Quantum programming languages

Dominique Unruh

Abstract We investigate the state of the art in the devel opment of quantum programming languages. Two kinds of such languages are distinguished, those targeting at practical applications like simulation or the programming of actual quantum computers, and those targeting the theoretical an alysis of quantum programs. We give an overview over ex isting work on both types and present open challenges yet to be resolved.

1 Why quantum programming languages?

A question that naturally arises is why there would be any need for quantum programming languages, since there are no quantum computers of notable complexity so far. In the present section we give some arguments why there may be need for such languages.

Theoretical examination of quantum algorithms. At the present time, quantum algorithms are mostly described either in some kind of partly formal pseudocode (cf. [11]) or by writing the algorithm as a quantum circuit (see [8] for an introduction). However, pseudocode lacks the ex actitude of a welldefined language, and the circuit model has only small expressivity. Comparing with classical (i.e., nonquantum) algorithm design, we note that al gorithms are only rarely described as circuits and that structured program code is often much more suited for presentation and analysis.

semantics and highlevel concepts, allowing abstraction from a concrete hardware model (like quantum circuits) and concentration on the main idea of the algorithm.

Experimental examination of quantum algorithms. In some cases it may not be possible to formally prove the correctness or efficiency of a given algorithm. It may depend on unproven mathematical assumptions or on heuristic methods. Then a test of the algorithm is ne cessary. Even in the absence of a quantum computer, a simulation could give instructive insight into the abil ities and problems of the algorithm (at least for small input sizes). And such trial and error searches for good al gorithms would then benefit from a simple and powerful method of entering and executing the algorithm.

Specification and verification of quantum cryptographic protocols. In the discipline of quantum cryptography, a multitude of protocols has been developed so far, some of which have even been implemented. However, in order to prove the security of a quantum protocol, it is in dispensable to formally define that protocol. Most ap proaches either use an informal description or quan tum circuits. Both have problems with the description of the concurrent execution of different parties. Further, when investigating larger quantum protocols (as may arise when composing a number of simple protocols to yield a large one, cf. [5, 29]) a description using circuits may become unwieldy.

Therefore the design and formal analysis of quantum pro tocols would benefit from a more elaborate formalism for describing the concurrent interaction of different entities.

From the above requirements two very different approaches to quantum programming languages arise: For proving the correctness of algorithms and protocols, programming lan guages with exactlydefined formal semantics are needed,

while experimentation and heuristic design of algorithms and when quantum computing hardware becomes avail able actual implementation needs powerful and easyto use languages, an intuitive specification of the behaviour may be sufficient instead of formal specification of the se mantics. For the sake of brevity we will call these two ap proaches formal programming languages and practical programming languages.

2 Practical programming languages

For experimental analysis of algorithms and, possibly, for the actual use of quantum computers, a software architecture is needed that eases the process of design and execution of quantum algorithms.

From the programmers point of view, a language should have the following features:

The language should be both simple and powerful. Sim plicity is needed so that no extended studies of the language should be necessary before using it, and to make the code more readable. A powerful language should guarantee that the programmer can concentrate on the main algorithmic problems without having to worry about technicalities.

The language should be technology independent (mean ing the technology of the underlying quantum computer, e.g., ion traps). At least at the current time, it is un known which quantum computer technology will even tually prove to be the most viable. Possibly, different technologies may even coexist. In this case the program mer should be able to write technology independent code which then is translated into a technology dependent se quence of instructions (like applying a laser pulse to some ion).

The language should transparently implement optimisa tions and error correction techniques. These are com plex but mechanisable processes. If the programmer had to integrate them manually, even a simple program would become very complex and unreadable. Secondly, optimisations and error correction techniques are of ten technologydependent (the costs of operations and the occurring kinds of errors vary). So a technology independent language should handle these transparently. It should be possible to execute the programs on a classi cal computer using a simulator. This has two advantages. First, prior to the advent of quantum computers, this al lows program testing. Secondly, even when large and fast quantum computers exist, there may still be some need for simulation: Due to the destructivity of measurements, it is impossible to inspect the state of a quantum com puter during debugging. In contrast, a simulator might

allow nonphysical operations like inspecting the state, forcing a given measurement outcome, etc., allowing for more comfortable and efficient debugging.

A potential software architecture for quantum programming has been proposed in [22]: Initially there is a technology independent highlevel quantum programming language, in which algorithms are implemented. This program is then converted by the front end of the system into a quantum intermediate representation (QIR). This QIR is still technol ogy independent. The QIR should be simple and suited for automated analysis and transformation. A possible choice for a QIR could be a description in terms of quantum cir cuits. The next layer of the proposed architecture is the tech nology independent optimiser. It transforms the QIR into a technology independent lowlevel description of the op erations to be performed, the quantum assembly language (QASM). The task of this optimiser is to perform all opti misations that do not depend on the actual technology (e.g., the cancellation of two consecutive Hadamard transform ations). In the next step the technology dependent optimiser transform the QASM into a quantum computing physical operations language (QCPOL). This language then contains the actual instructions for the physical system. The task of the optimiser is to make the calculation as fast and reliable as possible. At the end, the resulting QCPOL is either fed into a quantum computer for execution, or into a simulator for evaluation.

This architecture has the advantage that the different layers could be developed independently. So different quan tum programming languages could use different frontends, but the same optimisation code. And a change of underly ing technology only requires the use of another technology dependent optimiser. Further this separation might enable different groups of scientists to independently design tools for the different layers, but nevertheless guarantee interoper ability.

We now describe two quantum programming languages which aim at practical usability and come with a framework for execution and simulation.

2.1 QCL (Omer)

In [20, 21] the imperative quantum programming language QCL is presented. This language consists of a fullfledged set of classical operations (loops, branching, elementary and structured datatypes, etc.) augmented with quantum types.

The elementary quantum datatype is the quantum regis ter qureg. It represents a reference to one or several qubits on a socalled quantum heap (e.g., an external quantum computer). On these registers elementary operations can be performed: initialisation, unitary transformations, and measurements.

However, a special feature is the ability to write complex quantum operators. These operators are defined like clas sical procedures and functions, but they take one or more quantum registers as arguments. Unlike a classical proced ure that applies the same operations to the arguments, an operator can be inverted and in the special case of a basis permutation provides automatic management of scratch registers.

Besides the basic register type, there exist several variants of this datatype. Though the physical interpretation of these kinds of registers is identical, the different types impose different constraints on the operations on these bits. A constant register quconst implies that the operator may not mod ify the register (e.g., the controlling qubit in a CNOT might be a constant register). A void register quvoid is guaran teed to be empty at the beginning of the execution of an operator (e.g., when implementing a classical function f as x y x y f(x) , the second register is usually as sumed to be empty). A scratch register quscratch is also assumed to be empty at the beginning, and must also be left empty afterwards.

The approach that operators are defined like procedures has the advantage that programs can be written in a very homogeneous way: a classical function and the correspond ing quantum function are represented by essentially the same code. Further, rather powerful constructs like quantum branching are supported with this approach (cf. Sect. 3.5).

The language QCL has been implemented and comes with a simulator for testing. The software can be found at [19].

2.2 Q (Bettelli, Calarco, Serafini)

In contrast to QCL (see preceding section), the language Q presented in [3] has not been designed from scratch, but uses the object oriented features of C to implement quantum registers and operators. This has the advantage that the rich and powerful classical abilities of C and existing C/C libraries may be used, and that no special compiler needs to be implemented.

A quantum register is a class Qreg which as in QCL

represents a list of references to qubits. Another class Qop represents operators which can be applied to registers. Several operations on operators are available: inver sion, composition (sequential), reordering of input/output qubits, application to a registers, making a controlled op erator, creation from a classical function, etc. Complex op erators can therefore be build up from elementary ones. One problem should be noted: The creation of an opera tor from a classical function (i.e., constructing the oper ator x y x y f(x) ) takes a pointer to a function as argument, i.e., the function is given as a black box. It is easy to see that creating the operator needs an expo

nential number of queries to the function, even for simple functions.

The advantage of first constructing the operators and only then applying them to the register it that: the underlying li brary can optimise a given operator at construction time. If the operator is applied many times, it only has to be opti mised once. The disadvantage is that in contrast to QPL

a more restricted style of programming has to be used, resembling the creation of quantum circuits.

The language Q has been implemented together with a simulator. The software can be found at [4].

3 Formal programming languages

A formal specification of a quantum programming language is separated into two parts: syntax and semantics. Since the definition of a syntax is a well understood problem that does not seem to change significantly due to the quantum nature of the programs, most scientific work on formal quantum programming languages concentrates on how to model the semantics of a quantum program. In the present section we will discuss several different possible approaches of mod elling quantum programming languages.