Simulation of Quantum Computations in Lisp

Brecht Desmet, Ellie D'Hondt, Pascal Costanza and Theo D'Hondt
Programming Technology Lab – Vrije Universiteit Brussel
Pleinlaan 2 – 1050 Brussels, Belgium
$\{$*bdesmet,eldhondt,pascal.costanza,tjdhondt*$\}$*@vub.ac.be*

May 22, 2006

Abstract

This paper introduces QLisp, a compact and expressive language extension of Lisp that simulates quantum computations. QLisp is an open experimental and educational platform because it translates the quantum computations into the underlying mathematical formalism and offers the flexibility to extend the postulates of quantum mechanics. Although the complexity degree of quantum mechanics is inherently exponential, QLisp includes some optimizations that prune in both space and time.

1 Introduction

Quantum computation is a relatively young interdisciplinary research field that arose from an intersection of quantum mechanics [Dir47], mathematics [vN55] and computer science [Deu85]. The interest in this research topic received a profound boost when Shor [Sho96] showed how RSA encryption [RSA78] can be decrypted efficiently using a quantum computer. Unfortunately, we are still far from the implementation of a general-purpose quantum computer. The absence of such a machine motivates researchers to build and exploit quantum similators.

In this paper, we introduce QLisp, a quantum simulator integrated in the Common Lisp programming language. The main characteristics of QLisp are as follows. First, the simulator has the flavour of a model because all quantum computations are translated to the underlying mathematical formalism. Second, QLisp supports the flexibility to observe and modify quantum computations at runtime, something which is not allowed by the postulates of quantum mechanics. By thinking in terms of mathematical linear structures, the code becomes easier to read and comprehend. Next, we took advantage of the many powerful abstraction techniques of Lisp which resulted in a very compact and expressive language extension to describe quantum computations. Therefore, it is our belief that QLisp has interesting educational opportunities. Finally, QLisp has some optimizations included that prune in the exponential complexity degree of quantum mechanics.

This paper is organised as follows. Before we motivate some major design decisions of QLisp in Section 3, we provide a short introduction of quantum computations in Section 2. Next, Section 4 highlights the architecture and efficiency considerations of QLisp. We finish this paper with example code of some popular quantum algoritms in Section 5.

2 Quantum computation in a nutshell

Moore's law indicates that the speed of processors doubles every 18 months because of the miniaturisation of processor units. When these units reach the atomic level, the postulates of quantum mechanics become applicable instead of the classical laws of physics. The non-intuitive behaviour and properties of atoms constitute the basis for a new kind of computation called quantum computation, a subdomain of quantum mechanics. The latter describes the behaviour of light and matter at the atomic level. The important advantage of this alternative way of computing is the fact that some problems, like prime factorization, can be solved efficiently. So far, this is not possible with classical computations. This section provides a short introduction of the basic concepts of quantum computation required for this paper. For a detailed description of quantum computing, we refer to the book of Nielsen & Chuang [NC00].

The elementary building block of a classical Turing machine is a bit. Its state is deterministic (0 or 1) and always observable. Consequently, it is possible to copy the state of a bit. Furthermore, classical computations are not always reversible, like e.g. a conjunction that evaluated to false. This might seem straightforward, but this is not the case in quantum computation.

The possible states of a *qubit*, the quantum analogue of a bit, are $|0\rangle$ and $|1\rangle$ which correspond to the states of a classical bit. The Dirac notation '$|\rangle$' is the standard notation for states in quantum mechanics. The main difference between bits and qubits is that qubits can be in different states at the same time. In mathematical terms, an arbitrary qubit $|\psi\rangle$ is expressed by means of a linear combination, i.e. a superposition, of the basis states $|0\rangle$ and $|1\rangle$, as given in Equation 1. The variables $a, b \in \mathbb{C}$ used in this equation are called *amplitudes*.

$$|\psi\rangle = a|0\rangle + b|1\rangle = a\begin{bmatrix} 1 \\ 0 \end{bmatrix} + b\begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} a \\ b \end{bmatrix} \tag{1}$$

The state of qubits is not observable without disturbance in the sense that we cannot examine the amplitudes a and b directly. Instead, quantum mechanics only allows us to measure a qubit such that its state irrevocably collapses in one of its basis states. For instance, if we measure $|\psi\rangle$, the result will be either 0 with probability $|a|^2$ or 1 with probability $|b|^2$. Furthermore, it is not possible to clone qubits because of the no-cloning theorem [WZ82].

Single qubits can be composed to an n-qubit by means of the tensor product $\otimes$. Such multiple qubits consist of 2^n superposition terms of the basis states $|0\rangle$ and $|1\rangle$. This means that the representation of an n-qubit on a classical computer results in an exponential growth of space complexity. For instance, the composition $|\phi\rangle$ of the qubits $|\psi_1\rangle = a|0\rangle + b|1\rangle$ and $|\psi_2\rangle = c|0\rangle + d|1\rangle$ is presented in Equation 2. The equivalent decimal notation is presented in Equation 3.

$$\begin{aligned} |\phi\rangle &= |\psi_1\rangle \otimes |\psi_2\rangle \\ &= ac|00\rangle + ad|01\rangle + bc|10\rangle + bd|11\rangle \quad (2) \\ &= ac|0\rangle + ad|1\rangle + bc|2\rangle + bd|3\rangle \quad (3) \end{aligned}$$

Quantum computations can be represented graphically using *quantum circuits*. Figure 1 presents an example of a quantum circuit that implements the famous Deutsch-Jozsa algorithm [DJ92]. In this section, we only concentrate on the semantics of the

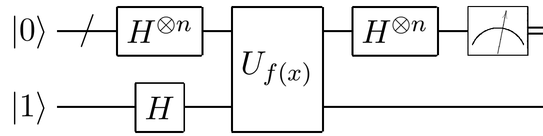

Figure 1: Quantum circuit of Deutsch-Jozsa algorithm

circuit, the algorithm itself is explained in Section 5.1. The horizontal single and double lines represent respectively qubit and bit wires. The slash added to the upper wire indicates that this wire transports multiple qubits. The initial state of each wire is indicated at the beginning of each wire. The topmost qubit wire is measured at the end which yields useful classical information.

The rectangular boxes placed on top of the wires represent quantum operators. Such operators are linear unitary operations which can be described by means of matrices. The actual application of a quantum operator consists of a matrix multiplication. For example, Equation 4 presents the application of the Hadamard quantum operator H to the qubit $|1\rangle$. Because of the unitary evolution of quantum states, all quantum computations are *reversible*.

$$H|1\rangle = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \cdot \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{-1}{\sqrt{2}} \end{bmatrix} \tag{4}$$

The application of H to $|\psi\rangle$ on a quantum computer requires only one computational step. We call this property *quantum parallelism* because a classical computer requires two computational steps to compute the result. This is exactly what makes quantum computers so powerful: their computing power grows exponentially in comparison with classical computations. Contrastingly, this is also what makes quantum simulators so inefficient because the processing of 2^n superposition terms needs to be serialized.

In the above circuit, the quantum operator $H^{\otimes n}$ indicates that the Hadamard operator is applied to all n qubits of the topmost wire. Quantum operators can also operate on different qubit wires. For instance, the operator U_f, which implements the function f, takes the topmost qubit wire as input for f and applies the result of f to the other qubit wire. Optionally, one can associate a control qubit with a quantum operator. The latter is called a *conditional quantum operator* and is only applied if the control qubit is set. For example, the quantum circuit presented in Figure 3 employs the conditional operator R. The vertical bar points to the control qubit.

3 Motivation

This section motivates the different design decisions of QLisp and compares them with existing approaches. First, Section 3.1 highlights two important characteristics of *reality-based simulation* and explains the concrete implications of these characteristics in quantum simulator development. Next, Section 3.2 introduces another simulation approach, which we call *simulation as a model*. We compare both approaches and motivate why QLisp is a simulation as a model. Finally, the educational opportunities of QLisp are explained in Section 3.3.

3.1 Reality-based simulation

The domain of reality-based simulation tries to *imitate* concepts of the real world as accurately as possible. The state of the art in quantum simulator development indicates a high interest in such reality-based simulators [Ö03, BSC01]. These quantum simulators typically transform high-level quantum operators into sequences of efficient primitives. Afterwards, these primitives can be executed on a hypothetical quantum processor, using a hypothetical quantum memory. The claim is that the simulated processor and memory could be replaced easily with an actual processor and memory. Whether eventual quantum processors and memory will correspond to the hypothetical ones, still remains to be seen.

Next, reality-based simulation prohibits to perform actions that are not realisable in the real world. In terms of a quantum simulator, this means that e.g. qubits cannot be observed or cloned without disturbance due to the postulates of quantum mechanics. However, we argue that current research should lay more emphasis on quantum simulators that enable flexible experimentation capabilities as a means for exploring and investigating new quantum algorithms. This explains why quantum simulators that exceed the postulates of quantum mechanics and allow exploration and manipulation facilities, become more and more important. Opening the quantum mechanical borders allows us to perform experiments that are not realisable on a real quantum computer - if it should exist.

3.2 Simulation as a model

QLisp follows the notion of a *simulation as a model* which contrasts reality-based simulation. This section describes the two main characteristics of a simulation as a model and compares them with the reality-based approach. Furthermore, both characteristics are illustrated with a concrete example that introduces the basics of QLisp.

First, QLisp translates the quantum computations directly to the underlying mathematical formalism of quantum mechanics. This contrasts approaches like e.g. the transformation of high-level quantum operators into a universal set of primitive quantum operators [Tuc99]. We believe that *thinking in terms of mathematical concepts* is much more meaningful for humans than thinking in terms of low-level primitives, which are intended to be meaningful for computers. This design choice consequently increases the accessability of QLisp towards scientists from various disciplines other than computer science.

Example 1 *The following example clarifies how one can take advantage of thinking in terms of mathematical concepts. The function* `(make-qureg n &optional init-fn)` *constructs a quantum register of size n which represents an n-qubit. Optionally, one can define a higher-order* `init-fn` *that initializes quantum registers in an arbitrary state. For instance, the* `hadamard-init` *function initializes the register in the state $H^{\otimes n}|0\rangle$ which means that the Hadamard operator is applied to all qubits of the register. This higher order function has direct access to the internal representation of a quantum register, that is a matrix of dimension 1×2^n containing all amplitudes. The mathematical meaning of $H^{\otimes n}|0\rangle$ is that each amplitude of the resulting n-qubit has the value $1/\sqrt{2^n}$.*

```
(defun hadamard-init ()
  "return hadamard init-fn for quantum register"
  (lambda (matrix)
    (matrix-do (matrix entry element)
      (setf element
        (/ 1 (sqrt (matrix-rows matrix)))))))

(make-qureg n (hadamard-init))
```

Next, reality-based simulators typically offer at the most limited relaxations of the postulates of quantum mechanics. This means that classical operations like accessing or copying quantum data cannot be realized without disturbing the quantum mechanical system. By using a quantum simulator, we can go beyond the postulates of quantum mechanics. This enables new interesting opportunities like e.g. exploring the domain of quantum error correction [AB97]. QLisp allows a user to freely observe, modify and clone quantum registers and operators. Nevertheless, the choice of exceeding the postulates of quantum mechanics is optional.

Example 2 *The* `mod-exp` *function implements the conditional modular exponentiation operation ($x^a \bmod n$) on* `_qureg2_` *using* `_qureg1_` *as the control qubit. Because we are able to observe and modify the amplitudes without disturbance, we can implement the modular exponentiation using an efficient classical algorithm. The modular exponentiation quantum operator is a crucial part of the algorithm of Shor which computes prime factorizations. Throughout this paper, we use the following coding convention. Quantum registers* `_qureg_` *are surrounded by underscores and quantum operators* `-qop-` *with dashes.*

```
(defun mod-exp (_qureg1_ _qureg2_ x n)
  "modular exponentiation simulation"
  (let* ((size1 (qureg-size _qureg1_))
         (size2 (qureg-size _qureg2_))
         (ampl1 (amplitude-count _qureg1_))
         (ampl2 (amplitude-count _qureg2_))
         (_result_ (make-qureg (+ size1 size2))))
    (loop for basis from 0 below ampl1 do
      (let ((result-basis
              (+ (* basis ampl2)
                 (modular-exponentiation x basis n))))
        (setf (get-amplitude _result_ result-basis)
          (get-amplitude _qureg1_ basis))))
    _result_))
```

In reality, one should implement another quantum circuit that computes the modular exponentiation by means of low-level qubit operations, like e.g. [VVE96].

3.3 Educational opportunities

Several reasons indicate that QLisp has interesting educational opportunities. First, the quantum computations are expressed in terms of the underlying mathematical formalism. This offers students a concrete grasp on the notion of quantum computations in terms of linear mathematical operations. Next, since QLisp extends the postulates

of quantum mechanics, students can inspect the evolution of a particular algorithm and learn what is happening in terms of amplitude distribution. Finally, the use of a compact and expressive language allows students to concentrate on the quantum computations without suffering from other irrelevant technicalities.

4 Implementation of QLisp

QLisp is a language extension built on top of Lisp. The details of this approach are discussed in Section 4.1. Next, two optimization techniques are included in QLisp that prune in the exponential complexity degree of quantum mechanics. These are explained in Section 4.2.

4.1 Language extension

One of the important concerns of a quantum simulator is to express quantum computations in the right high-level programming language. The most straightforward option is to express the quantum computations in a classical programming paradigm, such as the procedural [Ö03] or object-oriented [BSC01] paradigm and limit the flexibility of the paradigm in order to match the postulates of quantum mechanics.

In QLisp, we adhere to the following quote by Abselson & Sussman: *"Programs must be written for people to read, and only incidentally for machines to execute."* [AS96]. Concretely, we extended the Common Lisp [AI96] programming language with additional layers of abstraction. These layers contain abstract data types and simulation algorithms to implement arbitrary quantum circuits. Lisp is extremely useful for these purposes because it was especially designed to be extended [Gra96].

We take advantage of the multi-paradigm approach of Lisp and the various powerful abstraction techniques to express quantum circuits without limiting ourselves to a particular programming paradigm. For instance, whereas the functional paradigm is suited to implement the various mathematical linear operations, the object-oriented paradigm is better suited to implement datastructures. Furthermore, the repetitive tasks are generalised using macros that hide a maximum level of details to increase readability. An illustration of this is shown in Example 3. The result of this all is a very compact, expressive and comprehensible programming environment.

Example 3 *This example employs the* `qureg-do` *macro which supports easy conditional iteration over the amplitudes of a quantum register. We use this macro to implement the swap operator, which swaps two qubits of a multiple qubit. The simulation function* `swap` *takes as input a quantum register and two indices (*`qid1` *and* `qid2`*) that indicate the qubits that should be swapped. At each iteration step of* `qureg-do`*, the symbol* `basis` *contains the decimal value of the current basis state and the corresponding* `amplitude`*. Optionally, one can define a* `filter` *that skips basis states that do not conform to the filter query. For instance, the application of the filter query* $|x11\rangle$ *means that only the basis states with decimal value 3 and 7 are considered. The macro* `qureg-do` *performs this job efficiently because it uses binary arithmetic to filter basis states instead of repeated conditional tests.*

```
(defun swap (_qureg_ qid1 qid2)
  "perform swap operation with qid1 < qid2"
  (let* ((_result_ (copy-qureg _qureg_))
         (jump (calc-jump (qureg-size _qureg_) qid1))
         (filter (make-normal-filter '((,qid1 0) (,qid2 1)))))
    (qureg-do
      (_result_ basis amplitude filter)
      (setf
        amplitude (get-amplitude _qureg_ (+ basis jump))
        (get-amplitude _result_ (+ basis jump)) amplitude))
    _result_))
```

4.2 Efficiency considerations

The most fundamental problem of simulating quantum computations on a classical computer is the exponential complexity degree in both space and time. The exponential space complexity is due to the fact that an n-qubit is internally represented as a column matrix containing 2^n amplitudes. Each amplitude is a complex number which consists of two floating point numbers. Quantum computations also suffer from an exponential time complexity. On a real quantum computer, the application of a quantum operator to an n-qubit means that the operator is applied to all 2^n superposition terms in one computational step. This phenomenon, called quantum parallelism, needs to be serialized on a classical computer. Moreover, parallel computers, which have multiple processing units, can only reduce the time complexity with a linear factor.

This section presents two optimizations, *sparse matrices* and *single operator application*, that prune away in space and time. These optimizations are only stopgap measures which support small computational problems. Due to the nature of quantum mechanics, the simulation of quantum computations in QLisp still suffers from an inherent exponential time and space complexity.

The first optimization reduces the memory consumption in certain cases by using sparse matrices. Both quantum registers and operators use sparse matrices to represent their state. The elements of a sparse matrix are stored in a hash-table that contains only non-zero elements.

Next to that, we also optimized the case in which an arbitrary quantum operator V is applied to a single qubit $|q_i\rangle$ that is part of an n-qubit $|\psi\rangle = |q_0 q_1 \dots q_{n-1}\rangle$. In the worst case, this should be calculated as follows using I as the identity operator. First, a quantum operator $O = I^{\otimes i} \otimes V \otimes I^{\otimes n-i-1}$ needs to be computed that matches the size of $|\psi\rangle$. Second, we need to compute the matrix multiplication $O|\psi\rangle$. Because the operator O contains a total of 2^{2n} elements with only 2^{n+1} elements different from zero, the number of calculation steps for such operators can be reduced. This kind of optimization is standard practice in QLisp. The macro `(qc-apply _qureg_ op-instr)` applies a list `op-instr` of quantum instructions to a quantum register `_qureg_`. A quantum instruction is a 3-tuple `(-op- qid cqid)` that consists of a quantum operator `-op-`, the index of the target qubit `qid` and an optional index of the conditional qubit `cqid`.

Example 4 *The code extract below implements a modified version of the quantum teleportation circuit (see Figure 2) which moves a qubit from one place to another. We choose this modified version because it clearly illustrates the expressiveness of the* `qc-apply` *macro. The following operators are applied from left to right: conditional*

not (`-cnot-`*), Hadamard (*`-h-`*), conditional Pauli-X (*`-x-`*) and conditional Pauli-Z (*`-z-`*).*

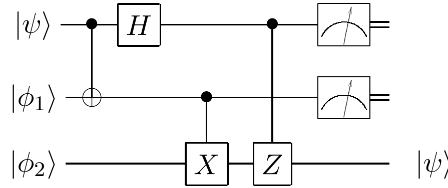

Figure 2: modified quantum teleportation circuit

The example uses the `init-qureg` *macro which initializes the qubits* $|\psi\rangle$ *and* $|\beta\rangle = |\phi_1\rangle \otimes |\phi_2\rangle$ *in a user-defined state. The function* `partial-measure-qureg` *measures a subset of qubits that are part of a quantum register by means of their index number.*

```
(let ((_psi_  (init-qureg
                ((/ 1 2) (/ (sqrt 3) 2))))
      (_beta_ (init-qureg
                ((/ 1 (sqrt 2)) 0 0 (/ 1 (sqrt 2))))))
  (partial-measure-qureg
    (qc-apply (tensor-items _psi_ _beta_)
      ((-cnot- 1 0) (-h- 0)
       (-x- 2 1) (-z- 2 0)))
    '(0 1)))
```

5 Applications

5.1 Algorithm of Deutsch-Jozsa

This algorithm solves the problem of Deutsch efficiently [DJ92]. Let us consider the function $f : \{0, 1, \ldots, 2^{n-1}\} \rightarrow \{0, 1\}$. We call the function f *constant* if it returns the same value, that is 0 or 1, for all elements of the domain. If f returns 0 for one half of the elements and 1 for the other half, we call f *balanced*. The problem of distinguishing between constant and balanced functions can be solved efficiently with the algorithm of Deutsch-Jozsa. The quantum circuit of this algorithm is displayed in Figure 1. The implementation in QLisp is as follows.

```
(defun deutsch-jozsa (n unitary-fn)
  "returns T if unitary-fn is constant"
  (let* ((_phi1_ (make-qureg n (hadamard-init)))
         (_phi2_ (qc-apply
                   (make-qureg 1 (standard-init 1)) (-h-)))
         (_psi_  (funcall unitary-fn
                   (tensor-items _phi1_ _phi2_))))
    (constant-qureg-p
     (collapse-basis
      (qc-apply-range _psi_ -h- 0 (1- n))))))
```

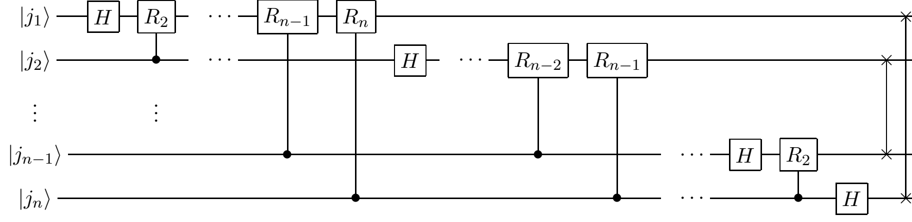

Figure 3: QFT quantum circuit

The parameter `unitary-fn` is a higher-order function that takes a quantum register as input, performs a quantum operation that implements $f(x)$ on it and returns the resulting quantum register. In the last but one step of the algorithm, the quantum operator $H^{\otimes n}$ is applied to the first n qubits. This is implemented with the `(qc-apply-range _qureg_ -op- from-qid to-qid)` function. The indices `from-qid` and `to-qid` determine the range of qubits of the quantum register `_psi_` to which the Hadamard quantum operator `-h-` is applied.

We obtain the result of the algorithm by measuring the quantum register `_psi_` with `collapse-basis`. The latter function returns the decimal value of the measured basis state using a random number generator. It can be mathematically proven that, if the measure result equals 0 or 1, the unitary function U_f is constant. In all other cases, U_f is balanced. This condition is checked by the `constant-qureg-p` predicate.

The best classical algorithm requires $2^n/2 + 1$ applications of $f(x)$ to verify if the function is constant or balanced. The algorithm of Deutsch-Jozsa can solve this problem more efficiently on a quantum computer consuming only one application of $f(x)$.

5.2 Quantum Fourier Transform

The Quantum Fourier Transform (QFT) [CEMM98, GN96] is the quantum counterpart of the classical Fourier transformation. Although the classical and quantum version have the same time complexity, we employ the quantum version because it is a required intermediate step of Shor's algorithm [Sho96]. The latter computes prime factorizations in polynomial time complexity. The quantum circuit that implements QFT is displayed in Figure 3. Equation (5) displays the matrix representation of the conditional quantum operator R_k used in this circuit.

$$R_k = \begin{pmatrix} 1 & 0 \\ 0 & e^{2\pi i/2^k} \end{pmatrix} \tag{5}$$

A parameterized quantum operator can be constructed using the `init-qop` macro. The latter facilitates the definition of user-defined quantum operators.

```
(defun get-operator-r (k)
  (let* ((t (/ (* 2 pi) (exp 2 k)))
         (factor (cis t)))
    (init-qop ((1 0) (0 factor)))))
```

We can now implement the quantum circuit of Figure 3. The repetitive parts of this circuit are reflected in the various loops that deal with the indices of the affected

qubits. In order to keep the design functional to a certain degree, we make use of the `copy-qureg` function that clones qubits. The latter violates the no-cloning theorem. The actual quantum computations are performed by the `qc-apply` macro.

```
(defun quantum-fourier (_qureg_)
  "apply quantum fourier via parallel network"
  (let ((size     (qureg-size _qureg_))
        (_result_ (copy-qureg _qureg_)))
    (loop for qid from 0 below size do
      (setf _result_ (qc-apply _result_ ((-h- qid))))
      (loop for c-qid from (1+ qid) below size
            for k from 2 do
        (setf _result_
          (qc-apply _result_ (((r-qop k) qid c-qid))))))
    (loop for i from 0 below (/ size 2)
          for j from (1- size) above (/ size 2) do
      (setf _result_ (swap _result_ i j)))
    _result_))
```

The result of the `quantum-fourier` function is a quantum register with the discrete Fourier transform applied to all amplitudes. The optimized simulation algorithm used in `qc-apply` and the quantum registers which are actually sparse matrices guarantee that the number of computational steps and memory usage to simulate the quantum computations are minimized.

6 Conclusion

QLisp is a compact, expressive and comprehensible language extension of the Lisp programming language for simulating quantum computations. This extension consists of abstract data types, simulation algorithms and useful macros that allow one to implement and compute arbitrary quantum circuits. The simulator has the flavour of a model because the quantum computations are translated into their mathematical counterpart. Furthermore, we relaxed the postulates of quantum mechanics in the sense that quantum states may be observed and modified anytime. These characteristics lead to a simulator that has a great potential for experimental and educational purposes.

There are two main optimizations included in QLisp that prune in the exponential complexity degree of quantum mechanics. The first optimization reduces memory consumption in certain cases by using sparse matrices. Next to that, we provide a simulation algorithm that reduces the number of calculation steps of a single quantum operator application.

The all-round capabilities of QLisp are illustrated by means of two applications. The algorithm of Deutsch-Jozsa solves the problem of Deutsch more efficiently than is possible with classical computations. Next, we implemented the quantum Fourier transform, an important intermediate step of the algorithm of Shor that calculates prime factorizations in polynomial time complexity.

References

[AB97] D. Aharonov and M. Ben-Or. Fault-tolerant quantum computation with constant error. *In Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing*, pages 176–188, 1997.

[AI96] American National Standards Institute and Information Technology Industry Council. *American National Standard for information technology: programming language — Common LISP: ANSI X3.226-1994*. 1996.

[AS96] Harold Abelson and Gerald J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA, USA, 1996.

[BSC01] S. Bettelli, Luciano Serafini, and T. Calarco. Toward an architecture for quantum programming. *CoRR*, cs.PL/0103009, 2001.

[CEMM98] R. Cleve, A. Ekert, C. Macchiavello, and M. Mosca. Quantum algorithms revisited. *Proc. R. Soc. of Lond. A*, 454:339–354, 1998.

[Deu85] D. Deutsch. Quantum theory, the Church-Turing principle and the universal quantum computer. *Proc. R. Soc. Lond. A*, 400:97–117, 1985.

[Dir47] P. A. M. Dirac. *The Principles of Quantum Mechanics*. The international series of monographs on physics. Clarendon Press, Oxford, 4 edition, 1947.

[DJ92] D. Deutsch and R. Jozsa. Rapid solution of problems by quantum computation. *In Proceedings of the Royal Society of London*, 439:553–558, 1992.

[GN96] R. B. Griffiths and C.-S. Niu. Semiclassical fourier transform for quantum computation. *Phys. Rev. Lett.*, 76:3228, 1996.

[Gra96] Paul Graham. *ANSI Common Lisp*. Prentice Hall Press, Upper Saddle River, NJ, USA, 1996.

[NC00] M. A. Nielsen and I. L. Chuang. *Quantum Computation and Quantum Information*. Cambridge University Press, Cambridge, UK, 2000.

[Ö03] Bernhard Ömer. *Structured Quantum Programming*. PhD thesis, Technical University of Vienna, 2003.

[RSA78] R. L. Rivest, A. Shamir, and L. Adleman. A method of obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[Sho96] P. W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM J. Comp.*, 26(5):1484–1509, 1996.

[Tuc99] Robert R. Tucci. A rudimentary quantum compiler(2cnd ed.). 1999.

[vN55] J. von Neumann. *Mathematical Foundations of Quantum Mechanics*. Princeton University Press, 1955.

[VVE96] A. Barenco V. Vedral and A. Ekert. Quantum networks for elementary arithmetic operations. *Phys. Rev. A*, 54:147, 1996.

[WZ82] W. Wootters and W. Zurek. A single quantum cannot be cloned. *Nature 299*, pages 802–803, 1982.