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.

# 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 well-defined language, and the circuit model has only small expressivity. Comparing with classical (i.e., non-quantum) 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 high-level 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 exactly-defined formal semantics are needed,

while experimentation and heuristic design of algorithms and – when quantum computing hardware becomes avail- able – actual implementation needs powerful and easy-to- 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”.

# 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 programmer’s 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 technology-dependent (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 non-physical 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 high-level 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 low-level 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 front-ends, 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.

* 1. QCL (O¨ mer)

In [20, 21] the imperative quantum programming language QCL is presented. This language consists of a full-fledged 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 so-called *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 dif- ferent 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].

* 1. 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 regis- ters. 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].

# 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.

* 1. Imperative and functional languages

A first and major semantic distinction is between impera- tive and functional languages. Imperative languages are de- scribed by specifying how the execution of a given program modifies a global state. The imperative approach has the fol- lowing two advantages: First, it models more closely how actual hardware works and thus makes it easier to actually implement these languages. Second, programmers without a background in formal languages tend to find the impera- tive approach more natural; in fact, programming is mostly done in imperative programming languages like C++ and Java. On the other hand, programs in functional languages map inputs to outputs, and more complex programs are built out of elementary functions. This approach puts more em- phasis on the data flow than the imperative one and is usu- ally easier to analyse and define formally. Particularly, the compile-time checking of data types can be handled much more easily with functional languages.

In this light, it is not surprising that the practical lan- guages described in Sect. 2 follow the imperative paradigm. Other examples for the imperative approach are the pro- cess calculi (see Sect. 3.2) and the modelling of programs as measurements [27] (see Sect. 3.6). Furthermore, the lan- guage QPL [23] (see Sect. 3.6), though formally a functional language, has the outer form of an imperative language and can therefore more easily be used by a programmer familiar with the imperative approach.

Examples of functional languages are the lambda cal- culi [1, 25, 30, 31] (see Sect. 3.6). Lambda calculi are pro- gramming languages that have the special feature that func- tions can again be considered as data, i.e. functions can take other functions as arguments. A crucial point in the devel- opment of a quantum lambda calculus is whether functions must be considered as classical data, or whether functions may be quantum data, i.e. a variable may contain a su- perposition of different functions with completely different program code. The languages of [25, 30, 31] require func- tions to be classical data, while [1] has a quantum data type representing a function (see Sect. 3.6). However it is ques- tionable whether such a datatype is physically justifiable since it is unclear how that datatype could be reduced to more primitive quantum operations (like manipulations of qubits). To the author’s knowledge, no physically realisable way of encoding functions as quantum data has yet been proposed.

* 1. Languages for concurrent processes

As we have seen in our motivating discussion, an import- ant application of formal quantum programming languages is quantum cryptography. However, since a cryptographic protocol usually consists of two or more parties that run con- currently and communicate, a programming language for the specification of protocols has to provide means for mes- sage transmission between different concurrent processes. A class of languages that contains primitives for communi- cation is given by the process calculi. Examples of quantum process calculi are QPAlg [12] and CQP [10] (see Sect. 3.6). However, for languages having concurrent processes the following problem arises: Since in most realistic scenarios machines are not perfectly synchronised, so-called race con- ditions may occur: E.g., two processes simultaneously send a message to a third one; which message reaches its destina- tion first? To give a process calculus formal semantics, such scheduling questions must be answered. The most immedi- ate solution – that is used both in [10, 12] – is to introduce nondeterminism: a protocol satisfies a property only if the property is guaranteed, even if at any point of the execution *any* scheduling decision may be made. This yields a suffi- ciently simple modelling, however it turns out that nondeter- ministic scheduling is too strict a modelling for the analysis of cryptographic protocols which have a small probability of failure. The following example demonstrates this.

Assume that in a larger quantum protocol the following situation arises at some point. Party *A* holds a classical se- cret *k* (e.g., a key). If a third party *E* manages to guess that key, it can break the protocol. Now, with nondeterministic scheduling, we cannot guarantee that *E* does not gain know- ledge of the key (even if *A* does not even use the secret *k*). Imagine that *E* performs the following experiment: it sends

pairs of message 0, 1 to itself and observes which messages come back first. This gives *E* a string *s* of bits. Nonde- terministic scheduling does not exclude the possibility that *s k*, which implies that *E* learns *k*. So, nondeterministic scheduling is so strict that it implies that no classical infor- mation can ever stay secret.

Note that these problems do also occur with classical probabilistic protocols. A solution is to model the schedul- ing as a kind of process that is queried for every scheduling decision (for the classical case, see e.g. [15]). Then a proto- col would be secure if it fulfils some security property for all schedulers. To the best of our knowledge, such approaches have not yet been investigated for quantum process calculi.1

=

* 1. Physical model

Another very important design decision in the specifica- tion of a quantum programming language is the choice of an underlying mathematical model for the laws of quantum physics.

The two most common abstractions of quantum physics used in quantum computing and cryptography are the unitary-operations-approach and the density-operators- approach.

In the unitary-operations-approach, the state of a sys- tem is assumed to be a vector in a Hilbert space. Oper- ations on the state are represented by unitary transform- ations, while measurements are given by projections onto orthogonal subspaces of the Hilbert space. When describ- ing the behaviour of a process that contains both unitary operations and measurements, one necessarily gets proba- bility distributions of states of the system, so-called ensem- bles. Therefore quantum programming languages based on the unitary-operations-approach usually have to explicitly model a probabilistic process on top of the unitary evolu- tions of the system.

The approach of using density-operators is based on the following observation: There are different ensembles of quantum states that are not distinguishable by any experi- ment. If one defines two ensembles as equivalent if they cannot be distinguished, one finds that a state (i.e., a equiv- alence class of ensembles) can be represented as a density- operator. The density-operator formalism gives a mathe- matically elegant solution to the question of experimental distinguishability of states (or ensembles) and is therefore especially popular in quantum cryptography. Furthermore, since the density-operators already allow the encoding of probabilistic mixtures, it is not necessary to additionally model a probabilistic process on top.

1 However, [29] specifies a model for quantum protocols with exter- nal scheduling based on the classical model [6]. No programming language is specified in that model.

For a thorough exposition of these concepts cf. e.g. the textbook [18].

An advantage of using the density-operator formalism in the semantics of a programming language is the fol- lowing: If two programs operate on a system or input in an indistinguishable way, a description of their opera- tion will have the same description in terms of density- operators, i.e., two programs have the same semantics if they cannot be distinguished. In contrast, using unitary op- erations two programs will most probably be considered different if they create different but indistinguishable en- sembles. (E.g., one program could encode a random bit in the computational basis, while the other program encodes it in the diagonal basis. If the random bit is not otherwise stored, these are indistinguishable, but the ensembles are clearly different.) Since this indistinguishability of ensem- bles is a very important concept in many quantum cryp- tographic protocols, semantics based on density-matrices might be more suited for the description of cryptographic protocols.

However, note that statements about the classical out- put of a program should not be dependent on the chosen approach, since both are models of the same physical the- ory and therefore cannot predict different outcomes for one experiment.

Examples of quantum programming languages with se- mantics based on density-operators are [23, 27], while uni- tary semantics are used in pQCL [26, 33], QPAlg [12], CQP [10] and the quantum lambda-calculi [1, 25, 30, 31] (see Sect. 3.6 for a short overview of these languages).

There are also quantum programming languages based on other models of quantum physics. E.g., in [7] a modal quantum logic is presented whose axioms only distinguish between events that *will necessarily* occur, *cannot* occur or *may* occur, i.e., no probabilities are associated with these events. Algorithms with small probability of failure cannot be modelled in that language.

Since there are different flavours of quantum logics (see [32] for a short introduction), many different program- ming languages based on these logics can be imagined. However, it may be difficult to adapt proofs that rely on tra- ditional modellings of Hilbert space quantum mechanics to settings based on quantum logics.

* 1. Augmenting classical languages

Classical programming languages are a well-understood area. Therefore it would seem advantageous to make use of existing classical languages that satisfy most of our needs, augmenting them with quantum processing. From this wish, a natural and viable approach for designing quantum pro- gramming languages emerges. It can be sketched by the following recipe:

Take a classical probabilistic language (for the sake of simplicity we assume an imperative one).

Add a new datatype: reference to a qubit (i.e., the index of a qubit of the classical state)

•

•

The global state consists of all information needed during the execution of the program (e.g., classical variables, in- struction pointer). Extend this global state to also contain a vector in a sufficiently large Hilbert space (the quantum state of the system).

•

Define elementary operations on qubits (e.g., unitary transformations, measurements): Formally this operation takes references to qubits and then modifies the global state as a side effect. But from the programmers point of view, the operation operates directly on the qubits.

•

Formal quantum languages based on this hybrid approach are e.g. QPAlg [12], CQP [10], and the lambda calculus in [25, 30] (cf. Sect. 3.6).

Note further that the practical languages QCL [20, 21] and Q [3] described in Sect. 2 are based on this approach, too.

Some care has to be taken that no operation is invoked with multiple references to the same qubit, since this would not have a well-defined physical meaning (it would imply cloning of the state). One approach is to raise runtime er- rors in this case (i.e., when an operation is invoked, it is checked whether the qubits are different), the other is to use an appropriate type system which enforces this condition at compile-time, i.e., no well-formed program can be written that applies an operation to non-disjoint qubits. Languages that make use of a such type system are e.g. the lambda cal- culi in [25, 30, 31] (cf. Sect. 3.6).

However, this approach also presents some problems. First, the recipe given above yields unitary-operations- semantics, extending it to density-operator-semantics may be non-trivial. Second, since we explicitly refer to the global state, it is not necessarily straightforward to adapt this approach to functional languages. Third, it does not give semantics for quantum branching in the sense that some part of the program is executed depending on the value of a qubit without destroying the superposition of that qubit (cf. Sect. 3.5). Finally, since this approach imposes a strict separation of quantum data and classical control, higher-order data types may be excluded. E.g., if func- tions shall be considered as data, this separation might pose a problem.

* 1. Useful features and challenges

In this section we present several possible features of quan- tum programming languages. Some of these can already be found in some of the languages proposed in the literature (cf. Sect. 3.6), while others are still open challenges.

* + 1. *Quantum branching*

By quantum branching we mean that the value of a given qubit conditions the execution of some piece of code. If the qubit is in superposition, execution and non-execution also happen in superposition. In this respect it differs from clas- sical branching which *measures* whether the condition is fulfilled, thereby destroying the superposition. A very sim- ple example for quantum branching is the CNOT-gate where one qubit is flipped if the other has value 1.

Two programs that differ only by a change of global phase have the same observable behaviour. However, when subject to a conditional execution, these two programs may suddenly behave differently (consider e.g. a conditional phase flip, which is different from the identity). Further, measurements and erasures do not seem to have a natural meaning when subject to a quantum condition. These two points make it more difficult to design program semantics that encompass both irreversible operations (measurements, erasures) and quantum branching.

The language QCL [21] (see also Sect. 2.1) implements quantum branching. In special cases, it even provides the possibility of using assignments to classical variables and measurements inside conditioned code. Furthermore, quan- tum loops are supported, as long as there is a classically known upper bound.

* + 1. *Continuous classical output*

Most algorithms take an input, calculate, and give an out- put. However, in some cases a program continuously outputs classical information, e.g., information about its progress. This output can be observed before the program’s termina- tion, and moreover, even a non-terminating program may have output. A further application of continuous output might be to describe the externally visible behaviour (i.e., the trace) of a process, e.g., in a cryptographic setting the re- quired security properties might be formulated in terms of the externally observable behaviour of processes.2

When following the hybrid approach of Sect. 3.4 it is sufficient to add the possibility of output to the underlying classical language. To the best of our knowledge, this has not yet been formally done. In the case of density-operator- based semantics this problem has been investigated in [27].

the scheduling turns out to be difficult, cf. Sect. 3.2. Exam- ples for languages with concurrent processes are QPAlg [12] and CQP [10] (see Sect. 3.2). To the author’s knowledge, the problems with nondeterministic scheduling outlined in Sect. 3.2 remain as yet unsolved.

* + 1. *Infinite datatypes*

Most languages proposed so far only have quantum data- types which live in a finite dimensional Hilbert space. Many quite elementary datatypes like integers or strings cannot be represented in such a model.3 However, since the math- ematics on countably dimensional Hilbert spaces is very similar to that on finite dimensional ones, in most cases a language designed for finite quantum datatypes can eas- ily be extended to the countably dimensional case. Only the uncountably dimensional case (e.g., real numbers) present a major definitional challenge.

To the author’s knowledge, infinite quantum datatypes have not yet been presented in the literature.

* + 1. *Higher-order data types*

While today’s quantum algorithms and protocols can suffi- ciently easily be expressed in terms of qubits as datatypes, more complex datatypes can ease the presentation. E.g., a high-level presentation of Shor’s algorithm for solving the discrete logarithm in arbitrary groups [24] would op- erate on group elements, and only when considering an actual implementation would one fix a concrete encoding of group elements as strings of qubits. Future development in quantum algorithms might require still more complex quantum datatypes, e.g., tupels, lists, records, etc. Trying to incorporate such datatypes presents us with a defini- tional challenge: Assume a program holds an list *a* in one quantum register, and an index *i* in another quantum reg- ister. Now the program accesses the *i*-th element of *a*. Since the superposition must not be destroyed (unless an explicit measurement occurs) we cannot check whether *a* indeed contains an *i*-th element and then raise an error. Therefore semantics coping with such advanced higher- order datatypes must specify a behaviour for such “errors in superposition”.

To the author’s knowledge, only in QPL (cf. Sect. 3.6) have semantics for higher-order datatypes been speci- fied [23]. However, these datatypes are not completely quantum, they represent *classical* tupels and lists of quan- tum bits, i.e., in the case of lists the length of the list is

3 Of course, in real (classical) machines integers are finite datatypes, too; however, in the course of the design of an algorithm one might want to use unlimited integers at an early development stage and only later replace them by their finite counterparts.

a classically observable property. Due to this restriction the aforementioned problems do not occur.

* + 1. *Powerful reasoning about programs*

It is not enough just to have a mathematical definition of the semantics of a programming language. To prove properties of an algorithm (manually or automatically), the language should come with a set of laws. These should be power- ful enough to allow one to concentrate on the main idea of the algorithm without losing oneself in language-specific details. Future experience in the work on and with quan- tum programming languages will probably show which ap- proaches to this problem are the most fruitful ones.