

**Tallinn University of Technology**  
Department of Computer Engineering

ANALYSIS OF PROGRAMMING LANGUAGES (IAG0050)

# QUIPPER – A QUANTUM PROGRAMMING LANGUAGE

Student: Keijo Kuusmik (122007IASM)

Lecturer: Vladimir Viies

Tallinn 2013

## CONTENTS

Introduction .....	3
Background .....	3
Quantum computing .....	3
Haskell .....	3
Quipper .....	4
Essential characteristics .....	5
Well defined syntactic and semantic definition .....	5
Reliability .....	5
Usability .....	6
Scalability .....	6
Machine independence .....	7
Desirable characteristics .....	7
Provability .....	7
Generality .....	7
Consistency with commonly used notations .....	7
Extensibility .....	7
Summary .....	8
Works Cited .....	9

## INTRODUCTION

## BACKGROUND

### QUANTUM COMPUTING

Quantum computing refers to computers utilizing the effects of quantum physics in order to implement its computation. At this moment there are computational problems for which an efficient implementation is available using model of quantum computing (by efficient I mean polynomial complexity) but not using model of classical computing (i.e. probabilistic Turing machine). This provides motivation for developing quantum computing.

For quantum computing elementary unit of information is represented mathematically by normalized vector in complex vector space (Hilbert space). Quantum bit or qubit for short is represented as sum of two basis vectors  $\alpha|0\rangle + \beta|1\rangle$  where  $\alpha, \beta \in \mathbb{C}$  and  $|\alpha|^2 + |\beta|^2 = 1$  representing the fact that when measured qubit will be one of two states (i.e. probabilities need to add to 1).

Any operation that can be performed on qubits (called quantum gate when referring to circuit model of quantum computation) is mathematically represented by unitary operator (i.e. unitary matrix). Requirement for linear operator to be unitary comes from the fact that the state of qubits needs to remain normalized (i.e. probabilities need to add up to 1) after applying the operator.

Quantum circuits also need to be reversible meaning that it is possible the reverse the computation and get initial data back. For example classical NOT gate is example of reversible gate because by observing the output one can deduce input. But classical AND gate on the other hand is not reversible since by only getting the output it is impossible to infer what the input was. It is always possible to convert gate/circuit to reversible one.

### HASKELL

Haskell is the host language for Quipper. This means Quipper utilizes Haskell's language features to implement its own functionality. Haskell itself is functional programming language.

Haskell more important features include:

- 1) Functional programming language
- 2) Pure VS tainted function separation
  - a. Pure functions are those that don't have side-effects. This means that calling a function with the same parameters will always yield same result (i.e. referential transparency).
  - b. Tainted functions can have side-effects. This basically means that these functions have implicit global input besides the explicit input parameters. For example think of function that reads a file. It will probably take as input the file name but its output will depend on the current contents of that file. In this scenario contents of that file would be the implicit global input and the file name would be the explicit input. It is clear in this case that just by looking at the explicit input one cannot know the output.

- c. Tainted functions are encapsulated by IO monad.
- 3) Strongly typed – types are checked at compile time
- 4) Type inference – it is not needed to provide type declaration for functions Haskell can infer that by itself. Although it is allowed to include type declaration (for readability).
- 5) Generics – functions can be described in terms of generic type parameter.

## QUIPPER

Quippers mission statement is the following: “accurately estimate and reduce the computational resources required to implement quantum algorithms on a realistic quantum computer” [1]

Quipper is quantum programming language meaning that it is intended for writing algorithms for quantum computers. It is implemented as an embedded programming language with Haskell as host language. Being a embedded programming language means using the language features of host language to achieve design goals. Quipper is basically defined by data types, combinators and library of functions combined with preferred way of programming (i.e. programming idiom). It is declarative language with monadic operational semantics (defines Circ monad for quantum circuits). As there is no physically built general purpose quantum computer Quipper cannot target any real machine architecture. Instead it's taking a high level description of quantum algorithm and producing a circuit described by that algorithm. This enables the main design goal of resource estimation along with possibility to simulate operation of generated circuit.

Quipper has 3 distinct phases:

- 1) Haskell program compilation – since Quipper uses Haskell as host language then first the program needs to be compiled using Haskell compiler.
- 2) Circuit generation time – program compiled in previous step can now produce quantum circuit. This stage takes as input circuit parameters (e.g. number of qubits). Output of this step could be executable circuit given that there would exist a quantum computer to run it on. But as one of the stated goals of Quipper is to estimate resources needed for implementing algorithms then this step is also able to give such estimations (number of gates, depth of circuit etc.). Also Quippers supports printing generated circuit to document (e.g. pdf).
- 3) Circuit execution / simulation - In this step it is possible to simulate generated circuit on classical computer. There are 3 modes of simulation available [1] :
  - a. Classical simulation - efficiently simulates classical circuits.
  - b. Stabilizer simulation - efficiently simulates Clifford group circuits
  - c. Quantum simulation - simulates any circuit (with exponential overhead).

It is noteworthy also that Quipper models quantum computer as being controlled by classical computer much like a co-processor. This is incorporated into Quipper by defining both quantum and classical bits.

## ESSENTIAL CHARACTERISTICS

### WELL DEFINED SYNTACTIC AND SEMANTIC DEFINITION

One of the key design requirements for Quipper was high level and concise representation of quantum algorithms.

```
mycirc :: Qubit -> Qubit -> Circ (Qubit, Qubit)
mycirc a b = do
  a <- hadamard a
  b <- hadamard b
  (a,b) <- controlled_not a b
  return (a,b)
```

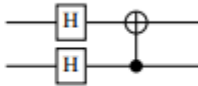


Figure 1 – Quipper - simple example function [2]

And although Quipper achieves its task of representing quantum circuits in a high level representation there is nothing inside the compiler itself (Haskell compiler) which assures that the programmer writes programs that make sense in the realm of Quipper. The compiler is able to verify programs compliance with Haskell's syntactic requirements but for a program to be a valid and useful Quipper program one should use data types, combinators, functions and preferred way of programming (programming idiom) provided by Quipper.

For a language like Quipper being embedded into Haskell is design tradeoff. Implementing a full compiler means considerable overhead. Using another language to achieve one's goals can be less expensive in terms of implementation time. This can certainly be justified for experimental language like Quipper whose main goal is scientific research.

### RELIABILITY

First component of reliability is compiler's ability to catch mistakes at compile time. It is obvious that not all mistakes can be caught at that time. This is especially true for semantic mistakes that might not be caught at all. Nevertheless it is still important to validate programs syntactic compliance and if at all possible enforce correct semantics also. As mentioned earlier Quipper is embedded language and programmer can write programs using the host language that don't follow Quipper's logic. But of course it is possible for Quipper's own functions to enforce correct usage by restricting usable types.

Second component of reliability is language's runtime ability to catch and handle error conditions. Quipper also makes use of runtime checks. Since Haskell type system lacks linear types and dependent types some properties that could be checked at compile time need to be checked in run time. Linear typed values are values (objects) that can only be used once. This correlates well with semantics of operating with qubits. Namely, since qubits cannot be copied (no cloning theorem of QM) and interactions change the states of qubits, then particular qubit state can only be used once. Such properties are checked by Quipper at runtime (or more correctly at circuit generation time).

## USABILITY

By usability I mean writeability and readability i.e. how simple it is to program in given language. This involves factors like conciseness of language so programmer could say exactly what he wants using the tools provided by the language. Also abstraction facilities play a major role in making a programming language usable.

Quipper addresses usability by first of all providing clear syntax and abstraction facilities. Let's review somewhat complex piece of code written in Quipper to clarify this matter:

```
qft' :: [Qubit] -> Circ [Qubit]
qft' [] = return []
qft' [x] = do
  hadamard x
  return [x]
qft' (x:xs) = do
  xs' <- qft' xs
  xs'' <- rotations x xs' (length xs')
  x' <- hadamard x
  return (x':xs'')
where
  rotations :: Qubit -> [Qubit] -> Int -> Circ [Qubit]
  rotations _ [] _ = return []
  rotations c (q:qs) n = do
    qs' <- rotations c qs n
    let m = ((n + 1) - length qs)
    q' <- rGate m q 'controlled' c
    return (q':qs')
```

Figure 2 - Quantum Fourier Transform [1]

First of all note that QFT (Quantum Fourier Transform) is defined recursive function. Also application of gate or sub-circuit is denoted mostly by single line with resulting qubit being in the left hand side.

## SCALABILITY

Scalability is especially important for Quipper as it aims to describe large scale quantum algorithms. Some of the implemented algorithms provided by Quipper have gate count in the trillions so gate by gate description is out of question. To tackle this Quipper programmer can define functions to represent sub-circuits and define them as boxed circuits (graphically represented by single box). Also generic functions allow defining circuit families (i.e. not specifying for how many qubits given circuit is for).

## MACHINE INDEPENDENCE

Quipper aims to be general enough to provide usable results for various possible architectures for quantum computers. Even further Quipper aims to be a tool which could help explore plausible architectures for future quantum computers by more clearly specifying requirements for implementing various algorithms.

## DESIRABLE CHARACTERISTICS

### PROVABILITY

Provability is strongly connected with language syntax & semantics and reliability. The stricter a language is the easier it is to prove some properties for it. For Quipper automatic verification is not one of the design goals. But nevertheless it is important to be able to check/prove if algorithm is working as intended. For this purpose Quipper has at the moment simulator which can simulate the generated quantum circuit.

### GENERALITY

Quippers is targeting general purpose quantum computer which would be controlled by classical computer. So being general enough to allow interfacing classical and quantum computer is something desirable for Quipper. For this purpose Quipper has data types representing both classical and quantum bits. Also it is possible to parameterize the generated circuits on classical data.

### CONSISTENCY WITH COMMONLY USED NOTATIONS

Quipper aims to provide possibility to represent quantum algorithms in programming language as close as possible to the form they are usually described in literature. This is first and foremost a challenge for abstracting facilities as quantum algorithms aren't usually described in terms of single gates but rather in terms of circuit level transformations. For this Quipper supports circuit level transformations (e.g. reversing circuit).

### EXTENSIBILITY

Quipper supports user defined data types and generic functions. User defined data types are based on the base type `QCData`. Generic functions on the other hand allow writing circuits that don't depend on the number of qubits they receive as inputs, effectively describing a whole family of circuits.

## SUMMARY

Quipper is a quantum programming language for designing quantum algorithms and estimating the circuit level resources needed for their implementation. Quipper is implemented as embedded language with Haskell as host. As such Quipper defines data types, combinators, library of functions and programming idiom to enable implementing quantum algorithms. Essential features of Quipper are clear syntax & semantics for describing quantum algorithms, scalability to enable do describe large circuits, reliability to assure that valid algorithms would compile while invalid wouldn't, usability to enable programmer to actually write and understand complex code.



## WORKS CITED

- [1] A. Green S, P. L. Lumsdaine, N. J. Ross, P. Selinger and B. Valiron, *An Introduction to Quantum Programming in Quipper*.
- [2] A. Green S, P. L. Lumsdaine, N. J. Ross, P. Selinger and B. Valiron, *Quipper: A Scalable Quantum Programming Language*.