International Journal of Quantum Information
Vol. 6, Supplement (2008) 733–738

World Scientific
www.worldscientific.com

SEMANTICS OF QUANTUM PROGRAMMING LANGUAGE LANQ

HYNEK MLNAŘÍK

Laboratory of Quantum Information Processing and Cryptography,
Faculty of Informatics, Masaryk University,
Brno, Czech Republic
hmlnarik@mail.muni.cz

Received 27 February 2008

We show a memory model of an imperative concurrent quantum programming language LanQ. The memory model is used to specify the shape of semantical structure upon which the language operational semantics is defined. We also outline the language abilities in the area of formal verification on an example implementation of teleportation protocol.

Keywords: LanQ; quantum programming language; quantum process algebra; semantics.

1. Introduction

LanQ is the first imperative quantum programming language capable of handling concurrency — new process creation and interprocess communication.[5] The language combines the best of the worlds of quantum process algebras (see e.g. Refs. 3 and 4) and imperative quantum programming languages (see e.g. Ref. 6). LanQ has well-defined operational semantics and its type soundness has been proved. Any quantum resource is proved to be owned by at most one process; a channel can be shared by at most two processes — a sender and a receiver. These features make the language usable for formal reasoning about quantum algorithms and protocols. To reason about programs, we must formally define their behavior by defining language semantics. The present paper shows the essential parts of LanQ semantics and their possible further usage for formal verification.

This paper is organized as follows. In Sec. 2, we describe memory model of LanQ language. The semantics is outlined in Sec. 3 and followed by a sketch of the proof of implementation correctness in Sec. 4.

2. Memory Model

In this section, we introduce memory model of the LanQ programming language. The memory model is crucial for the later definition of language semantics as it determines the shape of the semantical structure.

As we work both with duplicable data-classical- and nonduplicable data-resources-, we have two types of memory: *local* and *system*. All processes manage their own memory — a *local process memory* where duplicable values are stored. System manages the *system memory* used for storing nonduplicable resources. Processes cannot access the system memory directly, they work with resources by means of references to the system memory. Note that this strict separation of system memory and local process memories can be found in all formalisms which deal with both classical and quantum data and build their semantical structure on top of classical data.[3,4] This approach is suitable whenever it is assumed that the program uses mostly classical operations. Dual approach which builds the semantical structure on top of quantum data is also possible.[7]

The memory model consists of four layers as depicted in Fig. 1. From the bottom, the first layer is a *system memory layer* where the quantum systems and channels are physically stored in separate containers. The next one is a *value layer* which stores classical values and references to the system memory. One layer up is a *structured reference layer* which stores references to the value layer. The structured reference layer is particularly used for handling compound quantum systems. To topmost layer is the *variable layer* which contains variable names. The connection between neighboring layers is done via partial functions described later in Sec. 3.

The memory model depicted in Fig. 1 shows the situation where several processes are running concurrently. Individual process memories are shown in the middle level boxes. Variable names are shown in circles in the top level; duplicable variables are shown in white circle, nonduplicable in black. The variables v (integer), ρ and ψ (quantum systems), and c (channel) belonging to one process refer to structured references stored in a process local memory. The structured references are shown as dashed-circle containers. They refer to values — a value 3 in the case of the variable v container, a reference to a quantum system with index 1 in the case of the variables ρ and ψ container, and a reference to a channel in the case of the variable c container.

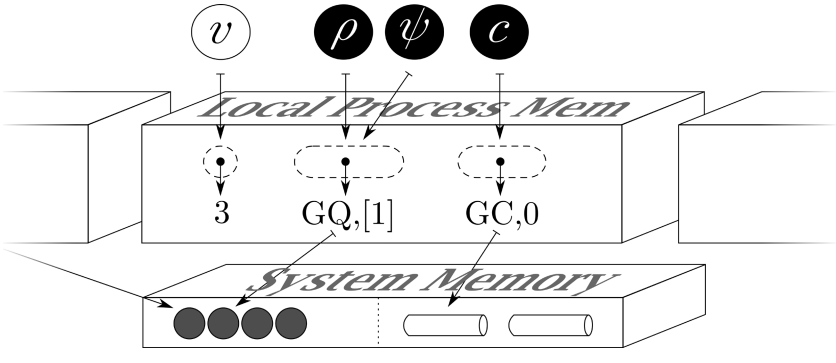

Fig. 1. Memory model of LanQ.

3. Semantics

In this section, we describe the structure used for definition of the operational semantics and outline the transition relation.

Configuration. The semantics is defined operationally as a transition relation on configurations and mixed configurations. The structure of configurations is given by the structure of the memory model.

A *configuration* is a tuple $[gs \,|\, ls_1 \parallel \cdots \parallel ls_n]$ where:

- gs is a *global* part of the configuration where the resources are stored, i.e. it represents the state of the system layer from the memory model. The global part is a tuple $gs = ((\rho, L), C)$ where:
  - (a) (ρ, L) stores the state of quantum systems where ρ is a density matrix of current quantum state. To enable LanQ to handle quantum states of arbitrary dimensions, L stores a list of quantum system dimensions, and
  - (b) C is a list of allocated communication channels.
- ls_i-s represent *local* states of individual processes. Each local process state ls_i is a tuple (lms_i, vp_i, ts_i) where:
  - (a) lms_i represents the state of the process memory, i.e. the mapping from the structured reference layer to the value layer,
  - (b) vp_i stores variable properties and is used to handle variable scope. It represents the mapping from the variable layer to the structured reference layer, and
  - (c) ts_i represents a stack of terms to be evaluated.

A *mixed configuration* is a probabilistic distribution over configurations C_i written as:

$$\boxplus_{i=1}^{q} p_i \bullet C_i \quad \text{where} \sum_{i=1}^{q} p_i = 1,$$

and q is the number of probabilistic branches, each running with probability p_i.

A *terminal configuration* is a configuration $C = [gs \,|\, ls_1 \parallel \cdots \parallel ls_n]$ where $ls_i = (lms_i, vp_i, ts_i)$ such that for all i, the term stack ts_i is either empty or contains only one term which denotes either a runtime error or a value.

Transition relation. The transition relation $\longrightarrow$ is used to transform a configuration to a mixed configuration and vice versa. In one step, only one of the processes stored in the configuration gets evolved. The choice of the evolving process is nondeterministic. Due to the probabilistic nature of measurements, the transition relation must also deal with probabilistic transitions. From Ref. 2 we know that whenever a probabilistic and a nondeterministic choice are to be resolved simultaneously, we must define which choice is resolved first. However, we have taken the same approach as in Ref. 3; the choice of transition type is resolved deterministically and

can never happen simultaneously: a configuration that is not mixed is evolved by the transition $\longrightarrow$ to a mixed configuration. For a mixed configuration, the next transition is probabilistic because no other transition is defined for it.

A feature of LanQ semantics which simplifies the implementation and might be useful for the future development of other types of semantics is that the choice of the evaluated process is the only source of nondeterminism what significantly simplifies the implementation of the language. The evaluation of individual processes is probabilistic and deterministic. This is in contrast with the existing quantum process algebras[3,4] where nondeterminism also arises from possible concurrent usage of a resource by two or more processes.[a]

The program execution proceeds as follows:

(1) The execution starts from the configuration **start**:

$$\mathbf{start} = [(((1), []), []) \mid (lms_0, vp_0, (\ main() \ ))]$$

where lms_0 stands for an empty local memory state what corresponds to empty memory and vp_0 for empty variable properties what corresponds to the state where no variables are known to the running program. The term $main()$ expresses an invocation of the *main* method which must be defined in the program and from which the program execution starts.

(2) Let $C = [gs \mid ls_1 \parallel \cdots \parallel ls_n]$ where $ls_i = (lms_i, vp_i, ts_i)$ is a local process state. If C is a terminal configuration, finish. Otherwise nondeterministically choose a number p between 1 and n. Perform the next transition according to the top element of the term stack ts_p. Note that this transition always transforms the configuration C to a mixed configuration.

(3) Let $M = \boxplus_{i=1}^{q} p_i \bullet C_i$ be a mixed configuration. Choose probabilistically one of the configurations C_i and perform a probabilistic transition to this configuration. Continue with the step 2.

Because of the space constraints, we omit the full definition of transition rules and kindly refer the reader to Ref. 5 instead.

4. Verification of Implementation Correctness

In this section, we show a sketch of the proof of correctness of implementation of the well-known teleportation protocol by Bennett *et al.*[1] This program demonstrates all features of LanQ: both quantum and classical data manipulation as well as multiprocess capabilities.

The program, as shown in Fig. 2, consists of three methods — *main*() which is the starting method of the program, and methods *angela*() and *bert*() which

[a]Note that we can take advantage of nondeterministic behaviour: It can be used e.g. to simply catch server environment serving requests from multiple clients where nondeterminism is used to resolve which request came from which client. It can be also used to model possible eavesdropping of communication.

```
void main() {
        qbit ψ_A,ψ_B;
        channel[int] c withends [c_0,c_1];
        ψ_EPR aliasfor [ψ_A, ψ_B];
        ψ_EPR = createEPR();
        c = new channel[int]();
        fork bert(c_1, ψ_A);
        angela(c_0, ψ_B);
}

void angela(channelEnd[int] c_0, qbit auxTeleportState) {
        int r;
        qbit φ;
        φ = computeSomething();
(*) ⟶   r = measure (BellBasis, φ, auxTeleportState);
        send (c_0, r);
}

qbit bert(channelEnd[int] c_1, qbit ξ) {
        int i;
        i = recv (c_1);
        if (i == 1) {
                σ_z(ξ);
        } else if (i == 2) {
                σ_x(ξ);
        } else if (i == 3) {
                σ_x(ξ); σ_z(ξ);
        }
(**) ⟶  doSomethingElse(ξ);
}
```

Fig. 2. Teleportation protocol implementation in LanQ.

represent a sender and a receiver of the teleported state, respectively. The detailed description of the program can be found in Ref. 5.

As the semantics is well-defined, we can now *formally* verify that the example program is implemented according to the specification and really teleports the state from Angela to Bert as expected.

The place in the program marked with $(*)$ is the place just before a joint measurement of the teleported particle and one particle from the EPR pair is performed. This is the last place where we can see the original teleported particle state. When we execute the program to this point, we get the following configuration $C_{(*)}$:

$$C_{(*)} = \left(\left(\left(\rho_{EPR} \otimes \begin{pmatrix} m_{00} & m_{01} \\ m_{10} & m_{11} \end{pmatrix}, [2,2,2] \right), [c] \right) \middle| (lms_a, vp_a, ts_a) \parallel (lms_b, vp_b, ts_b) \right)$$

where ρ_{EPR} is a density matrix of EPR state $(1/\sqrt{2})(|00\rangle + |11\rangle)$ and $\rho_\phi = (m_{ij})$ is a density matrix of the state ϕ.

Right after the measurement, we get a mixed configuration of four configurations which correspond to four different probabilistic branches given by the measurement result. After further evolution of the probabilistic branches, in each of them we get up to the place marked by $(**)$ where the configuration $C_{(**),p}$ is the following:

$$C_{(**),p} = \left(\left(\left(\begin{pmatrix} m_{00} & m_{01} \\ m_{10} & m_{11} \end{pmatrix} \otimes \rho_p, [2,2,2] \right), [c] \right) \middle| (lms_a^p, vp_a^p, ts_a^p) \parallel (lms_b^p, vp_b^p, ts_b^p) \right)$$

where p is the index of the probabilistic branch, $\rho_\xi = (m_{ij})$ is the density matrix of the state ξ, and ρ_p is the density matrix representing the two other qubits state.

In all cases we obtain that the state ρ_ψ of the teleported qubit ϕ before the actual teleportation is the same as the state ρ_ξ of the qubit ξ after finishing the teleportation protocol. This result is indeed obtained according to the semantics, hence we have formally proved that the implementation of the teleportation protocol is correct.

5. Conclusion

We have shown the memory model of LanQ programming language and sketched the language semantics. Using this semantics, we have shown the idea of the proof of formal verification of implementation correctness. The semantics in full detail, theorems about LanQ, other examples, and the implementation of the LanQ simulator, which is the first implementation of an imperative quantum programming language with a sound theoretical basis, can be found at the webpage http://lanq.sf.net/.

Acknowledgment

This work has been done under the financial support by the grants MSM0021622419 and GAČR 201/07/0603. The author would like to thank to Simon Gay, Philippe Jorrand, Rajagopal Nagarajan and Nick Papanikolaou for fruitful discussions that helped him in establishing LanQ semantics.

References

1. C. H. Bennett, G. Brassard, C. Crépeau, R. Jozsa, A. Peres and W. K. Wootters, Teleporting an unknown quantum state via dual classical and Einstein-Podolsky-Rosen channels, *Phys. Rev. Lett.* **70**(13) (March 1993) 1895–1899.
2. D. Cazorla, F. Cuartero, V. V. Ruiz, F. L. Pelayo and J. J. Pardo, Algebraic theory of probabilistic and nondeterministic processes, *J. Log. Algebra Program.* **55**(1–2) (2003) 57–103.
3. S. J. Gay and R. Nagarajan, Types and typechecking for communicating quantum processes, *Math. Struct. Comput. Sci.* **16** (2006) 375–406.
4. M. Lalire, *Développement d'une notation algorithmique pour le calcul quantique*, Ph.D. thesis, Grenoble Institute of Technology (2006).
5. H. Mlnařík, Quantum programming language LanQ, Ph.D. thesis, Faculty of Informatics, Masaryk University (2007). Also available from http://lanq.sf.net/.
6. B. Ömer, Structured quantum programming, Ph.D. thesis, TU Vienna (2003).
7. P. Selinger, Towards a quantum programing language, *Math. Struct. Comput. Sci.* **14**(4) (2004) 527–586.