The QLisp Project
Development of a Quantum Computer Programming Language

QLisp is a project that creates a formal design and implementation of:

- A computer language for processing data on machines with non-deterministic (probabilistic) and quantum registers;
- a software simulator that can execute quantum software programs written in the language;
- the design for a hardware (non-quantum) parallel processing system that can emulate the machine with quantum registers; and
- using approximate polynomial models for certain quantum calculations.

The motivations for designing and building a quantum programming language (abbreviated as QLisp) are presented in this summary, together with the aimed results of the project.

The new language (QLisp) is an extension of the AI language ‘Common Lisp’, and thus is a fully general computing language for digital (classical) computers. Extensions to Common Lisp to add data-types and operators are permitted by the Lisp specification. Extensions to the macro world that implement quantum physics appear to be permitted by the specifications of the universe itself.

Background

Simulating quantum events on non-quantum hardware is intrinsically difficult. ¹ A quantum computer can do anything a digital computer can do (because digital computers are made with quantum atoms), but it can do many things in much less time. Examples are searching for matches in a database, factoring primes, and of course solving the equations of quantum mechanics. Quantum computation is astonishingly efficient in processor usage and time. A quantum computer could, for example, solve problems in microseconds that would take classical computers hundreds of thousands of years.

Quantum computers, when they exist in large scale, will be incredibly useful. A quantum search engine, properly programmed, will be incredibly powerful.

Previous Research in Quantum Programming Languages

Previous research is catalogued and summarized in the full project proposal. A brief survey can be found online.²

Previous Related Research—QLisp

In addition to QLisp, there also exist various quantum markup languages (QMLs). These include, notably, the QML of the Fraunhofer Institute.³ These, while not implementing

---

³ For a description see http://www.qc.fraunhofer.de/doc/Manual/30qml/
general purpose computer languages, at least permit expressing quantum algorithms as a series of quantum gates. It is a goal of this project that the QLisp compiler should be able to output a QML from a functional description. That is, QLisp will not only permit the user to specify quantum computations, but it will produce circuit diagrams of the hardware that would implement the program.4

**Why QLisp?**

Although we cannot today build large-scale quantum computers, we can build small scale ones. In five years, at most, we will likely have large-scale quantum computers. What will we do with them? What will they look like, and who will program them?

No digital computer can accurately simulate a quantum computer, at least not a big one, and not for long. So why bother to write such a language? The answer is the same for quantum computers as for ordinary digital computers. Alan Turing (and before him Ada Lovelace) designed general purpose computer languages long before general purpose computers were actually built, and the design of these languages strongly influenced the way that the hardware was implemented. Programmers in the 1960s generally traced out the execution of their programs by hand, because computer time was too expensive to waste on debugging loops. As Professor Sussman at M.I.T. is fond of pointing out, the primary benefit of computer languages may not be that we can program computers with them, but that they force us to systematize our understanding of the world to the point where a digital computer can emulate it.

Existing proposals for QPLs are based on imperative language models, which is a paradigm that most physicists avoid. Quantum calculations must, for example, be time-reversible, and something as simple as assigning a value to some variable can violate this rule, so ordinary imperative languages (Java or C++, for example) tend not to be good languages for expressing the ideas of quantum computing. Most QPL implementations require a great deal of book-keeping and explicit inverse function calls by the programmer (in part because of the time-reversible requirement), although some provide assistance.

We are developing and implementing QLisp for these reasons:

(1) If you can program something, then you understand it in a very deep and special way. The best (and in some cases the only) way to find out if your understanding is correct, is to run the program and look at the results. Currently, we do not do this with quantum mechanics, except in very special situations and then with a lot of hand-calculating (and hand-waving).

(2) We believe that practical programmable quantum computers are nearly here. We want to study what we can do with them, and how we can do it, without actually waiting for Intel to develop the “Quantium chip”.

(3) We want to implement a language that not only has a firm foundation in quantum theory (including a well-specified grammar) but also one that mirrors the way in

---

4Programs that output hardware designs are not novel. This is how, for example, gate array logics are often implemented.
which physicists and mathematicians think about quantum mechanics.

(4) We want to implement a working model, not just a design toy. That is, the model should execute reasonable programs in reasonable time and should support a high-speed simulator. We want results, not just correct form.

(5) We want to use delayed (lazy) functional evaluation based on probabilistic calculations. This is the strategy of putting off evaluation as long as possible (until a result from one calculation is actually needed by another). This helps reduce the number of quantum states that the system must calculate. We anticipate that the methods may have application in non-deterministic non-quantum languages as well.

(6) We want to use a large grab-bag of other optimization methods to reduce or approximate actual quantum calculations. When the system is then used, for example, to output a QLisp “wiring diagram”, the device will be smaller, which will help design actual and realizable quantum computers.

(7) We want to design and build a language that makes it difficult to program “quantum bugs”. Programs that are profoundly quantum-illegal are easy to write in conventional languages for digital computers. The simple assignment of a variable to a fixed state will probably be a quantum-illegal statement, for example. Programs that make copies of quantum entangled objects are certainly quantum-illegal. These types of “qbugs” can easily be caught by a functional language, but are harder to catch in traditional imperative languages.

**What will QLisp Look like?**

Like this:            (MEASURE (QBLOCk
(+(QCOND
((QFN-EQUAL #'F1 #'F2) 0)
 ( T 1))
 (F1 0))))

where the arithmetic operators like “+” preserve superposed states, QBLOCk is a quantum block marker, QCOND is a quantum conditional, QFN-EQUAL tests whether two quantum operators are equivalent, and the operator (MEASURE x) collapses the quantum state of x into a classical state.\(^5\)

QPLs generally have provided an imperative programming environment that really is not suitable for mathematical manipulation. That is, it is hard to use them to write programs

\(^5\) QCOND is interesting. In such a quantum conditional, the computer is, in some sense, taking “both” paths of the conditional, and provided no measurement occurs within the conditional, quantum states can be preserved. The issue of what happens when measurements (or entanglements) occur within one branch but not the other is one that is difficult and that is not well understood. It is one of the things we want to understand better. Also, QBLOCk is formally unnecessary, as quantum blocks can back-propagate through compiler rewrite rules until they encounter (for example) a MEASURE operator.
that manipulate other programs. In classical computation, the lambda calculus provides an alternative computational model to the Turing machine, and fully equivalent to it. The lambda calculus was developed especially for the study of logical function theory, and it forms the basis of several computer languages, including Lisp.

We do not propose solely to develop a lambda calculus for quantum mechanics. That has already been achieved by several researchers, notably A. Tonder.6 This quantum lambda calculus combines the benefits of both quantum circuits and the quantum Turing machine, and describes functions that may be composed and manipulated algebraically, like quantum circuits. Tonder has also implemented, in Scheme (a dialect of Lisp) a part of his quantum lambda calculus. The calculus has been shown to be Turing equivalent.7 We propose to adopt, with modifications, the quantum lambda calculus of Tonder as the starting point for the formal specification of QLisp. We chose to use Lisp rather than Scheme primarily because of the richer set of debugging and optimization tools available, and because once one has Lisp, it is easy to write Scheme, but the converse is not true.

Generally, QLisp will look like Lisp. It will have some important differences, including a set of quantum-complete operators, quantum conditionals, and operators to reduce quantum calculations to real numbers (i.e. to “collapse” quantum states).

WHAT WILL WE ACHIEVE?

(1) We expect significant advances in quantum error-correction protocols and networking of qubits, and the development of quantum information theory.

(2) We expect to make significant advances in exploring and understanding the relationship between classical probabilistic non-deterministic systems and quantum systems, including the modeling of certain types of classical systems on the quantum simulator. For example, we want to study the relationship between the gate implementation of quantum mechanical functions (which does not include conditionals) and the functional implementation (which may include conditionals). We know from some deep mathematical results that these two are equivalent, but it is very hard to understand how that can be true. The philosophical impact of this work will be to develop intuitions about the relationship between different branches of imperative-type conditional statements and superposed states of quantum gates. This directly impacts on the nature of reality itself and the possibility (for example) of free will.

(3) We expect to make, at least in prototype form, a computer language that will actually be useful to program real quantum computers.

(4) We want to make the quantum computing language evaluator (the simulator) available on the internet to other researchers.

6 Ibid.