Quantum Programming Languages[⋆]

Dominique Unruh

Saarland University, Germany, unruh@cs.uni-sb.de

Abstract. We investigate the state of the art in the development 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 analysis of quantum programs. We give an overview over existing work on both types and present open challenges yet to be resolved.

Table of Contents

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. [Kni96]) or by writing the algorithm as a quantum circuit (see [Cle00] for an introduction). However, pseudocode lacks the exactitude 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 algorithms are only rarely described as circuits and that structured program code is often much more suited for presentation and analysis.
  Therefore the design and analysis of quantum algorithms might benefit from programming languages with clear semantics and high-level concepts, allowing abstraction from a concrete hardware model (like quantum circuits) and concentration on the main idea of the algorithm.

[⋆] Published in Informatik – Forschung und Entwicklung 21(1):55–63, Springer, 2006

- *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 necessary. Even in the absence of a quantum computer, a simulation could give instructive insight into the abilities and problems of the algorithm (at least for small input sizes). And such a trial and error searches for good algorithms 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 indispensable to formally define that protocol. Most approaches either use an informal description or quantum 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. [BOM04,Unr04c]) a description using circuits may become unwieldy.
  Therefore the design and formal analysis of quantum protocols 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 languages with exactly-defined formal semantics are needed, while experimentation and heuristic design of algorithms and—when quantum computing hardware becomes available—actual implementation needs powerful and easy-to-use languages, an intuitive specification of the behaviour may be sufficient instead of formal specification of the semantics. For the sake of brevity we will call these two approaches "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 programmer's point of view, a language should have the following features:

- The language should be both simple and powerful. Simplicity 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 (meaning the technology of the underlying quantum computer, e.g., ion traps). At least at the current time, it is unknown which quantum computer technology will eventually prove to be the most viable. Possibly, different technologies may even coexist. In this case the programmer should be able to write technology independent code which then is translated into an technology dependent sequence of instructions (like applying a laser pulse to some ion).
- The language should transparently implement optimisations and error correction techniques. These are complex 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 often 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 classical computer using a simulator. This has two advantages. First, prior to the advent of quantum computers, this allows 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 computer 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 [SCA$^+$04]: 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 technology 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 circuits. The next layer of the proposed architecture is the *technology independent optimiser.* It transforms the QIR into a technology independent low-level description of the operations to be performed, the *quantum assembly language (QASM).* The task of this optimiser is to perform all optimisations that do not depend on the actual technology (e.g., the cancellation of two consecutive Hadamard transformations). 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 quantum programming languages could use different front-ends, but the same optimisation code. And a change of underlying 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 interoperability.

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

2.1 QCL (Ömer)

In [Öme03a,Öme03b] 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 register `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 classical procedures and functions, but they take one or more quantum registers as arguments. Unlike a classical procedure 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 modify the register (e.g., the controlling qubit in a CNOT might be a constant register). A void register `quvoid` is guaranteed to be empty at the beginning of the execution of an operator (e.g., when implementing a classical function f as $|x\rangle|y\rangle \mapsto |x\rangle|y \oplus f(x)\rangle$, the second register is usually assumed 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 corresponding quantum function are represented by essentially the same code. Further, rather powerful constructs like quantum branching are supported with this approach (cf. Section 3.5).

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

2.2 Q (Bettelli, Calarco, Serafini)

In contrast to QCL (see preceding section), the language Q presented in [BCS03] 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: inversion, composition (sequential), reordering of input/output qubits, application to a registers, making a controlled operator, creation from a classical function, etc. Complex operators can therefore be build up from elementary ones. One problem should be noted: The creation of an operator from a classical function (i.e., constructing the operator $|x\rangle|y\rangle \mapsto |x\rangle|y \oplus f(x)\rangle$) 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 exponential 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 library can optimise a given operator at construction time. If the operator is applied many times, it only has to be optimised 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 [Bet].

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 modelling quantum programming languages.

3.1 Imperative and functional languages

A first and major semantic distinction is between imperative and functional languages. Imperative languages are described by specifying how the execution of a given program modifies a global state. The imperative approach has the following 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 imperative 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 function. This approach puts more emphasis on the data flow than the imperative one and is usually 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 languages described in Section 2 follow the imperative paradigm. Other examples for the imperative approach are the process calculi (see Section 3.2) and the modelling of programs as measurements [Unr04a] (see Section 3.6). Furthermore, the language QPL [Sel04] (see Section 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 calculi [vT04,Val04,SV04,AD04] (see Section 3.6). Lambda calculi are programming languages that have the special feature that functions can again be considered as data, i.e. functions can take other functions as arguments. A crucial point in the development 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 superposition of different functions with completely different program code. The languages of [vT04,Val04,SV04] require functions to be classical data, while [AD04] has a quantum data type representing a function (see Section 3.6). However it is questionable 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.

3.2 Languages for concurrent processes

As we have seen in our motivating discussion, an important application of formal quantum programming languages is quantum cryptography. However, since a cryptographic protocol usually consists of two or more parties that run concurrently and communicate, a programming language for the specification of protocols has to provide means for message transmission between different concurrent processes. A class of languages that contains primitives for communication is given by the process calculi. Examples of quantum process calculi are QPAlg [LJ04] and CQP [GN04] (see Section 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 conditions may occur: E.g., two processes simultaneously send a message to a third one; which message reaches its destination first? To give a process calculus formal semantics, such scheduling questions must be answered. The most immediate solution—that is used both in [LJ04, GN04]—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 sufficiently simple modelling, however it turns out that nondeterministic 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 secret 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 knowledge 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. Nondeterministic 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 information can ever stay secret.

Note that these problems do also occur with classical probabilistic protocols. A solution is to model the scheduling as a kind of process that is queried for every scheduling decision (for the classical case, see e.g. [MMS03]). Then a protocol 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]

3.3 Physical model

Another very important design decision in the specification 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 system is assumed to be a vector in a Hilbert space. Operations on the state are represented by unitary transformations, while measurements are given by projections onto orthogonal subspaces of the Hilbert space. When describing the behaviour of a process that contains both unitary operations and measurements, one necessarily gets probability distributions of states of the system, so-called ensembles. Therefore quantum programming languages based on the unitary-operations-approach usually have to explicitly model a probabilistic process on top of the unitary evolutions 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 experiment. If one defines two ensembles as equivalent if they cannot be distinguished, one finds that a state (i.e., a equivalence class of ensembles) can be represented as a density operator. The density-operator formalism gives a mathematically 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.

For a thorough exposition of these concepts cf. e.g. the textbook [NC00].

An advantage of using the density-operator formalism in the semantics of a programming language is the following: If two programs operate on a system or input in an indistinguishable way, a description of their operation 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 operations two programs will most probably be considered different if they create different but indistinguishable ensembles. (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 ensembles is a very important concept in many quantum cryptographic protocols, semantics based on density matrices might be more suited for the description of cryptographic protocols.

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

Examples of quantum programming languages with semantics based on density operators are [Sel04,Unr04a], while unitary semantics are used in pQCL [SZ00,Zul04], QPAlg [LJ04], CQP

[1] However, [Unr04c] specifies a model for quantum protocols with external scheduling based on the classical model [BPW04]. No programming language is specified in that model.

[GN04] and the quantum lambda-calculi [vT04,Val04,SV04,AD04] (see Section 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 [BS04] 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 [Wil03] for a short introduction), many different programming languages based on these logics can be imagined. However, it may be difficult to adapt proofs that rely on traditional modellings of Hilbert space quantum mechanics to settings based on quantum logics.

3.4 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 programming 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, instruction 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 [LJ04], CQP [GN04], and the lambda calculus in [Val04,SV04] (cf. Section 3.6).

Note further that the practical languages QCL [Öme03a,Öme03b] and Q [BCS03] described in Section 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 errors 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 calculi in [vT04,Val04,SV04] (cf. Section 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. Section 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 functions shall be considered as data, this separation might pose a problem.

3.5 Useful features and challenges

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

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 classical branching which *measures* whether the condition is fulfilled, thereby destroying the superposition. A very simple 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 [Ōme03b] (see also Section 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, quantum loops are supported, as long as there is a classically known upper bound.

Continuous classical output Most algorithms take an input, calculate, and give an output. 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 termination, 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 required security properties might be formulated in terms of the externally observable behaviour of processes.[2]

When following the hybrid approach of Section 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 [Unr04a].

Concurrent processes For cryptologic applications it is very important to be able to express and to have semantics for the interaction of concurrent processes. Yet formally specifying the behaviour of the scheduling turns out to be difficult, cf. Section 3.2. Examples for languages with concurrent processes are QPAlg [LJ04] and CQP [GN04] (see Section 3.2). To the author's knowledge, the problems with nondeterministic scheduling outlined in Section 3.2 remain as yet unsolved.

Infinite datatypes Most languages proposed so far only have quantum datatypes which live in a finite dimensional Hilbert space. Many quite elementary datatypes like integers or strings cannot

[2] E.g., in the quantum security model [Unr04c] this approach is chosen following the classical model [BPW04].

be represented in such a model.[3] However, since the mathematics 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 easily 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.

Higher-order data types While today's quantum algorithms and protocols can sufficiently 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 [Sho94] would operate 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 definitional challenge: Assume a program holds an list a in one quantum register, and an index i in another quantum register. 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. Section 3.6) have semantics for higher-order datatypes been specified ([Sel04]). However, these datatypes are not completely quantum, they represent *classical* tupels and lists of quantum bits, i.e., in the case of lists the length of the list is a classically observable property. Due to this restriction the aforementioned problems do not occur.

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 powerful 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 quantum programming languages will probably show which approaches to this problem are the most fruitful ones.

3.6 Proposed languages

In this section we present an exemplary selection of formal quantum programming languages. We try to present the basic idea underlying the languages' semantics.

Quantum Flow Charts, QPL (Selinger) The language presented in [Sel04] represents a program as a flow chart. Such a flow chart consists of elementary building blocks such as unitary operations, measurements, etc. A flow chart may have several input and output edges, representing the control flow of the program; each edge can be thought of as a different *classical* choice. E.g., a binary measurement would have two outgoing edges, one for each outcome.

Since a flow chart can again be used as an elementary building block, modular design of flow charts is easily realisable.

[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.

Loops and recursion are realised as flow charts with cycles. A block where one outgoing edge is at the same time its ingoing edge is a loop that can only be left through another outgoing edge.

The semantics are described in terms of density-operators and superoperators, so two indistinguishable programs have the same semantics.

Since flow charts, though conceptually interesting, are cumbersome to manipulate in formal calculations, a textual variant of the language exists called QPL.

Interestingly, though the language is a functional one, the outer form of QPL strongly resembles that of an imperative language. This may make it easier to use for people unfamiliar with functional programming.

Lambda calculus (van Tonder) In [vT04] a lambda calculus is presented. To ensure reversibility (i.e., unitary evolution), after each step some information is saved to allow the reconstruction of the previous state. A type system is added to restrict the allowed operations and to enable equational reasoning about programs.

It can be shown that in a superposition of different states (lambda terms) all terms have the same outer form, i.e., they only differ in the values of the actual qubits. Therefore one can image the state as consisting of a classical term containing qubits (similar to the hybrid approach, Section 3.4).

qGCL (Zuliani, Sanders) The language qGCL [SZ00,Zul04] is based on the classical probabilistic language pGCL [MM99] which again is based on Dijkstra's guarded-command language [Dij76]. A new datatype is added to pGCL, the quantum register, containing a vector from a finite dimensional Hilbert space. Then operations like initialisations, unitary transformations, and measurements can be specified as pGCL subroutines. This automatically gives the language qQCL clear semantics, and it inherits the laws of pGCL, so that reasoning about programs is supported, too. Further, qGCL also contains the possibility of nondeterministic choices, allowing for a refinement calculus.

However, due to the fact that each register contains a vector from an individual Hilbert space, no entanglement between different registers is possible. So any quantum algorithm that uses entanglement between different registers has to be rewritten to contain only one register.

Particularly, quantum cryptographic protocols probably cannot be expressed.

QPAlg (Lalire, Jorrand) In [LJ04] the language QPAlg is presented. This language is based on a classical process calculus similar to CCL [Mil89] and Lotos [BB87]. This process calculus is then extended by probabilism and quantum datatypes (using a method similar to the hybrid approach sketched in Section 3.4). Unfortunately, due to the non-determinism arising from scheduling, the problems explained in Section 3.2 occur.

The no-cloning property is ensured, since upon application of a unitary transformation to a list of qubits there only is an inference rule if the qubit references are different. In other words, the program gets stuck if it tries to apply a unitary to multiple references to the same qubit.

CQP (Gay, Nagarajan) The language CQP [GN04] is based on the π-calculus [MPW92a, MPW92b]. This process calculus is then extended to encompass probabilism and quantum datatypes (using a method similar to the hybrid approach sketched in Section 3.4). Unfortunately, due to the non-determinism arising from scheduling, the problems explained in Section 3.2 occur.

The no-cloning property is ensured in the same way as with QPAlg.

Lambda calculus (Selinger, Valiron) In [Val04,SV04] a lambda calculus is extended with qubits, using a method similar to the hybrid approach in Section 3.4.

A type system is developed to ensure that qubits cannot be referenced twice, thus ensuring the non-cloning property without the need of runtime checks.

Lambda calculus (Arrighi, Dowek) In [AD04] a variant of the lambda calculus is presented. The semantics are based on a term rewriting system (TRS), even the basic mathematical elements like complex numbers are encoded in the TRS, resulting in many reduction rules.

A very interesting feature of this language is that functions can be regarded as data. More specifically, an operator $\mathbb{C}^n \to \mathbb{C}^m$ can be represented as a state in $\mathbb{C}^n \otimes \mathbb{C}^m$ (since there is a natural isomorphism between the set of linear operators and these states). However, it is not clear that there can be a physical realisation of such a datatype, since applying an operator (given as a state) to another state might be a difficult problem.

Programs as measurements (Unruh) In [Unr04a,Unr04b], a mathematical framework for specifying the semantics of languages with continuous classical output (cf. Section 3.5) is presented. The approach is based on density operators. The basic idea is the following: A program with continuous output takes a density operator as input, gives a classical value as classical output, and—if it terminates—leaves the system in another state (a new density operator). The semantics of a program can therefore be considered as a measurement process. The semantics of unitary transformations, branches and loops are worked out. A special elementary subprogram `print` allows the output of classical data before the program's termination.

4 Conclusions

Concerning the design of both practical as well as formal programming languages there is still much room for development.

For practical languages, the main challenges are probably the creation of powerful constructs to abstract from the very machine-oriented model of quantum circuits towards higher-level programming, as well as the development of compilers and optimisers that get the best results out of the available resources.

For formal languages, challenges are expressive, easy-to-read languages with well-defined semantics, as well as support in reasoning about programs. Such support can consist of powerful mathematical laws that aid the manual proof, as well as mechanised proof systems where the computer carries out parts of the analysis.

Ultimately, the two approaches could be unified, resulting in *one* language that is both suited for actual programming and execution of programs and protocols as well as for analysing these programs and giving rigorous proofs of their correctness.

References

AD04. Pablo Arrighi and Gilles Dowek. Operational semantics for formal tensorial calculus. In Peter Selinger, editor, *2nd International Workshop on Quantum Programming Languages*, pages 21–38, 2004. Online available at `http://quasar.mathstat.uottawa.ca/~selinger/qpl2004/PDFS/03Arrighi-Dowek.pdf`.

BB87. Tommaso Bolognesi and Ed Brinksma. Introduction to the iso specification language lotos. *Comput. Netw. ISDN Syst.*, 14(1):25–59, 1987. Online available at `http://lotos.site.uottawa.ca/ftp/pub/Lotos/Intro/BB-LotosTutorial.pdf`.

BCS03. S. Bettelli, T. Calarco, and L. Serafini. Toward an architecture for quantum programming. *Eur. Phys. J. D*, 25:181–200, 2003. Online available at http://www.edpsciences.org/articles/epjd/abs/2003/09/d02124/d02124.html.

Bet. S. Bettelli. A repository for an implementation of the Q language. http://sra.itc.it/people/serafini/qlang/.

BOM04. M. Ben-Or and D. Mayers. General security definition and composability for quantum & classical protocols, September 2004. Online available at http://xxx.lanl.gov/abs/quant-ph/0409062.

BPW04. Michael Backes, Birgit Pfitzmann, and Michael Waidner. Secure asynchronous reactive systems. IACR ePrint Archive, March 2004. Online available at http://eprint.iacr.org/2004/082.ps.

BS04. Alexandru Baltag and Sonja Smets. The logic of quantum programs. In Peter Selinger, editor, *2nd International Workshop on Quantum Programming Languages*, pages 39–56, 2004. Online available at http://quasar.mathstat.uottawa.ca/~selinger/qpl2004/PDFS/04Baltag-Smets.pdf.

Cle00. Richard Cleve. An introduction to quantum complexity theory. In C. Macchiavello, G. M. Palma, and A. Zeilinger, editors, *Collected Papers on Quantum Computation and Quantum Information Theory*, pages 103–127. World Scientific, 2000. Online available at http://www.cpsc.ucalgary.ca/~cleve/pubs/intro_complexity_qph.ps.

Dij76. Edsger W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

GN04. Simon J. Gay and Rajagopal Nagarajan. Communicating quantum processes. In Peter Selinger, editor, *2nd International Workshop on Quantum Programming Languages*, pages 91–107, 2004. Online available at http://quasar.mathstat.uottawa.ca/~selinger/qpl2004/PDFS/07Gay-Nagarajan.pdf.

Kni96. E. Knill. Conventions for quantum pseudocode. Technical Report LAUR-96-2724, Los Alamos National Laboratory, 1996. Online available at http://www.eskimo.com/~knill/cv/reprints/knill:qc1996e.ps.

LJ04. Marie Lalire and Philippe Jorrand. A process algebraic approach to concurrent and distributed quantum computation: operational semantics. In Peter Selinger, editor, *2nd International Workshop on Quantum Programming Languages*, pages 109–126, 2004. Online available at http://quasar.mathstat.uottawa.ca/~selinger/qpl2004/PDFS/08Lalire-Jorrand.pdf.

Mil89. R. Milner. *Communication and concurrency.* Prentice-Hall, Inc., 1989.

MM99. Carroll Morgan and Annabelle McIver. pGCL: formal reasoning for random algorithms. *South African Computer Journal*, 22:14–27, 1999. Online available at http://web.comlab.ox.ac.uk/oucl/research/areas/probs/pGCL.ps.gz.

MMS03. P. Mateus, J. Mitchell, and A. Scedrov. Composition of cryptographic protocols in a probabilistic polynomial-time process calculus. In Roberto Amadio and Denis Lugiez, editors, *Concurrency Theory, Proceedings of CONCUR '03*, volume 2761 of *Lecture Notes in Computer Science*, pages 327–349. Springer-Verlag, 2003. Online available at http://wslc.math.ist.utl.pt/ftp/pub/MateusP/03-MMS-seccomp.pdf.

MPW92a. Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, i. *Inf. Comput.*, 100(1):1–40, 1992. Online available at http://www.lfcs.inf.ed.ac.uk/reports/89/ECS-LFCS-89-85/index.html.

MPW92b. Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, ii. *Inf. Comput.*, 100(1):41–77, 1992. Online available at http://www.lfcs.inf.ed.ac.uk/reports/89/ECS-LFCS-89-86/index.html.

NC00. M. Nielsen and I. Chuang. *Quantum Computation and Quantum Information.* Cambridge University Press, Cambridge, 2000.

Öme. Bernhard Ömer. Qcl - a programming language for quantum computers. http://tph.tuwien.ac.at/~oemer/qcl.html.

Öme03a. Bernhard Ömer. Quantum programming in QCL. Master's thesis, TU Vienna, 2003. Online available at http://tph.tuwien.ac.at/~oemer/doc/quprog.pdf.

Öme03b. Bernhard Ömer. *Structured Quantum Programming.* PhD thesis, TU Vienna, 2003. Online available at http://tph.tuwien.ac.at/~oemer/doc/structquprog.pdf.

SCA$^+$04. K. Svore, A. Cross, A. Aho, I. Chuang, and I. Markov. Toward a software architecture for quantum computing design tools. In Peter Selinger, editor, *2nd International Workshop on Quantum Programming Languages*, pages 145–162, 2004. Online available at `http://quasar.mathstat.uottawa.ca/~selinger/qpl2004/PDFS/10Svore-Cross-Aho-Chuang-Markov.pdf`.

Sel04. Peter Selinger. Towards a quantum programming language. *Mathematical Structures in Computer Science*, 14(4):527–586, 2004. Online available at `http://quasar.mathstat.uottawa.ca/~selinger/papers/qpl.ps.gz`.

Sho94. Peter W. Shor. Algorithms for quantum computation: Discrete logarithms and factoring. In *35th Annual Symposium on Foundations of Computer Science, Proceedings of FOCS 1994*, pages 124–134. IEEE Computer Society, 1994.

SV04. Peter Selinger and Benoît Valiron. A lambda calculus for quantum computation with classical control, November 2004. Preprint, online available at `http://quasar.mathstat.uottawa.ca/~selinger/papers/qlambda.ps.gz`.

SZ00. J. W. Sanders and Paolo Zuliani. Quantum programming. In *MPC '00: Proceedings of the 5th International Conference on Mathematics of Program Construction*, Springer LNCS, pages 80–99. Springer-Verlag, 2000. Online available at `http://web.comlab.ox.ac.uk/oucl/research/areas/probs/qp.ps.gz`.

Unr04a. Dominique Unruh. Classical control in quantum programs. 5th European QIPC Workshop, Roma, September 2004. Poster, online available at `http://iaks-www.ira.uka.de/home/unruh/publications/unruh04classical.html`.

Unr04b. Dominique Unruh. Classical control in quantum programs. Manuscript, 2004.

Unr04c. Dominique Unruh. Simulatable security for quantum protocols, September 2004. Online available at `http://arxiv.org/ps/quant-ph/0409125`.

Val04. Benoît Valiron. Quantum typing. In Peter Selinger, editor, *2nd International Workshop on Quantum Programming Languages*, pages 163–178, 2004. Online available at `http://quasar.mathstat.uottawa.ca/~selinger/qpl2004/PDFS/11Valiron.pdf`.

vT04. André van Tonder. A lambda calculus for quantum computation. *SIAM Journal on Computing*, 33(5):1109–1135, 2004. Online available at `http://epubs.siam.org/sam-bin/dbq/article/43216`.

Wil03. Alexander Wilce. Quantum logic and probability theory. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy.* Metaphysics Research Lab, CSLI, Stanford University, Spring 2003. Online available at `http://plato.stanford.edu/archives/spr2003/entries/qt-quantlog/`.

Zul04. Paolo Zuliani. Non-deterministic quantum programming. In Peter Selinger, editor, *2nd International Workshop on Quantum Programming Languages*, pages 179–195, 2004. Online available at `http://quasar.mathstat.uottawa.ca/~selinger/qpl2004/PDFS/12Zuliani.pdf`.