018. arXiv: 1804.03719. [Online]. Available: <http://arxiv.org/abs/1804.03719>.
- [26] S. Beauregard, *Circuit for shor’s algorithm using $2n+3$ qubits*, Feb. 2003. [Online]. Available: <https://arxiv.org/abs/quant-ph/0205095>.
 - [27] S. Parker and M. B. Plenio, “Efficient factorization with a single pure qubit and logn-mixed qubits,” *Physical Review Letters*, vol. 85, no. 14, pp. 3049–3052, Oct. 2000, ISSN: 1079-7114. DOI: 10.1103/physrevlett.85.3049. [Online]. Available: <http://dx.doi.org/10.1103/PhysRevLett.85.3049>.
 - [28] A. Pavlidis and D. Gizopoulos, *Fast quantum modular exponentiation architecture for shor’s factorization algorithm*, 2013. arXiv: 1207.0511 [quant-ph].
 - [29] C. Zalka, “Fast versions of shor’s quantum factoring algorithm,” *arXiv: Quantum Physics*, 1998. [Online]. Available: <https://arxiv.org/abs/quant-ph/9806084>.
 - [30] R. Hindley, “The principal type-scheme of an object in combinatory logic,” *Transactions of the American Mathematical Society*, vol. 146, pp. 29–60, 1969, ISSN: 00029947. [Online]. Available: <http://www.jstor.org/stable/1995158>.
 - [31] R. Milner, “A theory of type polymorphism in programming,” *Journal of Computer and System Sciences*, vol. 17, pp. 348–375, 1978.
 - [32] M. A. Nielsen and I. L. Chuang, *Quantum computation and quantum information*. Cambridge University Press, 2019.

A

Quantum gates

Table A.1 shows the complete set of gates that are included in funQ. The left column shows the syntax to invoke the gate, and the right the full name of the gate. Detailed descriptions of the gates can be found in [32].

Table A.1: Gates included in set of gates in funQ.

H	Hadamard
X	Pauli-X
Y	Pauli-Y
Z	Pauli-Z
I	Identity
S	S gate
T	T gate
CNOT	CNOT
TOF	Toffoli
SWP	Swap
FRDK	Fredkin
QFTn	Quantum Fourier transform
QFTIn	Inverse quantum Fourier transform
CRn	Controlled rotation/phase shift
CRIn	Inverse controlled rotation
CRRn	Controlled controlled rotation
CRRIn	Inverse controlled controlled rotation

B

Grammar

This document was automatically generated by the *BNF-Converter*. It was generated together with the lexer, the parser, and the abstract syntax module, which guarantees that the document matches with the implementation of the language (provided no hand-hacking has taken place).

The lexical structure of Parser

Literals

Integer literals *Integer* are nonempty sequences of digits.

FunVar literals are recognized by the regular expression ‘lower ([”_”] | digit | letter)* ’ ’ * ’ : ’ ‘

Var literals are recognized by the regular expression ‘lower ([”_”] | digit | letter)* ‘

GateIdent literals are recognized by the regular expression ‘upper (digit | upper)* ‘

Lambda literals are recognized by the regular expression ‘ ’ \ ‘

Reserved words and symbols

The set of reserved words is the set of terminals appearing in the grammar. Those reserved words that consist of non-letter characters are called symbols, and they are treated in a different way from those that are similar to identifiers. The lexer follows rules familiar from languages like Haskell, C, and Java, including longest match and spacing conventions.

The reserved words used in Parser are the following:

Bit	CNOT	FREDKIN	H
I	QBit	S	SWAP
T	TOFFOLI	X	Y
Z	else	if	in
let	then		

The symbols used in Parser are the following:

*	(,)
=	.	\$!
><	-o		

Comments

Single-line comments begin with `-`. Multiple-line comments are enclosed with `{-` and `-}`.

The syntactic structure of Parser

Non-terminals are enclosed between `<` and `>`. The symbols `->` (production), `|` (union) and **eps** (empty rule) belong to the BNF notation. All other symbols are terminals.

<i>Program</i>	->	<i>[FunDec]</i>
<i>Term3</i>	->	<i>Var</i>
		<i>Bit</i>
		<i>Gate</i>
		<i>Tup</i>
		*
		(<i>Term</i>)
<i>Term2</i>	->	<i>Term2 Term3</i>
		<i>Term3</i>
<i>Term1</i>	->	if <i>Term</i> then <i>Term</i> else <i>Term</i>
		let (<i>LetVar</i> , <i>[LetVar]</i>) = <i>Term</i> in <i>Term</i>
		<i>Lambda FunVar Type . Term</i>
		<i>Term2 \$ Term1</i>
		<i>Term2</i>
<i>Term</i>	->	<i>Term1</i>
<i>LetVar</i>	->	<i>Var</i>
<i>[LetVar]</i>	->	<i>LetVar</i>
		<i>LetVar</i> , <i>[LetVar]</i>
<i>Tup</i>	->	(<i>Term</i> , <i>[Term]</i>)
<i>[Term]</i>	->	<i>Term</i>
		<i>Term</i> , <i>[Term]</i>
<i>Bit</i>	->	<i>Integer</i>
<i>FunDec</i>	->	<i>FunVar Type Function</i>
<i>[FunDec]</i>	->	eps
		<i>FunDec [FunDec]</i>
<i>Function</i>	->	<i>Var [Arg] = Term</i>
<i>Arg</i>	->	<i>Var</i>
<i>[Arg]</i>	->	eps
		<i>Arg [Arg]</i>
<i>Type2</i>	->	Bit
		QBit
		T
		! <i>Type2</i>
		(<i>Type</i>)
<i>Type1</i>	->	<i>Type2</i> >< <i>Type1</i>
		<i>Type2</i> -o <i>Type1</i>
		<i>Type2</i>
<i>Type</i>	->	<i>Type1</i>
<i>Gate</i>	->	H
		X
		Y
		Z
		I
		S
		T
		CNOT
		TOFFOLI
		SWAP
		FREDKIN
		<i>GateIdent</i>

C

Error types

The semantic errors that were used in the semantic analyzer.

```
data SemanticError
  = FunNameMismatch String -- ^ Definition and function signature names must match
  | DuplicateFunction String -- ^ Function declared more than once
  | UnknownGate String      -- ^ A gate that is not defined in the language was used
  | InvalidBit String       -- ^ Bit must be 0 or 1
  | TooManyArguments String -- ^ Too many arguments in function definition
```

The type errors that were used in the type checker.

```
data ErrorTypes
  = NotFunction Type      -- ^ A type was expected to be a function but was not.
  | Mismatch Type Type    -- ^ Expected a type but found another type.
  | NotProduct Type       -- ^ A type was expected to be a product but was not.
  | NotLinearTop String    -- ^ A function that is linear used many times.
  | NotLinearTerm Term Type -- ^ A term that breaks a linearity constraint
  | NoCommonSuper Type Type -- ^ No common supertype was found
  | NotInScope String     -- ^ A function was not in scope.
  deriving Eq

data TypeError = TError String ErrorTypes
  deriving Eq
```


D

Order-finding algorithm

The complete algorithm is available at <https://github.com/NicklasBoto/funQ/blob/main/test/interpreter-test-suite/qft-adder3.fq>

```
orderfind : (QBit >< QBit >< QBit >< QBit >< QBit >< QBit)
-o !(Bit >< Bit >< Bit)
-o !(Bit >< Bit >< Bit)
-o (QBit >< QBit >< QBit)
-o (QBit >< QBit >< QBit >< QBit >< QBit >< QBit >< QBit >< QBit)
>< !(Bit >< Bit >< Bit >< Bit >< Bit >< Bit >< Bit)

orderfind inA inB inN inX =
  let (x2,x1,x0,v,c,a3,a2,a1,a0) = cUa inA inB inN inX in
  let (v,m0) = (v, meas (H c)) in
  let (v,c) = (v, H (new 0)) in

  let (x2,x1,x0,v,c,a3,a2,a1,a0) = cUa (v,c,a3,a2,a1,a0) inB inN (x2,x1,x0) in
  let (v,m1) = (v, meas (cR Z m0 c)) in
  let (v,c) = (v, H (new 0)) in

  let (x2,x1,x0,v,c,a3,a2,a1,a0) = cUa (v,c,a3,a2,a1,a0) inB inN (x2,x1,x0) in
  let (v,m2) = (v, meas (cR1 Z (m0,m1) c)) in
  let (v,c) = (v, H (new 0)) in

  let (x2,x1,x0,v,c,a3,a2,a1,a0) = cUa (v,c,a3,a2,a1,a0) inB inN (x2,x1,x0) in
  let (v,m3) = (v, meas (cR2 Z (m0,m1,m2) c)) in
  let (v,c) = (v, H (new 0)) in

  let (x2,x1,x0,v,c,a3,a2,a1,a0) = cUa (v,c,a3,a2,a1,a0) inB inN (x2,x1,x0) in
  let (v,m4) = (v, meas (cR3 Z (m0,m1,m2,m3) c)) in
  let (v,c) = (v, H (new 0)) in

  let (x2,x1,x0,v,c,a3,a2,a1,a0) = cUa (v,c,a3,a2,a1,a0) inB inN (x2,x1,x0) in
  let (v,m5) = (v, meas (cR4 Z (m0,m1,m2,m3,m4) c)) in
  let (v,c) = (v, new 0) in
  (v, meas c, x2,x1,x0,a3,a2,a1,a0,m4,m5,m3,m2,m1,m0)
```

The cUa function is the periodic function that the order-finding part operates on. The classically rotation gates are denoted cR_i , where $i = n + 1$ is the argument.



UNIVERSITY OF
GOTHENBURG



CHALMERS
UNIVERSITY OF TECHNOLOGY