Introduction to LanQ – an Imperative Quantum Programming Language

Hynek Mlnařík*

July 21, 2006

Abstract

LanQ is an imperative high-level quantum programming language that allows combining of quantum and classical computations in the way of *classical control, quantum data.* Moreover, it offers tools for communication and parallel running of programs. In the paper, we introduce the language.

1 Introduction

Quantum programming languages started to be developed when there was a need to write quantum algorithms in a machine-readable form. Several languages have been developed so far, mainly λ-calculi [vT03a, vT03b, SV05] and functional languages [Sel04, AG04].

Even though interest in quantum programming languages is still growing, only few of the developed are imperative: QCL developed by Ömer [Ôme00, Ôme03] or Q language based on C++ developed by Bettelli, Calarco and Serafini [BCS01]. Another language qGCL developed by Sanders and Zuliani [Zul01] can be viewed as an imperative language as well as a specification language.

Programming languages mentioned in the previous paragraph do not support communication and process management. This restricts their use to implementation of quantum algoritms. Implementation of multiparty quantum protocols is not possible in these languages at all.

Implementation of multiparty quantum protocols requires a language that can handle both quantum phenomena and process management – allowing processes to run in parallel and to mutually communicate. Interprocess

*Faculty of Informatics, Masaryk University, Brno, CZ. e-mail: `xmlnarik@fi.muni.cz`

communication and process parallelism are studied in the field of quantum process algebras [LJ04, Lal05, GN04, GN05]. Even though quantum algorithms can be implemented in these algebras, they are primarily designed for process description. The syntax is therefore unclear to programmers.

The limitation of current quantum programming languages mentioned earlier led the author to development of a new language – LanQ. Contrary to previously mentioned programming languages, LanQ offers also tools for process creation and classical/quantum interprocess communication. Therefore, it is possible to implement multiparty quantum protocols in LanQ.

LanQ is an imperative high-level language with syntax similar to that of C language that allows expression of both quantum and classical computations. It offers tools for creation of new processes and interprocess communication. In the following text, we will describe the language.

2 Informal introduction

We begin our introduction to LanQ by an example of implementation of a well-known multiparty quantum protocol – teleportation [BBC+93]. In LanQ, it can be written as a program shown in the Figure 1.

We now briefly describe the program. Three methods, **main**, **angela** and **bert**, are defined. Method **main()** can be invoked with no parameters. It returns no value what can be seen from the void in front of method name. Method **angela()** has to be invoked with two parameters – a channel end of a channel that can be used to send values of type int and one qubit. It also returns no value. Method **bert()** takes the other end of the channel and another qubit. This method returns a value of type qbit.

The **main()** method declares variables used in the method body in its first three lines. The type of variables ψ_A, ψ_B is qbit. Variable ψ_{EPR} is declared to be alias for $\psi_A \otimes \psi_B$. Channel c capable of sending integer numbers is declared on the next line. The individual channel ends are named c_0 and c_1.

On next lines, method **main** invokes method **createEPR()** which creates an EPR-pair and stores reference to the created pair into variable ψ_{EPR}. After that, a new channel is allocated and assigned to the variable c. The next command causes the running process to split into two. One of the processes continues its run and invokes method **angela()**. The second process starts its run from the method **bert()**.

The **angela()** method receives one channel end and one qubit as arguments. After declaring variables r and ϕ, it assigns a result of running of

```
void main() {
    qbit $\psi_A, \psi_B$;
    $\psi_{EPR}$ aliasfor [$\psi_A, \psi_B$];
    channel[int] $c$ withends [$c_0, c_1$];

    $\psi_{EPR}$ = createEPR();
    $c$ = new channel[int]();
    fork bert($c_0$, $\psi_B$);

    angela($c_1$, $\psi_A$);
}

void angela(channelEnd[int] $c_1$, qbit $ats$) {
    int $r$;
    qbit $\phi$;

    $\phi$ = doSomething();
    $r$ = measure ($BellBasis$, $\phi, ats$);
    send ($c_1$, r);
}
```

```
qbit bert(channelEnd[int] $c_0$, qbit $stto$) {
    int $i$;

    $i$ = recv ($c_0$);
    if ($i$ == 0) {
        op$B_0$($stto$);
    } else if ($i$ == 1) {
        op$B_1$($stto$);
    } else if ($i$ == 2) {
        op$B_2$($stto$);
    } else {
        op$B_3$($stto$);
    }
    doSomethingElse($stto$);
}
```

Figure 1: Teleportation implemented in LanQ

the method **doSomething()** to ϕ. Then it measures qubits ϕ and ats in the Bell basis, assigns the result of the measurement to the variable r and sends it over the channel end c_0.

The **bert()** method receives one channel end and one qubit as arguments. After declaring variable i, it receives an integer value from the channel end c_1 and assigns it to the variable i. Depending on the received value it applies one of the operators opB_0, opB_1, opB_2 and opB_3 onto the qubit *stto*. Finally, it invokes method **doSomethingElse()** and passes the variable *stto* as an argument of this method.

The reason for distinguishing between channels and channel ends is described in the Section 4.2.

3 Syntax

Syntax of LanQ is similar to the syntax of C language. Therefore we will not describe it in more detail, it can be found in Appendix A. We will mention three extensions of the syntax – syntax of process creation, quantum variable aliasing and channel declaration.

LanQ has a primitive for creating a new process: reserved word **fork**. Its usage is the following:

$$\textbf{fork methodName}(arg_1, arg_2, \dots);$$

This construction creates a new process and runs given method with given arguments as a root method of the new process. This is the way of running multiple processes in parallel.

Quantum variable aliasing is an important part of the language. Using variable alias, it is possible to work either with a compound quantum system or with individual subsystems of that system as needed. The declaration of an compound system ψ_C which is a composition of subsystems $\psi_0, \psi_1, \dots, \psi_N$ is:

$$\psi_C \textbf{ aliasfor } [\psi_0, \psi_1, \dots, \psi_N];$$

A type of a channel for sending values of a type T is $\mathsf{channel[T]}$. When the process needs to access individual channel ends of that channel then the declaration of that channel (named c int the example) must read:

$$\mathsf{channel[T]}\ c \textbf{ withends } [c_0, c_1];$$

Then the process can access both the channel c and its ends c_0 and c_1 after the channel allocation (see Section 5).

4 Process management

4.1 Forking

As LanQ is designed to be a language that can be used for implementation of multiparty protocols, it must provide tool for process management. For creation of a new process, a language primitive **fork** is introduced.[1] Its syntax has been shown in the Section 3.

After forking, the original process where **fork** was invoked continues its run by execution of subsequent statements. **fork** creates a new process which is started from the method specified as an argument of the **fork**. After forking, these two processes run in parallel.

Values of duplicable variables that are passed as arguments of the forked method are available in both processes after forking. On the other hand, values of non-duplicable types passed as arguments to the forked method are not available in the original method after forking.

[1]The idea of process forking is similar to UNIX process creation.

Moreover, passing a channel end to the forked process causes also the corresponding channel to become unavailable in the original process. The reason is that the channel is controlled by a process just till the process controls both its ends.

4.2 Communication

The language offers tools for communication between processes. It is possible for a process to allocate a channel. When two different processes get individual ends of the channel (*eg.* by passing a channel end to a forked process), they can communicate over these channel ends.

The reason for distinguishing between channels and channel ends is the following. It is natural to require that one channel is controlled by at most two processes at one time – a sender and a receiver. However, it is hard to ensure this property. But it is simple to ensure that only one channel end can be controlled by just one process at one time. The papers [KPT96, GH05] are relevant to this approach.

LanQ offers two primitives for controlling a channel end – **send** for sending a value over a corresponding channel and **recv** for receiving a value from that channel. These primitives are synchronous, *ie.* **recv** delays program run until there is a value received from the channel and **send** delays a program run until the sent value is received.

5 Types

Types are very important part of the language. Types help to check well-formedness of the program before its run. The other benefit of using types is a possibility to distinguish whether a value can be copied or not. Depending on the latter criterion we separate types into two groups: duplicable and non-duplicable.

- *Duplicable* (*ie.* non-linear) types are types of classical values, *eg.* bits, integers, booleans *etc.*. They are characterized by the fact that any value of a duplicable type can be exactly copied.

- *Non-duplicable* (*ie.* linear) types are types of resources – quantum values, channels and channel ends. Any value of a non-duplicable type can be neither perfectly cloned nor copied at all. Any non-duplicable resource must be allocated before it is used.

We require quantum types to be non-duplicable because cloning is impossible due to no-cloning theorem.

We require a channel to be controlled by at most two processes at one time – a sender and a receiver. Handling individual channel ends separately gives us a similar requirement – channel ends are not duplicable as we want any channel end to be owned by exactly one process. If we allowed copying, some process could allocate a channel, create a number of copies of its channel ends and these ends distribute among its children processes. This would break our requirement that one channel is controlled by at most two processes at one time.

6 Abstract syntax

Abstract syntax defines internal representation of the programs. Semantics of a programming language is defined on the level of its abstract syntax. For this reason, abstract syntax is much more interesting than specific syntax.

Semantics of LanQ language can be found in the article [Mln06].

Symbols of abstract syntax of LanQ are *expressions* and *statements.*

Definition 6.1. Expression *is any phrase derivable from nonterminal* expr*.*

Definition 6.2. Statement *is any phrase derivable from nonterminal* code*.*

6.1 Expressions

LanQ recognizes expressions listed below:

- Constant – a predefined constant, *eg.* true, false, 0, 1,...
- Variable – a variable name
- Bracketted expression – an expression enclosed in brackets
- Allocation – allocation of a quantum system or a channel
- Assignment – assignment of a value to a variable
- Method call – invocation of an method
- Measurement – measurement of a quantum variable
- Receiving a value from a channel end

Expressions evaluate to a value which is called a *return value.* LanQ deals with expressions in a by-reference manner where the reference determines memory type (classical, quantum, channel, channel end) and position of the value in the memory. For this reason, whenever it is possible, both the value and the reference to that value is returned to the evaluator. A special reference `none` is reserved for the cases when the return value is not stored in memory and hence cannot be referred in the time of evaluation.

Assignment, method call, measurement, allocation and receiving from a channel can also act as statements. In that case a return value of such expressions is discarded.

6.2 Statements

LanQ recognizes following statements:

- Skip statement
- Block – block statement defines sequence of statements
- Variable declaration
- If – conditional execution of code
- While – conditional looping of code
- Return – return from a method, possibly returning some value from the method
- Fork – forking a new process
- Sending a value over a channel end

Statements are characterized by the fact that they do not evaluate to any return value. If an expression acts as a statement, then its return value is simply discarded.

Acknowledgements

I would like to thank Philippe Jorrand and Rajagopal Nagarajan for an invaluable discussion and Simon Gay for pointing me to the articles [KPT96] and [GH05]. I would also like to thank Stefano Bettelli and Michael Nölle for their comments.

7 Conclusion and future work

LanQ imperative programming language was introduced in the paper. It was shown that it can be used for implementation of both quantum algorithms and quantum multiparty protocols. Ideas of process management and interprocess communication were shown.

The operational semantics of the language is actually being developed. After that, denotational and categorical semantics will be developed. Development of the semantics is crucial for further development of optimization in LanQ.

The optimization techniques can be among other used to safely turn pieces of classical code that fulfil certain criteria into purely quantum code by a compiler/interpreter (*eg.* by conversion of **if** statements into controlled gates).

A Syntax

In this appendix, we define the syntax of the LanQ language.

The reserved words of the language are written in **bold** and the identifier names are in *italic*. The syntax of these names is left to the implementation. Grammar is given in nondeterministic extended Backus-Naur form (EBNF). The root of grammar is the nonterminal program.

A.1 Code

program	::=	method+
code	::=	assignment **;** | methodCall **;** | quantumOp **;** | measurement **;** | fork **;** | send **;** | recv **;** | return **;** | block | if | while | **;**
methodCall	::=	*methodName* **(** methodParams? **)**
methodParams	::=	expr (**,** expr)*
assignment	::=	*variableName* **=** expr
measurement	::=	**measure (** *basisName* (**,** *variableName*)+ **)**
expr	::=	indivExpr (op expr)? | **new** nonDupType **()**
indivExpr	::=	*value* | *variableName* | **(** expr **)** | recv | assignment | methodCall | measurement
op	::=	$+$ | $-$ | $\otimes$ | *etc.*
quantumOp	::=	*quantumOpName* **(** methodParams? **)**

A.2 Block structure

method	::=	methodHeader block
block	::=	**{** varDeclaration code* **}**
methodHeader	::=	type *methodName* **(** methodDeclParamList? **)**
methodDeclParamList	::=	methodDeclParam (**,** methodDeclParam)*
methodDeclParam	::=	nonVoidType *paramName*
varDeclaration	::=	(oneVarDecl **;**)*
oneVarDecl	::=	nonVoidType *variableName* (**,** *variableName*)* |
		channelType *variableName* (**withends**
		[*variableName* **,** *variableName*])? |
		variableName **aliasfor**
		[*variableName* (**,** *variableName*)*]

A.3 Program flow

fork	::=	**fork** methodCall
return	::=	**return** expr

A.4 Conditionals and loops

if	::=	**if (** expr **)** code (**else** code)?
while	::=	**while (** expr **)** code

A.5 Communication

recv	::=	**recv (** expr **)**
send	::=	**send (** expr **,** expr **)**

A.6 Types

type	::=	**void** | nonVoidType
nonVoidType	::=	dupType | nonDupType
dupType	::=	**int** | **boolean** | *etc.*
nonDupType	::=	**channelEnd** [nonVoidType] | channelType | qType
channelType	::=	**channel** [nonVoidType]
qType	::=	qBasicType ($\otimes$ qType)?
qBasicType	::=	**qbit** | **qtrit** | *etc.*

References

[AG04] Thorsten Altenkirch and Jonathan Grattage. A functional quantum programming language. *quant-ph/0409065*, 2004.

[BBC$^+$93] C. H. Bennett, G. Brassard, C. Crépeau, R. Jozsa, A. Peres, and W. K. Wooters. Teleporting an unknown quantum state via dual classical and Einstein-Podolsky-Rosen channels. *Physical Review Letters*, (70):1895–1899, 1993.

[BCS01] S. Bettelli, T. Calarco, and L. Serafini. Toward an architecture for quantum programming, 2001.

[GH05] Simon Gay and Malcolm Hole. Subtyping for session types in the pi calculus. *Acta Informatica*, 42(2):191–225, 2005.

[GN04] Simon J. Gay and Rajagopal Nagarajan. Communicating Quantum Processes. *quant-ph/0409052*, 2004.

[GN05] Simon J. Gay and Rajagopal Nagarajan. Communicating quantum processes. In *POPL '05: Proceedings of the 32nd ACM Symposium on Principles of Programming Languages*, pages 145–157, 2005.

[KPT96] Naoki Kobayashi, Benjamin C. Pierce, and David N. Turner. Linearity and the pi-calculus. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 358–371, New York, NY, USA, 1996. ACM Press.

[Lal05] Marie Lalire. A Probabilistic Branching Bisimulation for Quantum Processes. *quant-ph/0508116*, 2005.

[LJ04] Marie Lalire and Philipe Jorrand. A process-algebraic approach to concurrent and distributed quantum computation: operational semantics. *quant-ph/0407005*, 2004.

[Mln06] Hynek Mlnařík. LanQ operational semantics. 2006. Available online from `http://www.fi.muni.cz/~xmlnarik/lanq/opsemantics.pdf`.

[Öme00] B. Ömer. Quantum programming in QCL. Master's thesis, TU Vienna, 2000.

[Öme03] B. Ömer. *Structured Quantum Programming.* PhD thesis, TU Vienna, 2003.

[Sel04] Peter Selinger. Towards a quantum programming language. *Mathematical. Structures in Comp. Sci.*, 14(4):527–586, 2004.

[SV05] Peter Selinger and Benoît Valiron. A Lambda Calculus for Quantum Computation with Classical Control. *Lecture Notes in Computer Science*, 3461 / 2005:354–368, 2005.

[vT03a] André van Tonder. A Lambda Calculus for Quantum Computation. *quant-ph/0307150*, 2003.

[vT03b] André van Tonder. Quantum computation, categorical semantics and linear logic. *quant-ph/0312174*, 2003.

[Zul01] Paolo Zuliani. *Quantum Programming.* PhD thesis, University of Oxford, 2001.