LanQ – Operational Semantics of Quantum Programming Language LanQ

by

Hynek Mlnařík

FI MU Report Series FIMU-RS-2006-10

December 2006

Publications in the FI MU Report Series are in general accessible via WWW:

`http://www.fi.muni.cz/reports/`

Further information can obtained by contacting:

Faculty of Informatics
Masaryk University
Botanická 68a
602 00 Brno
Czech Republic

Operational Semantics of Quantum Programming Language LanQ

Hynek Mlnařík[*]
Faculty of Informatics, Masaryk University
Brno, Czech Republic
xmlnarik@fi.muni.cz

December 21, 2006

Abstract

We present new imperative quantum programming language LanQ which was designed to support combination of quantum and classical programming and basic process operations – process creation and interprocess communication. The language can thus be used for implementing both classical and quantum algorithms and protocols. Its syntax is similar to that of C language what makes it easy to learn for existing programmers. In this paper, we present operational semantics of the language. We provide an example run of a quantum random number generator.

1 Introduction

Quantum computing is a young branch of computer science. Its power lies in employing quantum phenomena in computation. These laws are different to those that rule classical world: Quantum systems can be entangled. Quantum evolution is reversible. One can compute exponentially many values in one step.

Quantum phenomena were successfully used for speeding up a solution of computationally hard problems like computing discrete logarithm or factorisation of integers [Sho94]. Another successful application of quantum phenomena in computing, namely in cryptography, is secure quantum key generation [BB84, Eke91, Ben92]. Quantum key

[*]This work has been supported by the grants No. 201/04/1153 and MSM0021622419.

generation overcomes the classical in the fact that its security relies on the laws of nature, while classical key generation techniques rely on computational hardness of solving some problems. A nice example of quantum phenomena usage is a teleportation of an unknown quantum state [BBC$^+$93].

For the formal description of quantum algorithms and protocols, several quantum programming languages and process algebras have already been developed. Some of them support handling quantum data only, however most of them allow combining of quantum and classical computations. Obtaining classical data from quantum systems is done by *measurement* which is probabilistic by its nature. This implies that quantum formalisms must be able to to handle probabilistic computation.

Existing formalisms are usually based on existing classical programming languages and process algebras. From imperative languages, we should mention Ömer's QCL (Quantum Computation Language, [Öme00]) whose syntax is based on that of C language; Betteli, Calarco and Serafini's Q language built as an extension of C++ basic classes [BCS01]. However, semantics of these imperative languages is not formally defined formally. Zuliani's qGCL (quantum Guarded Command Language, [Zul01]) based on pGCL (probabilistic Guarded Command Language) has denotational semantics defined but does not support recursion.

Many of developed languages are functional because of relatively straightforward definition of its operational semantics. Van Tonder developed a quantum λ-calculus [vT03]; quantum λ-calculus was also developed by Selinger and Valiron [SV05]; Selinger proposed functional static-typed quantum flow-chart programming language QFC and its text form QPL [Sel04]. Another functional programming language QML was developed by Altenkirch and Grattage [AG04] and refined into nQML in [LGP06].

Quantum process algebras differ to classical ones in the way they handle quantum systems. The main issue solved here is that they must guarantee that any quantum system is accessible by only one process at one time (because of the no-cloning theorem [WZ82]). The quantum process algebras QPAlg by Lalire and Jorrand [LJ04, JL04, Lal05] and CQP by Gay and Nagarajan [GN04, GN05, GN06] can describe both classical and quantum interaction and evolution of processes. QPAlg was inspired by CCS, originally using nontyped channels for interprocess communication. Recently [Lal06], Lalire has added support for fixpoint operator and typed channels to QPAlg.

The presented language LanQ is an imperative quantum programming language. It allows combination of quantum and classical computations to be expressed. Moreover, it has features of quantum process algebras – it supports new process creation and interprocess communication. Its syntax is similar to the syntax of C language. The formal definition of syntax can be found in Appendix B. In the present paper, we define its internal syntax and operational semantics.

The paper is structured as follows: we start with an example of an program written in LanQ. Internal syntax is defined in Section 3. The main result is operational semantics which is presented in Section 4. An example of a simple program execution can be found in Appendix A.

2 Informal introduction

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

```
void main() {
        qbit ψ_A, ψ_B;
        ψ_EPR aliasfor [ψ_A, ψ_B];
        channel[int] c withends [c_0, c_1];

        ψ_EPR = createEPR();
        c = new channel[int]();
        fork bert(c_0, ψ_B);

        angela(c_1, ψ_A);
}

void angela(channelEnd[int] c_0, qbit ats) {
        int r;
        qbit φ;

        φ = doSomething();
        r = measure (BellBasis, φ, ats);
        send (c_0, r);
}
```

```
int bert(channelEnd[int] c_1, qbit stto) {
        int i;

        i = recv (c_1);
        if (i == 0) {
                opB_0(stto);
        } else if (i == 1) {
                opB_1(stto);
        } else if (i == 2) {
                opB_2(stto);
        } else {
                opB_3(stto);
        }
        doSomethingElse(stto);
        return i;
}
```

Figure 1: Teleportation implemented in LanQ

We now briefly describe the program. In LanQ, a program is a set of methods. Three methods, **main**, **angela** and **bert**, are defined. The control is passed to a method called **main()** when the program is run. This method can be invoked with no parameters. It returns no value what can be seen from the word void in front of the 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 (*ie.* a quantum bit). It also returns no value. Method **bert()** takes a channel and a qubit and returns a value of type int.

Method **main()** 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 an alias for a compound system $\psi_A \otimes \psi_B$. 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 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 method **bert()**.

Method **angela()** receives one channel end and one qubit as arguments. After declaring variables r and ϕ, it assigns a result of running of method **doSomething()** to ϕ. Then it measures qubits ϕ and ats in the Bell basis, assigns the result of the measurement to variable r and sends it over the channel end c_0.

Method **bert()** 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 variable i. Depending on the received value it applies one of the operators opB_0, opB_1, opB_2 and opB_3 onto qubit $stto$. Then, it invokes method **doSomethingElse()** and passes variable $stto$ as an argument of this method. Finally, it returns the value of variable i to the caller.

3 Internal syntax

In this section, we define the internal syntax of LanQ.

Using the concrete syntax, a LanQ program is written as a set of method declarations. This notation does not allow direct execution of the program. Hence we need a representation of the program execution – a syntax that allows us evaluation of a program by means of rewriting program terms. The rewriting rules are then presented in the Section 4 where operational semantics is defined.

The internal syntax is defined in the Figure 2. We need the following basic syntactic entities: numbers (N), lists (L), recursive lists (RL), references (R), constants (C), identifiers (I), types (T) and values $(\mathbf{v})$. The processes (P) consist of statements (S) or expressions (E). Several expressions are classified as promotable expressions (PE) – expressions that can act as statements when postfixed by semicolon. For the evaluation of subexpressions we need a concept of a *hole* $(\bullet)$ which stands for the awaited result of subexpression evaluation. The syntactic entities Sc (resp. Ec) represent partially evaluated statements (resp. expressions) whose subexpression is being evaluated, *ie.* they represent *evaluation contexts*.

Remark 3.1. *For the sake of clarity, we use the following notation in the rule body. We denote by $\bar{\mathsf{S}}$ an abbreviation of BNF rule body "$(\mathsf{S})*$", and by $\widetilde{\mathsf{E}}$ an abbreviation of "$(\mathsf{E}\ (,\ \mathsf{E})*)?$".*

$$
\begin{array}{lll}
\mathsf{N} & ::= & 0 \mid 1 \mid \ldots \\
\mathsf{L} & ::= & [] \mid [\widetilde{\mathsf{N}}] \\
\mathsf{RL} & ::= & \mathsf{L} \mid [\widetilde{\mathsf{RL}}] \\
\mathsf{R} & ::= & \mathsf{none} \mid (\textbf{Classical},\mathsf{N}) \mid (\textbf{Quantum},\mathsf{RL}) \mid (\textbf{Channel},\mathsf{N}) \mid \\
 & & (\textbf{ChannelEnd}_{\mathbf{0}},\mathsf{N}) \mid (\textbf{ChannelEnd}_{\mathbf{1}},\mathsf{N}) \mid (\textbf{GQuantum},\mathsf{L}) \mid (\textbf{GChannel},\mathsf{N}) \\
\mathsf{C} & ::= & \textbf{true} \mid \textbf{false} \mid \bot \mid \ldots \mid \mathsf{R} \mid \ldots \\
\mathrm{I} & ::= & x \mid y \mid z \mid \ldots \mid + \mid - \mid \ldots \\
\mathsf{T} & ::= & \mathsf{void} \mid \mathsf{int} \mid \mathsf{qbit} \mid \mathsf{channel}[\mathsf{T}] \mid \mathsf{channelEnd}[\mathsf{T}] \mid \mathsf{T} \otimes \mathsf{T} \mid \ldots \\
\mathbf{v} & ::= & <\mathsf{R},\mathsf{C}> \\
 & & \\
\mathsf{PE} & ::= & \textbf{new}\ \mathsf{T}() \mid \mathrm{I} = \mathsf{E} \mid \mathrm{I}(\widetilde{\mathsf{E}}) \mid \textbf{measure}(\widetilde{\mathsf{E}}) \mid \textbf{recv}(\mathsf{E}) \\
\mathsf{E} & ::= & \mathsf{C} \mid \mathrm{I} \mid \mathbf{v} \mid (\mathsf{E}) \mid \mathsf{PE} \\
\mathsf{VD} & ::= & \mathsf{T}\ \widetilde{\mathrm{I}}; \mid \mathsf{channel}[\mathsf{T}]\ \mathrm{I}\ \textbf{withends}[\mathrm{I},\mathrm{I}]; \mid \mathrm{I}\ \textbf{aliasfor}\ [\widetilde{\mathrm{I}}]; \\
\mathsf{S} & ::= & ; \mid \mathsf{PE}; \mid \circ_{\mathsf{L}} \mid \{\bar{\mathsf{B}}\} \mid \textbf{if}\ (\mathsf{E})\ \mathsf{S}\ \textbf{else}\ \mathsf{S} \mid \textbf{while}\ (\mathsf{E})\ \mathsf{S} \mid \\
 & & \circ_{\mathsf{M}} \mid \textbf{return}; \mid \textbf{return}\ \mathsf{E}; \mid \textbf{fork}\ \mathrm{I}(\widetilde{\mathsf{E}}); \mid \textbf{send}(\mathsf{E},\mathsf{E}); \\
\mathsf{B} & ::= & \mathsf{VD} \mid \mathsf{S} \\
\mathsf{P} & ::= & \mathbf{0} \mid (\mathsf{P} \parallel \mathsf{P}) \mid \mathsf{S} \mid \mathsf{E} \\
 & & \\
\mathsf{Ec} & ::= & \mathrm{I} = \bullet \mid \mathrm{I}(\widetilde{\mathbf{v}},\bullet,\widetilde{\mathsf{E}}) \mid \textbf{measure}(\widetilde{\mathbf{v}},\bullet,\widetilde{\mathsf{E}}) \mid \textbf{recv}(\bullet) \\
\mathsf{Sc} & ::= & \bullet; \mid \textbf{if}\ (\bullet)\ \mathsf{S}\ \textbf{else}\ \mathsf{S} \mid \textbf{fork}\ \mathrm{I}(\widetilde{\mathbf{v}},\bullet,\widetilde{\mathsf{E}}); \mid \textbf{send}(\bullet,\mathsf{E}); \mid \textbf{return}\ \bullet;
\end{array}
$$

Figure 2: Internal syntax

Before a method can be invoked to be run, we must rewrite its body so that it is derivable using internal syntax rules. Fortunately, the method bodies derived using concrete syntax and internal syntax rules differ only in the following:

- In internal representation, all **if** statements have **else** part, *ie.* a statement **if** (E) P is rewritten to **if** (E) P **else** ; where **;** denotes a skip statement,
- In internal representation, all operators are written in the prefix notation and seen as a method call, *ie.* $E \odot F$ is converted to $\odot(E, F)$.

Obviously, there is an algorithm which rewrites any method body derived using concrete syntax to the internal representation.

An example of a block written using concrete syntax and its representation in internal syntax is shown in the Figure 3.

```
{
        int r;
        qbit φ;

        φ = doSomething();
        r = measure (BellBasis, φ, ats);
        send (c0, r);
        if (r == 0) {
                measure (StdBasis, φ);
        }
}
```

(a)

```
{
        int r;
        qbit φ;

        φ = doSomething();
        r = measure (BellBasis, φ, ats);
        send (c0, r);
        if (== (r, 0)) {
                measure (StdBasis, φ);
        } else ;
}
```

(b)

Figure 3: Block derived using concrete syntax (a) and the same block converted to internal syntax (b).

4 Operational semantics

In this section, we introduce operational semantics of LanQ programming language.

4.1 Notation

We use the following notation in the text:

- Set of natural numbers with zero: $\mathbb{N}_0 = \mathbb{N} \cup \{0\}$
- Let S be a set, $\perp \notin S$. $S_\perp = S \cup \{\perp\}$
- Let S be a set. S-list $s = [s_1, \ldots, s_n]$ is a list where $n \in \mathbb{N}_0$ and $s_1, \ldots, s_n \in S$

- Let S be a set. Set of finite S-lists: $S_{[]} = \{s \mid s \text{ is a finite } S\text{-list}\}$
- Length of a list L: $|L|$
- Concatenation of two lists: $[l_{1,1}, \dots, l_{1,n}] \cdot [l_{2,1}, \dots, l_{2,m}] = [l_{1,1}, \dots, l_{1,n}, l_{2,1}, \dots, l_{2,m}]$
- Set of items of a list $L = [l_1, \dots, l_n]$: $\mathrm{set}(L) = \{l_1, \dots, l_n\}$

Let S be a set. We define a *recursive S-list* recursively as:

- any S-list $[s_1, \dots, s_k]$ is a recursive S-list for any $k \in \mathbb{N}_0$
- $[e_1, \dots, e_m]$ is a recursive S-list for any $m \in \mathbb{N}_0$ if $e_1, \dots, e_m$ are recursive S-lists

The set of finite recursive S-lists is denoted by $S_{[\circlearrowright]}$.

For example, $[[[1, 2, 3], [2, 3]], [1]]$ and $[]$ are recursive $\mathbb{N}$-lists.

A linearization function $\mathrm{linearize} : S_{[\circlearrowright]} \to S_{[]}$ which converts a recursive S-list into an S-list can be now defined:

$$\begin{aligned}
\mathrm{linearize}(s) &\overset{\mathrm{def}}{=} s \text{ if } s \in S_{[]} \\
\mathrm{linearize}([e_1, \dots, e_m]) &\overset{\mathrm{def}}{=} \mathrm{linearize}(e_1) \cdot \ldots \cdot \mathrm{linearize}(e_m) \text{ where } e_1, \dots, e_m \in S_{[\circlearrowright]}
\end{aligned}$$

For a set $S_\perp$ with a special element $\perp \in S_\perp$ we define a function $\mathrm{linearize}_\perp : (S_\perp)_{[\circlearrowright]} \to (S_\perp)_{[]} \cup \{\perp\}$ as:

$$\mathrm{linearize}_\perp(s) \overset{\mathrm{def}}{=} \begin{cases} \mathrm{linearize}(s) = [s_1, \dots, s_n] & \text{if } s_i \neq \perp \text{ for } 1 \leq i \leq n \\ \perp & \text{otherwise} \end{cases}$$

Let S be a set, we define $\mathrm{set}_{[\circlearrowright]} : S_{[\circlearrowright]} \to S$ as $\mathrm{set}_{[\circlearrowright]} \overset{\mathrm{def}}{=} \mathrm{set} \circ \mathrm{linearize}$. We also define function for getting length of a recursive list $|-|_{[\circlearrowright]} : S_{[\circlearrowright]} \to \mathbb{N}$ as $|l|_{[\circlearrowright]} \overset{\mathrm{def}}{=} |\mathrm{linearize}(l)|$.

A *memory reference* specifies position of a value in memory. We distinguish *global* and *local* references:

- References to global channel storage: $\mathtt{Ref}_{\mathtt{GCh}} = \{\perp\} \cup (\{\mathtt{GChannel}\} \times \mathbb{N}_\perp)$.
- References to global quantum storage: $\mathtt{Ref}_{\mathtt{GQ}} = \{\perp\} \cup (\{\mathtt{GQuantum}\} \times \mathbb{N}_{[]})$.
- References to local classical value storage: $\mathtt{Ref}_{\mathtt{Cl}} = \{\mathsf{none}\} \cup (\{\mathtt{Classical}\} \times \mathbb{N}_\perp)$.
- References to local quantum systems reference storage:
  $\mathtt{Ref}_{\mathtt{Q}} = \{\mathsf{none}\} \cup (\{\mathtt{Quantum}\} \times (\mathbb{N}_\perp)_{[\circlearrowright]})$.

- References to local channel reference storage: $\mathtt{Ref}_{\mathtt{Ch}} = \{\textsf{none}\} \cup (\{\mathtt{Channel}\} \times \mathbb{N}_{\perp})$.
- References to local channel end reference storage:
  $\mathtt{Ref}_{\mathtt{ChE}} = \{\textsf{none}\} \cup (\{\mathtt{ChannelEnd}_0, \mathtt{ChannelEnd}_1\} \times \mathbb{N}_{\perp})$.

We define $\textsf{none}$ to be a special reference that refers to no value. We define sets $\mathtt{Ref}_G = \mathtt{Ref}_{GCh} \cup \mathtt{Ref}_{GQ}$ of global references, $\mathtt{Ref}_L = \mathtt{Ref}_{Cl} \cup \mathtt{Ref}_Q \cup \mathtt{Ref}_{Ch} \cup \mathtt{Ref}_{ChE}$ of local references and $\mathtt{Ref} = \mathtt{Ref}_G \cup \mathtt{Ref}_L$. A memory reference is therefore an element from the set $\mathtt{Ref}$. We denote by $\mathtt{Ref}_{nd} = \mathtt{Ref}_Q \cup \mathtt{Ref}_{Ch} \cup \mathtt{Ref}_{ChE}$ the set of references to non-duplicable values.

We define a countable set $\mathtt{Types}$ of types of classical values. Among others, $\textsf{void} \in \mathtt{Types}$, $\textsf{boolean} \in \mathtt{Types}$, $\textsf{real} \in \mathtt{Types}$, $\textsf{int} \in \mathtt{Types}$ and in particular, the type of references $\textsf{Ref}$ corresponding to the set $\mathtt{Ref}$ belongs to $\mathtt{Types}$. We denote by $\mathrm{val}(\textsf{T})$ a set of values of type $\textsf{T}$.

We define a set $\mathtt{Values}$ as a set of values of all types:

$$\mathtt{Values} = \bigcup_{\textsf{T} \in \mathtt{Types}} \mathrm{val}(\textsf{T}).$$

4.2 Configuration

Configuration of the abstract machine which realizes operational semantics is composed of two basic parts – *global* and *local*. Global part of the configuration stores information about quantum state of the whole system and relations between channels and their ends. Local part of the configuration stores information about individual processes – state of their classical memory, variables and term to be evaluated.

A configuration is written as $[gs \,|\, ls_1 \parallel \cdots \parallel ls_n]$. Global part of a configuration is represented by gs. Local configurations of n processes running in parallel are represented by $ls_1 \parallel \cdots \parallel ls_n$, where ls_j represents local configuration of j-th process.

A configuration can evolve by a probabilistic transition to a mixture of configurations. A probabilistic mixture of configurations is written as:

$$\boxplus_{i=1}^{q} p_i \bullet [gs_i \,|\, ls_{i,1} \parallel \cdots \parallel ls_{i,n}].$$

It represents configurations of q different computational branches, each of them will run with probability p_i.

Configuration is composed of the following components:

- Global part of the configuration is a pair (Q, C) where:

- Q describes quantum part of the configuration.

  In the present paper, we represent quantum state of the system by a pair (ρ, L) of a finite density matrix and a finite list of natural numbers. The list represents dimensions of individual quantum subsystems. The order of list elements is given by order of quantum system allocations.

- C represents channel part of the configuration.

  Channels and their ends are stored as pairs (c_0, c_1) written as $c_0 \models\!\!\!=\!\!\!\dashv c_1$ where c_0, c_1 represent channel ends.

• Local part of the configuration is a tuple $(\mathit{lms}, \mathit{vp}, \mathit{ts})$ where:

  - Local memory configuration is a tuple $\mathit{lms} = (\mathit{lms}_{Cl}, \mathit{lms}_{Q}, \mathit{lms}_{Ch}, \mathit{lms}_{ChE})$ which stores state of classical memory and references to non-duplicable resources which are available to the process:

    * $\mathit{lms}_{Cl} : \mathrm{Ref}_{Cl} \rightharpoonup \mathit{Values}_{\perp}$ is a partial function which returns a classical value stored at given position in memory. Set of all such partial functions is denoted by LMS_{Cl}.
    * $\mathit{lms}_{Q} : \mathrm{Ref}_{Q} \rightharpoonup (\mathrm{Ref}_{GQ})_{\perp}$ returns a reference to a global quantum memory. Set of all such partial functions is denoted by LMS_{Q}.
    * $\mathit{lms}_{Ch} : \mathrm{Ref}_{Ch} \rightharpoonup (\mathrm{Ref}_{GCh})_{\perp}$ returns a reference to the channel. Set of all such partial functions is denoted by LMS_{Ch}.
    * $\mathit{lms}_{ChE} : \mathrm{Ref}_{ChE} \rightharpoonup (\mathrm{Ref}_{GCh})_{\perp}$ returns a reference to the channel corresponding to the channel end. Set of all such partial functions is denoted by LMS_{ChE}.

    To simplify notation, we formally regard lms itself as a partial function. Note that $\mathrm{Ref}_{GQ} \subseteq \mathit{Values}$ and $\mathrm{Ref}_{GCh} \subseteq \mathit{Values}$. Now we can define $\mathit{lms} = (\mathit{lms}_{Cl}, \mathit{lms}_{Q}, \mathit{lms}_{Ch}, \mathit{lms}_{ChE}) : \mathrm{Ref} \rightharpoonup \mathit{Values}_{\perp}$ where:

$$\mathit{lms}(r) = \begin{cases} \mathit{lms}_{Cl}(r) & \text{if } r \in \mathrm{Ref}_{Cl} \\ \mathit{lms}_{Q}(r) & \text{if } r \in \mathrm{Ref}_{Q} \\ \mathit{lms}_{Ch}(r) & \text{if } r \in \mathrm{Ref}_{Ch} \\ \mathit{lms}_{ChE}(r) & \text{if } r \in \mathrm{Ref}_{ChE} \\ \perp & \text{if } r = \textsf{none} \end{cases}$$

    Set of all such quadruples lms is denoted by LMS.

- Variable properties are stored in a list $\mathtt{vp}$ of lists of partial function tuples $\mathsf{f} = (\mathsf{f}_{\mathsf{var}}, \mathsf{f}_{\mathsf{ch}}, \mathsf{f}_{\mathsf{qa}})$ where:
  - $\mathsf{f}_{\mathsf{var}} : \mathsf{Names} \rightharpoonup \mathsf{Ref}$ maps a variable name to a reference.
  - $\mathsf{f}_{\mathsf{ch}} : \mathsf{Names} \rightharpoonup \mathsf{Names} \times \mathsf{Names}$ maps a channel variable name to variable names representing ends of the channel.
  - $\mathsf{f}_{\mathsf{qa}} : \mathsf{Names} \rightharpoonup \mathsf{Names}_{[]}$ maps a variable name representing a quantum system to variable names that represent its subsystems.

  We define $\mathsf{VarProp}$ to be a set of all finite lists of lists of such partial function tuples f. An element of $\mathsf{VarProp}$ represents stack state during a program run. The inner list (whose elements are concatenated by $\circ_L$) follows the block structure of method, the outer list (elements concatenated by $\circ_G$) represents the sequence of method calling. The creation of stack is shown in the Subsection 4.3.

  We define an *empty variable properties tuple* as a partial function tuple $\mathsf{f} = (\mathsf{f}_{\mathsf{var}}, \mathsf{f}_{\mathsf{ch}}, \mathsf{f}_{\mathsf{qa}})$ where $\mathrm{dom}(\mathsf{f}_{\mathsf{var}}) = \mathrm{dom}(\mathsf{f}_{\mathsf{ch}}) = \mathrm{dom}(\mathsf{f}_{\mathsf{qa}}) = \emptyset$.
- Term stack $\mathtt{ts}$: stack of terms to be evaluated. We use such a notation where individual stack items are underlined for readability. The empty term stack is denoted by ε.

4.3 Variable stack

Variable stack stores properties of variables. In LanQ, the properties of a variable are: value of the variable (the value is a memory reference), ends of a channel represented by the variable (if the variable represents a channel with channel end variables declared using **withends** primitive) and quantum systems that the variable is composed of (if the variable is declared to be a composition of other quantum systems using **aliasfor** primitive). Therefore we have a tuple of three partial functions that represent variable properties: $\mathsf{f} = (\mathsf{f}_{\mathsf{var}}, \mathsf{f}_{\mathsf{ch}}, \mathsf{f}_{\mathsf{qa}})$.

The variable stack must also respect the variable scope:

- A variable can be accessed only from within the block where it was declared. This is ensured by using a list of tuples separated by $\circ_L$, where a new tuple is appended to the list when a block is started and removed when the block ends.

- Only variables from the running method are accessible to the running method. This is ensured by using a list of lists of tuples separated by $\circ_G$, where the tuples are appended to the list when a method is invoked and removed when a method ends.

This can be seen from the following example. The right side gives a method, the left side gives the variable stack as the method is run. For the sake of brevity, we use a symbol $\Diamond$ for an empty variable properties tuple (*ie.* for ([],[],[])) and $\Box$ for an empty list of variable properties tuples (*ie.* for []):

1	int a(int c)	$[vp_G \circ_G [\Box \circ_L ([c \mapsto r_c], [], [])]]$
2	{	$[vp_G \circ_G [[\Box \circ_L ([c \mapsto r_c], [], [])] \circ_L \Diamond]]$
3	bool b;	$[vp_G \circ_G [[\Box \circ_L ([c \mapsto r_c], [], [])] \circ_L ([b \mapsto \text{none}], [], [])]]$
4	b = **true**;	$[vp_G \circ_G [[\Box \circ_L ([c \mapsto r_c], [], [])] \circ_L ([b \mapsto r_b], [], [])]]$
5	**if** (b) {	$[vp_G \circ_G [[[\Box \circ_L ([c \mapsto r_c], [], [])] \circ_L ([b \mapsto r_b], [], [])] \circ_L \Diamond]]$
6	int i;	$[vp_G \circ_G [[[\Box \circ_L ([c \mapsto r_c], [], [])] \circ_L ([b \mapsto r_b], [], [])] \circ_L ([i \mapsto \text{none}], [], [])]]$
	⋮	
7	}	$[vp_G \circ_G [[\Box \circ_L ([c \mapsto r_c], [], [])] \circ_L ([b \mapsto r_b], [], [])]]$
8	}	$[vp_G \circ_G [\Box \circ_L ([c \mapsto r_c], [], [])]]$

When a method a is called, a list of variable properties tuples $[\Box \circ_L ([c \mapsto r_c], [], [])]$ is appended to the caller variable stack vp_G ($[vp_G \underline{\circ_G [\Box \circ_L ([c \mapsto r_c], [], [])]]}$, line 1). In this appended list, method parameters values are passed to the called method; in our case, the value of the method parameter c is stored in the memory in the place referred by the reference r_c. On line 2 a new block is started, therefore a new empty variable properties tuple is appended ($[[\Box \circ_L ([c \mapsto r_c], [], [])] \underline{\circ_L \Diamond}]$). On the next line, a variable b is declared and written onto the head element of the local stack. On line 4, b is assigned a value **true** which is stored into memory in a place referred by r_b. On line 5, a new block is started, therefore a new empty variable properties tuple is appended. On line 6, a new integer variable i is declared and stored into the stack head. On lines 7 and 8, two blocks end and the appropriate local stacks are discarded.

4.4 Variable stack and memory handling functions

In this subsection, we define functions for memory and variable handling.

Let $f : X \rightharpoonup Y$ be a partial function. We define a partial function $f[x \mapsto y] : X \rightharpoonup Y \cup \{y\}$:

$$f[x \mapsto y](a) \overset{\text{def}}{=} \begin{cases} y & \text{if } a = x \text{ and } f(x) \text{ is defined} \\ f(a) & \text{otherwise} \end{cases}$$

Note that the $f[x \mapsto y](x)$ is defined iff $f(x)$ is defined.

Let $f : X \rightharpoonup Y$ be a partial function. We define a partial function $f[x \mapsto y]_+ : X \cup \{x\} \rightharpoonup Y \cup \{y\}$:

$$f[x \mapsto y]_+(a) \overset{\text{def}}{=} \begin{cases} y & \text{if } a = x \\ f(a) & \text{otherwise} \end{cases}$$

Note that the $f[x \mapsto y]_+(x)$ is defined even if $f(x)$ is not defined.

A *partial function tuple* is a tuple $f = (f_0, \ldots, f_n)$ where $f_0, \ldots, f_n$ are partial functions.

We will need a list of partial function tuples to represent scope of variables in a method. We define *list of partial function tuples* recursively as:

- $[] = \square$ is a list of partial function tuples.
- $[L \circ_L f]$ is a list of partial function tuples if L is a list of partial function tuples and f is a partial function tuple.

We also define a *replacement* $L[x \mapsto y]_i$ and an *update* $L[x \mapsto y]_{+,i}$ of outermost x in an i-th partial function of a list of partial function tuples L as:

$$[L \circ_L (f_0, \ldots, f_n)][x \mapsto y]_i \overset{\text{def}}{=} \begin{cases} [L \circ_L (f_0, \ldots, f_i[x \mapsto y], \ldots, f_n)] & \text{if } f_i(x) \text{ is defined} \\ [L[x \mapsto y]_i \circ_L (f_0, \ldots, f_n)] & \text{otherwise} \end{cases}$$

$$\square[x \mapsto y]_i \overset{\text{def}}{=} \square$$

$$[L \circ_L (f_0, \ldots, f_n)][x \mapsto y]_{+,i} \overset{\text{def}}{=} [L \circ_L (f_0, \ldots, f_i[x \mapsto y]_+, \ldots, f_n)]$$

$$\square[x \mapsto y]_{+,i} \overset{\text{def}}{=} \square$$

To capture variable scope between method calls we define a *list of lists of partial function tuples* recursively as:

- $[] = \blacksquare$ is a list of lists of partial function tuples.
- $[L_1 \circ_G L_2]$ is a list of lists of partial function tuples if L_1 is a list of lists of partial function tuples and L_2 is a list of partial function tuples.

In the case of lists of lists of partial function tuples, a *replacement* $L[x \mapsto y]_i$ and an *update* $L[x \mapsto y]_{+,i}$ of mapping of x in the outermost list of partial function tuples is defined as:

$$\begin{aligned} [L_1 \circ_G L_2][x \mapsto y]_i &\overset{\text{def}}{=} [L_1 \circ_G L_2[x \mapsto y]_i] \\ \blacksquare[x \mapsto y]_i &\overset{\text{def}}{=} \blacksquare \end{aligned}$$

$$\begin{aligned} [L_1 \circ_G L_2][x \mapsto y]_{+,i} &\overset{\text{def}}{=} [L_1 \circ_G L_2[x \mapsto y]_{+,i}] \\ \blacksquare[x \mapsto y]_{+,i} &\overset{\text{def}}{=} \blacksquare \end{aligned}$$

The list of partial function tuples represents variable stack of a method run. The list of lists of partial function tuples represents variable stack of the whole program run.

Next, we define a function varRef which returns a reference to value of given variable, a function $\mathsf{chanEnds}$ which returns variable names that represent ends of a given channel and a function $\mathsf{aliasSubsyst}$ which returns names of variables that a compound system of a given system is composed of.

A function $\mathsf{varRef} : \mathsf{Names} \times \mathsf{VarProp} \to \mathsf{Ref}_L$ for getting references to values from a variable name and a list of lists of partial function tuples is defined as:

$$\begin{aligned} \mathsf{varRef}(x, [L_1 \circ_G L_2]) &\overset{\text{def}}{=} \mathsf{varRef}_L(x, L_2) \\ \mathsf{varRef}(x, \blacksquare) &\overset{\text{def}}{=} \text{none} \end{aligned}$$

where $\mathsf{varRef}_L : \mathsf{Names} \times \mathsf{VarProp}_L \to \mathsf{Ref}_L$ is a function for getting references to values from a variable name and a list of partial function tuples:

$$\begin{aligned} \mathsf{varRef}_L(x, [L \circ_L (f_{var}, f_{ch}, f_{qa})]) &\overset{\text{def}}{=} \begin{cases} f_{var}(x) & \text{if } f_{var}(x) \text{ is defined} \\ \mathsf{varRef}_L(x, L) & \text{otherwise} \end{cases} \\ \mathsf{varRef}_L(x, \square) &\overset{\text{def}}{=} \text{none} \end{aligned}$$

Analogously, we define a function $\mathit{chanEnds} : \mathit{Names} \times \mathit{VarProp} \to (\mathit{Names} \times \mathit{Names})_\perp$ for getting variable names that represent individual ends of a given channel from a name of the channel and a list of lists of partial function tuples:

$$\begin{aligned}\mathit{chanEnds}(x, [L_1 \circ_G L_2]) &\overset{\text{def}}{=} \mathit{chanEnds}_L(x, L_2)\\ \mathit{chanEnds}(x, \blacksquare) &\overset{\text{def}}{=} \perp\end{aligned}$$

where $\mathit{chanEnds}_L : \mathit{Names} \times \mathit{VarProp}_L \to (\mathit{Names} \times \mathit{Names})_\perp$ is a function for getting variable names that represent individual ends of a given channel from a name of the channel and a list of partial function tuples:

$$\begin{aligned}\mathit{chanEnds}_L(x, [L \circ_L (f_{var}, f_{ch}, f_{qa})]) &\overset{\text{def}}{=} \begin{cases} f_{ch}(x) & \text{if } f_{ch}(x) \text{ is defined}\\ \mathit{chanEnds}_L(x, L) & \text{otherwise}\end{cases}\\ \mathit{chanEnds}_L(x, \square) &\overset{\text{def}}{=} \perp\end{aligned}$$

Next, we define a function $\mathit{aliasSubsyst} : \mathit{Names} \times \mathit{VarProp} \to (\mathit{Names}_{[]})_\perp$ for getting a list of variable names that represent individual parts of a compound system from a name of the compound system and a list of lists of partial function tuples:

$$\begin{aligned}\mathit{aliasSubsyst}(x, [L_1 \circ_G L_2]) &\overset{\text{def}}{=} \mathit{aliasSubsyst}_L(x, L_2)\\ \mathit{aliasSubsyst}(x, \blacksquare) &\overset{\text{def}}{=} \perp\end{aligned}$$

where $\mathit{aliasSubsyst}_L : \mathit{Names} \times \mathit{VarProp}_L \to (\mathit{Names}_{[]})_\perp$ is a function for getting a list of variable names that represent individual parts of a compound system from a name of the compound system and a list of partial function tuples:

$$\begin{aligned}\mathit{aliasSubsyst}_L(x, [L \circ_L (f_{var}, f_{ch}, f_{qa})]) &\overset{\text{def}}{=} \begin{cases} f_{qa}(x) & \text{if } f_{qa}(x) \text{ is defined}\\ \mathit{aliasSubsyst}_L(x, L) & \text{otherwise}\end{cases}\\ \mathit{aliasSubsyst}_L(x, \square) &\overset{\text{def}}{=} \perp\end{aligned}$$

We define a function $\mathit{localAliasedVars} : \mathit{VarProp} \to \mathit{Names}_{[]}$ for getting the list of all names of variables representing compound systems in the local variable list from a list of lists of partial function tuples:

$$\begin{aligned} \mathsf{localAliasedVars}([L_1 \circ_G L_2]) &\overset{\text{def}}{=} \mathsf{localAliasedVars}_L(L_2) \\ \mathsf{localAliasedVars}(\blacksquare) &\overset{\text{def}}{=} [] \end{aligned}$$

where $\mathsf{localAliasedVars}_L : \mathsf{Names} \rightarrow \mathsf{Names}_{[]}$ is a function for getting the list of all names of variables representing compound systems in the local variable list from a list of partial function tuples:

$$\begin{aligned} \mathsf{localAliasedVars}_L([L \circ_L (f_{var}, f_{ch}, f_{qa})]) &\overset{\text{def}}{=} [qa_1, \dots, qa_n] \cdot \mathsf{localAliasedVars}_L(L) \\ &\qquad \text{where } \{qa_1, \dots, qa_n\} = \mathsf{dom}(f_{qa}) \\ \mathsf{localAliasedVars}_L(\square) &\overset{\text{def}}{=} [] \end{aligned}$$

We define a function $\mathsf{unmap}_{nd} : \mathsf{Ref}_L \times \mathsf{LMS} \rightarrow \mathsf{LMS}$ for unmapping a reference to a non-duplicable value from local memory:

$$\mathsf{unmap}_{nd}((\mathsf{refType}, n), \mathsf{lms}) \overset{\text{def}}{=} \begin{cases} \mathsf{unmap}_Q(n, \mathsf{lms}) & \text{if } \mathsf{refType} = \mathsf{Quantum} \\ \mathsf{unmap}_{Ch}(n, \mathsf{lms}) & \text{if } \mathsf{refType} = \mathsf{Channel} \\ \mathsf{unmap}_{ChE}((i, n), \mathsf{lms}) & \text{if } \mathsf{refType} = \mathsf{ChannelEnd}_i \\ \mathsf{lms} & \text{otherwise} \end{cases}$$

where

- Function $\mathsf{unmap}_Q : \mathbb{N}_{[\circlearrowleft]} \times \mathsf{LMS} \rightarrow \mathsf{LMS}$ is defined as:

$$\mathsf{unmap}_Q(n, (\mathsf{lms}_{Cl}, \mathsf{lms}_Q, \mathsf{lms}_{Ch}, \mathsf{lms}_{ChE})) = (\mathsf{lms}_{Cl}, \mathsf{lms}'_Q, \mathsf{lms}_{Ch}, \mathsf{lms}_{ChE}).$$

  where lms'_Q is defined as:

$$\mathsf{lms}'_Q((\mathsf{Quantum}, l)) \overset{\text{def}}{=} \begin{cases} \mathsf{lms}_Q((\mathsf{Quantum}, l)) & \text{if } \mathrm{set}_{[\circlearrowleft]}(n) \cap \mathrm{set}_{[\circlearrowleft]}(l) = \emptyset \\ \bot & \text{otherwise} \end{cases}$$

- Function $\mathsf{unmap}_{Ch} : \mathbb{N} \times \mathsf{LMS} \rightarrow \mathsf{LMS}$ is defined as:

$$\mathsf{unmap}_{Ch}(n, (\mathsf{lms}_{Cl}, \mathsf{lms}_Q, \mathsf{lms}_{Ch}, \mathsf{lms}_{ChE})) = (\mathsf{lms}_{Cl}, \mathsf{lms}_Q, \mathsf{lms}'_{Ch}, \mathsf{lms}'_{ChE}).$$

  where $\mathsf{lms}'_{Ch} \overset{\text{def}}{=} \mathsf{lms}_{Ch}[(\mathsf{Channel}, n) \mapsto \bot]$
  $\mathsf{lms}'_{ChE} \overset{\text{def}}{=} \mathsf{lms}_{ChE}[(\mathsf{ChannelEnd}_0, n) \mapsto \bot, (\mathsf{ChannelEnd}_1, n) \mapsto \bot]$

- Function $\mathtt{unmap}_{\mathtt{ChE}} : \mathbb{N} \times \mathbb{N} \times \mathsf{LMS} \to \mathsf{LMS}$ is defined as:

$$\mathtt{unmap}_{\mathtt{ChE}}(i, n, (\mathtt{lms}_{\mathtt{Cl}}, \mathtt{lms}_{\mathtt{Q}}, \mathtt{lms}_{\mathtt{Ch}}, \mathtt{lms}_{\mathtt{ChE}})) = (\mathtt{lms}_{\mathtt{Cl}}, \mathtt{lms}_{\mathtt{Q}}, \mathtt{lms}'_{\mathtt{Ch}}, \mathtt{lms}'_{\mathtt{ChE}}).$$

$$\begin{aligned} \text{where}\quad \mathtt{lms}'_{\mathtt{Ch}} &\overset{\text{def}}{=} \mathtt{lms}_{\mathtt{Ch}}[(\mathtt{Channel}, n) \mapsto \bot] \\ \mathtt{lms}'_{\mathtt{ChE}} &\overset{\text{def}}{=} \mathtt{lms}_{\mathtt{ChE}}[(\mathtt{ChannelEnd}_i, n) \mapsto \bot] \end{aligned}$$

We extend the function $\mathtt{unmap}_{\mathtt{nd}}$ so that we can also use any set of references as the first argument, *ie.* for any $R = \{r_1, \ldots, r_k\} \subseteq \mathtt{Ref}$ we define:

$$\mathtt{unmap}_{\mathtt{nd}}(R, \mathtt{lms}) \overset{\text{def}}{=} \mathtt{unmap}_{\mathtt{nd}}(r_1, \mathtt{unmap}_{\mathtt{nd}}(r_2, \ldots \mathtt{unmap}_{\mathtt{nd}}(r_k, \mathtt{lms}) \ldots)).$$

4.5 Functions for handling aliasfor constructs

Handling the **aliasfor** construction is a little complicated. Two cases must be handled:

- A quantum variable p, which is possibly used in several **aliasfor** constructs as a subsystem, is assigned a value. Then this variable and all the compound systems using this variable are to be updated.

- A variable q, which is an alias for a compound quantum system, is associated a value. Then all its subsystems must be appropriately modified. However, the subsystems can be also used in several other **aliasfor** constructs as subsystems and all these compound quantum systems are to be updated.

To handle the cases we define four functions (we assume that the assigned variable is q):

- Function $f_{\mathtt{assignQSystem}}$ that updates variable references and local memory state of all variables where q is used as a subsystem including q itself.

- Function $f_{\mathtt{assignQSystemDirect}}$ that updates variable references and local memory state of q itself.

- Function $f_{\mathtt{assignQSystemInAlias}}$ that updates variable references and local memory state of all systems where q is used as a subsystem.

- Function $f_{\mathtt{assignQAlias}}$ that updates variable references and local memory state of all variables that are subsystems of q.

Function $f_{assignQSystem} : LMS_Q \times VarProp \times Names \times Ref_Q \to LMS_Q \times VarProp$ is defined as:

$$
\begin{aligned}
&f_{assignQSystem}(lms_Q, vp, name, value) \overset{\text{def}}{=} (lms_{Q,ret}, vp_{ret})\\
&\text{where} \quad (lms_{Q,0}, vp_0) = f_{assignQSystemDirect}(lms_Q, vp, name, value),\\
&\qquad (lms_{Q,i}, vp_i) = f_{assignQSystemInAlias}(lms_{Q,i-1}, vp_{i-1}, qcs_i, value, l_i) \text{ for all } 1 \le i \le k\\
&\qquad lms_{Q,ret} = lms_{Q,k},\ vp_{ret} = vp_k\\
&\text{given that} \quad QCS = \{qcs \in \text{set}(localAliasedVars(vp)) | \ name \in \text{set}(aliasSubsyst(qcs, vp)\}\\
&\qquad QCS \text{ is indexed by numbers } i \in \mathbb{N} : 1 \le i \le k\\
&\qquad qcs_i \in QCS\\
&\qquad aliasSubsyst(qcs_i, vp) = [qcs_{i,1}, \dots, qcs_{i,m_i}]\\
&\qquad qcs_{i,l_i} = name
\end{aligned}
$$

Function $f_{assignQSystemDirect} : LMS_Q \times VarProp \times Names \times Ref_Q \to LMS_Q \times VarProp$ is defined as:

$$
\begin{aligned}
&f_{assignQSystemDirect}(lms_Q, vp, name, value) \overset{\text{def}}{=} (lms_{Q,ret}, vp_{ret})\\
&\text{where} \quad lms_{Q,ret} = lms_Q[(Quantum, q) \mapsto gq] vp_{ret} = vp[name \mapsto value]_{var}\\
&\text{given that} \quad value = (Quantum, q)\\
&\qquad gq = \begin{cases} \perp & \text{if linearize}_\perp(q) = \perp \\ (GQuantum, \text{linearize}_\perp(q)) & \text{otherwise} \end{cases}
\end{aligned}
$$

Function $f_{assignQSystemInAlias} : LMS_Q \times VarProp \times Names \times Ref_Q \times \mathbb{N} \to LMS_Q \times VarProp$ is defined as:

$$
\begin{aligned}
&f_{assignQSystemInAlias}(lms_Q, vp, name, value, index) \overset{\text{def}}{=} (lms_{Q,ret}, vp_{ret})\\
&\text{where} \quad (lms_{Q,0}, vp_0) = f_{assignQSystemDirect}(lms_Q, vp, name, value),\\
&\qquad (lms_{Q,i}, vp_i) = f_{assignQSystemInAlias}(lms_{Q,i-1}, vp_{i-1}, qcs_i, value, l_i) \text{ for all } 1 \le i \le k\\
&\qquad vp_{ret} = vp[name \mapsto (Quantum, [v'_1, \dots, v'_k]) v'_j = \begin{cases} value & \text{if } j = index \\ v_j & \text{otherwise} \end{cases}\\
&\qquad lms_{Q,ret} = lms_Q[(Quantum, [v'_1, \dots, v'_k]) \mapsto gq]\\
&\text{given that} \quad value = (Quantum, [v_1, \dots, v_k])\\
&\qquad gq = \begin{cases} \perp & \text{if linearize}_\perp(q) = \perp \\ (GQuantum, \text{linearize}_\perp(q)) & \text{otherwise} \end{cases}
\end{aligned}
$$

Function $f_{assignQAlias} : LMS_Q \times VarProp \times Names \times Ref_Q \to LMS_Q \times VarProp$ is defined as:

$$\begin{array}{ll}
\multicolumn{2}{l}{f_{assignQAlias}(lms_Q, vp, name, value) \overset{\text{def}}{=} (lms_{Q,ret}, vp_{ret})} \\
\text{where} & lms_{Q,0} = lms_Q,\ vp_0 = vp \\
& (lms_{Q,i}, vp_i) = f_{assignQAlias}(lms_{Q,i-1}, vp_{i-1}, q_i, (Quantum, v_i)) \text{ for all } 1 \leq i \leq k \\
& lms_{Q,ret} = lms_{Q,k},\ vp_{ret} = vp_k \\
\text{given that} & aliasSubsyst(name, vp) = [q_1, \ldots, q_k] \\
& value = (Quantum, [v_1, \ldots, v_k])
\end{array}$$

4.6 Methods

Methods are reusable parts of the code. One can pass zero or many parameters to the methods and a method can return zero or one return value as its result. A method must be declared before it can be used. A method declaration consists of two parts, a method header and a method body. Method header specifies a name and a type of the method. A sequence of computational steps that describe method computation is specified in the method body. We define a partial function $methodBody$ which given a method name, returns this method's body transformed into internal syntax representation.

In the following example, a method named $mBody$ of type $T_1, \ldots, T_n \to T$ is declared. The parts of the method declaration are annotated on the right side.

$$\begin{array}{ll}
T\ mName(T_1\ a_1, \ldots, T_n\ a_n) & \}\ \text{method header} \\
\left.\begin{array}{l} \{ \\ \quad \text{... statements ...} \\ \} \end{array}\right\} & \text{method body written using concrete syntax}
\end{array}$$

4.7 Internal values

Expressions evaluate to values and possibly to references. Operational semantics uses both values and references so we define *internal value* to be a pair $< ref, val > \in Ref_L \times Values$.

4.8 Transitions

We define operational semantics in terms of the following relations:

- $\longrightarrow_v$ – This defines transitions of expressions to values
- $\longrightarrow_e$ – This defines transitions of expressions to expressions – the order of evaluation is encoded here.
- $\longrightarrow_s$ – This defines transitions of statements to statements, used for execution of a statement.
- $\longrightarrow_r$ – This defines transitions of statements to statements, used for rewriting of abbreviation statement to corresponding statements.
- $\longrightarrow_p$ – This defines transitions of processes.
- $\xrightarrow{p}$ – This defines probabilistic transitions of processes, p is the probability of the transition.

The relations $\longrightarrow_v$, $\longrightarrow_e$, $\longrightarrow_s$, $\longrightarrow_r$ and $\xrightarrow{p}$ define deterministic and probabilistic single process evolution. Nondeterminism is introduced by parallel evolution of processes but there is no nondeterminism in the evolution of individual processes even when run in parallel with other processes. This is an improvement over existing quantum process algebras where it is possible for three or more processes to share one channel. In these algebras, when these processes use the channel simultaneously, the resulting behaviour is nondeterministic. This type of nondeterminism is avoided by using channel ends for communication instead of channels and imposing a constraint that one channel end is owned by exactly one process at one time. This approach was also studied in the context of π-calculus in [KPT96, GH05].

When probabilistic and nondeterministic choice are to be evaluated simultaneously, we must decide which choice is resolved first. We have taken the same approach as many other authors (*eg.* [LJ04, GN04]): the nondeterministic choice must be resoved first.

When we get to the situation when no rule is applicable to the configuration, the configuration becomes *stuck*. Because LanQ is a typed language, many of such situations can be avoided by proving series of standard lemmas in style of Wright and Felleisen ([WF94]). However, there exist some unavoidable cases caused by by-reference handling of variables. For example, a process P can send a qubit referred by variable ψ to other process and then try to measure qubit referred by ψ, but there is none. This way the process configuration becomes stuck. In future, such cases will be handled by runtime errors.

4.9 Processes

Let gs be a global configuration state, lms, lms_i be local memory states and vp, vp_i be variable properties.

We define special processes **start**, **0** and a set of local process configurations $\mathbf{0}_c$ as:

$$\textbf{start} \overset{\text{def}}{=} [(((1), \square), \square) \,|\, ((\square, \square, \square, \square), \blacksquare, \underline{main()})]$$

$$\mathbf{0} \overset{\text{def}}{=} ((((1), \square), \square) \,|\, ((\square, \square, \square, \square), \blacksquare, \varepsilon)$$

$$\mathbf{0_c} \overset{\text{def}}{=} \{(lms, vp, \varepsilon) \mid lms \in LMS, vp \in VarProp\}$$

The *terminal process configuration* is defined as

$$[gs \,|\, (lms_1, vp_1, v_1) \parallel \cdots \parallel (lms_n, vp_n, v_n)] \text{ where } v_i \text{ is either } \varepsilon \text{ or a value.}$$

4.9.1 Structural congruence

SC-NIL	$[gs \,\vert\, P \parallel 0]$	$\equiv$	$[gs \,\vert\, P]$	for any $0 \in \mathbf{0_c}$
SC-COMM	$[gs \,\vert\, P \parallel Q]$	$\equiv$	$[gs \,\vert\, Q \parallel P]$	
SC-ASSOC	$[gs \,\vert\, (P \parallel Q) \parallel R]$	$\equiv$	$[gs \,\vert\, P \parallel (Q \parallel R)]$	

4.9.2 Nondeterminism and parallelism

NP-PROPAGPROB
$$\frac{[gs \,|\, P] \longrightarrow_p \boxplus_i\, p_i \bullet [gs_i \,|\, P_i]}{[gs \,|\, P \parallel Q] \longrightarrow_p \boxplus_i\, p_i \bullet [gs_i \,|\, P_i \parallel Q]}$$

NP-CONG
$$\frac{[gs \,|\, P] \longrightarrow_p \boxplus_i\, p_i \bullet [gs_i \,|\, P_i] \quad P \equiv P' \quad P_i \equiv P'_i \text{ for all } i}{[gs \,|\, P'] \longrightarrow_p \boxplus_i\, p_i \bullet [gs_i \,|\, P'_i]}$$

NP-PROBEVOL
$$\boxplus_i\, p_i \bullet [gs_i \,|\, P_i] \xrightarrow{p_i} [gs_i \,|\, P_i]$$

4.10 Evaluation

4.10.1 Basic rules

The first four rules define configuration change on a skip statement (rule OP-SKIP), constant (OP-CONST), variable (OP-VAR) and bracketed expression (OP-BRACKET). Next rule (OP-BLOCKHEAD) is used to evaluate sequence of statements from first to last. Last two rules (OP-SUBSTE and OP-SUBSTS) defines substitution of evaluated expressions.

OP-SKIP	$[gs \mid (lms, vp, \underline{;}\ ts)]$	$\longrightarrow_s$	$[gs \mid (lms, vp, ts)]$
OP-CONST	$[gs \mid (lms, vp, \underline{c}\ ts)]$	$\longrightarrow_v$	$[gs \mid (lms, vp, \underline{< \text{none}, c >}\ ts)]$ if c is a constant
OP-VAR	$[gs \mid (lms, vp, \underline{x}\ ts)]$	$\longrightarrow_v$	$[gs \mid (lms, vp, \underline{< ref, lms(ref) >}\ ts)]$ where $ref = varRef(x, vp)$
OP-BRACKET	$[gs \mid (lms, vp, \underline{(E)}\ ts)]$	$\longrightarrow_e$	$[gs \mid (lms, vp, \underline{E}\ ts)]$
OP-BLOCKHEAD	$[gs \mid (lms, vp, \underline{\widetilde{B}}\ ts)]$	$\longrightarrow_r$	$[gs \mid (lms, vp, \underline{head(\widetilde{B})}\ \underline{tail(\widetilde{B})}\ ts)]$
OP-SUBSTE	$[gs \mid (lms, vp, \mathbf{v}\ \underline{Ec}\ ts)]$	$\longrightarrow_e$	$[gs \mid (lms, vp, \underline{Ec[\mathbf{v}]}\ ts)]$
OP-SUBSTS	$[gs \mid (lms, vp, \mathbf{v}\ \underline{Sc}\ ts)]$	$\longrightarrow_e$	$[gs \mid (lms, vp, \underline{Sc[\mathbf{v}]}\ ts)]$

4.10.2 Promotable expressions

Promotable expressions are expressions that can be turned into statements by appending a semicolon. The expression is evaluated (rule OP-PROMOEXPR) but the resulting value is then forgotten (rule OP-PROMOFORGET).

$$\text{OP-PromoExpr} \quad [gs \,|\, (lms, vp, \underline{PE;}\ ts)] \ \longrightarrow_e\ [gs \,|\, (lms, vp, \underline{PE}\ \underline{\bullet;}\ ts)]$$

$$\text{OP-PromoForget} \quad [gs \,|\, (lms, vp, \underline{\mathbf{v};}\ ts)] \ \longrightarrow_s\ [gs \,|\, (lms, vp, ts)]$$

4.10.3 Allocation

Allocating a resource is performed by an evaluation of expression "**new** $\mathsf{T}()$" where T is a type of the resource, *ie.* a type of a channel or a quantum system. Type of quantum systems of dimension d are denoted by Q_d, *eg.* $\mathsf{qbit} = \mathsf{Q}_2$.

Allocation of a channel resource is handled by rule OP-AllocC, quantum resource allocation is handled by rule OP-AllocQ.

$$\begin{aligned}
\text{OP-AllocQ} \quad & [gs \,|\, (lms, vp, \underline{\mathbf{new}\ \mathsf{Q}_d()}\ ts)] \ \longrightarrow_v \\
& \quad [gs' \,|\, (lms', vp, \underline{<(Quantum, [l]), (GQuantum, [l])>}\ ts)] \\
& \text{where } \ l = |L| \\
& \qquad gs' = ((\rho \otimes (\tfrac{1}{d} I_d), L \cdot [d]), C) \\
& \qquad lms' = (lms_{Cl}, lms'_Q, lms_{Ch}, lms_{ChE}) \\
& \qquad lms'_Q = lms_Q[(Quantum, [l]) \mapsto (GQuantum, [l])] \\
& \text{given that } \ gs = ((\rho, L), C) \\
& \qquad lms = (lms_{Cl}, lms_Q, lms_{Ch}, lms_{ChE})
\end{aligned}$$

$$\begin{aligned}
\text{OP-AllocC} \quad & [gs \,|\, (lms, vp, \underline{\mathbf{new}\ \mathsf{channel}[\mathsf{T}]()}\ ts)] \ \longrightarrow_v \\
& \quad [gs' \,|\, (lms', vp, \underline{<(Channel, l), (GChannel, l)>}\ ts)] \\
& \text{where } \ l = |C| \\
& \qquad gs' = (Q, C \cdot [c_0 \models\!\!=\!\!\dashv c_1]) \\
& \qquad lms' = (lms_{Cl}, lms_Q, lms'_{Ch}, lms'_{ChE}) \\
& \qquad lms'_{Ch} = lms_{Ch}[(Channel, l) \mapsto (GChannel, l)] \\
& \qquad lms'_{ChE} = lms_{ChE}[(ChannelEnd_0, l) \mapsto (GChannel, l), \\
& \qquad\qquad (ChannelEnd_1, l) \mapsto (GChannel, l)] \\
& \text{given that } \ gs = (Q, C) \\
& \qquad lms = (lms_{Cl}, lms_Q, lms_{Ch}, lms_{ChE})
\end{aligned}$$

4.10.4 Variable declaration

Variable declaration is an addition of a variable to the innermost list of mappings of variable names to references. We consider any variable declaration of multiple variables of the same type: $\mathsf{T}\ a, b, c;$ to be an abbreviation of $\mathsf{T}\ a;\ \mathsf{T}\ b;\ \mathsf{T}\ c;$.

For declaration of a quantum compound system a construction q **aliasfor** $[q_0, \dots, q_n]$ is used where $q_0, \dots, q_n$ are names of quantum variables. Some of them can again be compound systems. To deal with this feature, all variables from $\{q_0, \dots, q_n\}$ that represent compound systems are expanded. This can be seen from the following example – we require the declarations on the left and right side to be equivalent:

$\mathsf{qbit}\ q_0, q_1, p;$	$\mathsf{qbit}\ q_0, q_1, p;$
q **aliasfor** $[q_0, q_1];$	q **aliasfor** $[q_0, q_1];$
r **aliasfor** $[p, q];$	r **aliasfor** $[p, q_0, q_1];$

OP-VARDECLMULTI $[gs \mid (lms, vp, \underline{T\ x, \widetilde{I};}\ ts)] \quad \longrightarrow_r$

$[gs \mid (lms, vp, \underline{T\ x;}\ \underline{T\ \widetilde{I};}\ ts)]$

OP-VARDECL $[gs \mid (lms, [L \circ_G [S_1 \circ_L S_2]], \underline{T\ x;}\ ts)] \quad \longrightarrow_s$

$[gs \mid (lms, [L \circ_G [S_1 \circ_L S_2']], ts)]$

where $S_2' = S_2[x \mapsto \textsf{none}]_{+,var}$

OP-VARDECLCHE $[gs \mid (lms, [L \circ_G [S_1 \circ_L S_2]], \underline{\textsf{channel}[T]\ c\ \textbf{withends}[c_0, c_1];}\ ts)] \quad \longrightarrow_s$

$[gs \mid (lms, [L \circ_G [S_1 \circ_L S_2']], ts)]$

where $S_2' = S_2[c \mapsto \textsf{none}]_{+,var}[c_0 \mapsto \textsf{none}]_{+,var}[c_1 \mapsto \textsf{none}]_{+,var}[c \mapsto (c_0, c_1)]_{+,ch}$

OP-VARDECLALF $[gs \mid (lms, [L \circ_G [S_1 \circ_L S_2]], \underline{q\ \textbf{aliasfor}\ [q_0, \ldots, q_n];}\ ts)] \quad \longrightarrow_s$

$[gs \mid (lms, [L \circ_G [S_1 \circ_L S_2']], ts)]$

where $S_2' = S_2[q \mapsto (Quantum, [l_0, \ldots, l_n])]_{+,var}[q \mapsto [q_0', \ldots, q_n']]_{+,qa}$

$$\text{given that} \quad l_i = \begin{cases} l & \text{if } varRef_L(q_i, [S_1 \circ_L S_2]) = (Quantum, l) \\ \perp & \text{otherwise} \end{cases}$$

$$q_i' = \begin{cases} p_0, \ldots, p_k & \text{if } aliasSubsyst_L(q_i, [S_1 \circ_L S_2']) = [p_0, \ldots, p_k] \\ q_i & \text{otherwise} \end{cases}$$

$varRef_L(q_i, [S_1 \circ_L S_2'])$ is defined for $0 \le i \le n$

4.10.5 Assignment

Assignment command $x = e$ has to be divided into two rules: one where expression e is evaluated (OP-ASSIGNEXPR) and the other where the result of evaluation of e is bound to variable x and possibly stored into memory (rules OP-ASSIGNNEWVALUE and OP-ASSIGNVALUE). The value is stored into memory if it was not there yet what is indicated by reference part of the internal value equal to none.

Assigning a quantum system to a variable can be complicated when the variable was declared using the **aliasfor** construct. For example, let ψ be a variable that represents a quantum system composed of systems ψ_A and ψ_B (it was declared as: ψ **aliasfor** $[\psi_A, \psi_B]$). Assigning a value to ψ must appropriately modify both ψ_A and ψ_B

and can be only performed if the assigned value represents a compound system made of two subsystems (rule OP-ASSIGNQAVALUE). Similarly, assigning a value to ψ_A must also modify ψ (rule OP-ASSIGNQVALUE).

OP-ASSIGNEXPR $\quad [gs \,|\, (lms, vp, \underline{x = e}\ ts)] \longrightarrow_e [gs \,|\, (lms, vp, \underline{e}\ \underline{x = \bullet}\ ts)]$

OP-ASSIGNNEWVALUE $\quad [gs \,|\, (lms, vp, \underline{x = \mathbf{v}}\ ts)] \longrightarrow_v [gs \,|\, (lms', vp', \underline{< lr', lv >}\ ts)]$

where $lr' = (Classical, nc)$

$lms'_{Cl} = lms_{Cl}[lr' \mapsto lv]_+$

$vp' = vp[x \mapsto lr']_{var}$

given that $\mathbf{v} =< lr, lv >$

nc is some natural number such that $lms_{Cl}((Classical, nc))$ is not defined

$lms = (lms_{Cl}, lms_Q, lms_{Ch}, lms_{ChE})$

$lms' = (lms'_{Cl}, lms_Q, lms_{Ch}, lms_{ChE})$

$lr = \text{none} \wedge lv \neq \perp$

OP-ASSIGNQVALUE $\quad [gs \,|\, (lms, vp, \underline{x = \mathbf{v}}\ ts)] \longrightarrow_v [gs \,|\, (lms', vp', \underline{\mathbf{v}}\ ts)]$

where $(lms'_Q, vp') = f_{assignQSystem}(lms_Q, vp, x, lr)$

given that $\mathbf{v} =< lr, lv >$

$lr = (Quantum, q)$ and $aliasSubsyst(x, vp)$ is not defined

$lms = (lms_{Cl}, lms_Q, lms_{Ch}, lms_{ChE})$

$lms' = (lms_{Cl}, lms'_Q, lms_{Ch}, lms_{ChE})$

OP-ASSIGNQAVALUE $\quad [gs \,|\, (lms, vp, \underline{x = \mathbf{v}}\ ts)] \longrightarrow_v [gs \,|\, (lms', vp', \underline{\mathbf{v}}\ ts)]$

where $(lms'_Q, vp') = f_{assignQAlias}(lms_Q, vp, x, lr)$

given that $\mathbf{v} =< lr, lv >$

$lr = (Quantum, q)$ and $aliasSubsyst(x, vp)$ is defined

$lms = (lms_{Cl}, lms_Q, lms_{Ch}, lms_{ChE})$

$lms' = (lms_{Cl}, lms'_Q, lms_{Ch}, lms_{ChE})$

$$\text{OP-AssignValue} \quad [gs \,|\, (lms, vp, \underline{x = \mathbf{v}}\ ts)] \quad \longrightarrow_v \quad [gs \,|\, (lms, vp', \underline{\mathbf{v}}\ ts)]$$

$$\text{where } vp' = \begin{cases} vp[x \mapsto lr]_{var}[x_0 \mapsto lr]_{var}[x_1 \mapsto lr]_{var} & \text{if } lr = (Channel, x) \\ & \quad \text{and } chanEnds(x, vp) = (x_0, x_1) \\ vp[x \mapsto lr]_{var} & \text{otherwise} \end{cases}$$

$$\text{given that} \quad \mathbf{v} =< lr, lv >$$
$$(lr = \text{none} \wedge lv = \bot) \vee (lr \neq (Quantum, q))$$

4.10.6 Block

Block command is used to limit scope of variables and to execute multiple statements:

$$\text{OP-Block} \qquad [gs \,|\, (lms, [vp_G \circ_G vp], \underline{\{\bar{B}\}}\ ts)] \quad \longrightarrow_s$$
$$[gs|(lms, [vp_G \circ_G [vp \circ_L \square]], \underline{\bar{B}}\ \underline{\circ_L}\ ts)]$$

$$\text{OP-BlockEnd} \quad [gs \,|\, (lms, [vp_G \circ_G [vp \circ_L vp_L]], \underline{\circ_L}\ ts)] \quad \longrightarrow_s$$
$$[gs|(lms, [vp_G \circ_G vp], ts)]$$

4.10.7 Conditional statement – if

Conditional expression **if** (E) S_1 **else** S_2 has to be split into three rules: one where the condition is evaluated (OP-IfExpr) and rules for reduction when the condition evaluates to **true** (OP-IfTrue) and **false** (OP-IfFalse).

$$\text{OP-IFEXPR}\quad [gs \,|\, (lms, vp, \underline{\textbf{if}\ (E)\ S_1\ \textbf{else}\ S_2}\ ts)] \longrightarrow_e (lms, vp, \underline{E}\ \underline{\textbf{if}\ (\bullet)\ S_1\ \textbf{else}\ S_2}\ ts)]$$

$$\text{OP-IFTRUE}\quad [gs \,|\, (lms, vp, \underline{\textbf{if}\ (\mathbf{v})\ S_1\ \textbf{else}\ S_2}\ ts)] \longrightarrow_s [gs \,|\, (lms, vp, \underline{S_1}\ ts)] \quad \text{if } \mathrm{isTrue}(\mathbf{v})$$

$$\text{OP-IFFALSE}\quad [gs \,|\, (lms, vp, \underline{\textbf{if}\ (\mathbf{v})\ S_1\ \textbf{else}\ S_2}\ ts)] \longrightarrow_s [gs \,|\, (lms, vp, \underline{S_2}\ ts)] \quad \text{if } \mathrm{isFalse}(\mathbf{v})$$

4.10.8 Conditional cycle – while

While is syntactically converted to a corresponding **if** statement.

$$\text{OP-WHILE}\quad [gs \,|\, (lms, vp, \underline{\textbf{while}\ (E)\ S}\ ts)] \longrightarrow_r [gs \,|\, (lms, vp, \underline{\textbf{if}\ (E)\ \{S\ \textbf{while}\ (E)\ S\}\ \textbf{else}\ ;}\ ts)]$$

4.10.9 Method call

Call of a method m whose parameters are expressions is rewritten to a call of method m whose parameters are values. Parameters passed to the method are evaluated in the original variable context vp (rule OP-METHODCALLEXPR).

The call of the method m with value parameters is evaluated in two different ways depending on whether m represents a classical method or a quantum operator.

In the case when m represents a classical method, the call of a method m is rewritten to the unwound body of method m (rule OP-DOMETHODCALLCL) translated to the internal syntax by function $\mathrm{methodBody}$.

If m represents a quantum operator $\mathcal{E}_m$, the operator $\mathcal{E}_m$ is applied to a quantum subsystem specified by the parameters $\mathbf{v}_1 =< r_1, v_1 >, \ldots, @\mathbf{v}_n =< r_n, v_n >$. Values $v_1, \ldots, v_n$ are either global references to quantum storage $(\mathrm{GQuantum}, l_{r_1})$, $\ldots, (\mathrm{GQuantum}, l_{r_n})$ or $\perp$. In the case when $\perp$ is referred, a configuration becomes stuck (a run-time error occurs in a real implementation).

The condition that all manipulated quantum system are physically different can be reformulated as: all the indices in lists $l_{r_1}, \dots, l_{r_n}$ are mutually different, *ie.* $\mathrm{set}_{[\circlearrowright]}(l_{r_j}) \cap \mathrm{set}_{[\circlearrowright]}(l_{r_k}) = \emptyset$ and $|l_{r_j}|_{[\circlearrowright]} = |\mathrm{set}_{[\circlearrowright]}(l_{r_j})|$ for all $1 \le j, k \le n,\ j \ne k$.

The list qsi of indices of quantum systems to be measured is given by a concatenation of individual linearized lists: $qsi = l_{r_1} \cdot \ldots \cdot l_{r_n}$, which determines quantum system q_{qsi}. Dimension $d_{q_{qsi}}$ of the quantum system q_l is calculated from the global part $((\rho, L), C)$ of the configuration as

$$d_{q_{qsi}} = \sum_{i=1}^{|qsi|} L_{qsi_i}.$$

We denote d the order of matrix ρ and $\bar{d}$ the dimension of untouched part of the system, $\bar{d} = d/d_{q_{qsi}}$ (rule OP-DoMethodCallQ).

OP-MethodCallExpr $\quad [gs \,|\, (lms, vp, \underline{m(\widetilde{\mathbf{v}}, E, \widetilde{E})}\ ts)) \longrightarrow_e$

$\qquad [gs \,|\, (lms, vp, \underline{E}\ \underline{m(\widetilde{\mathbf{v}}, \bullet, \widetilde{E})}\ ts)]$

OP-DoMethodCallCl $\quad [gs \,|\, (lms, vp, \underline{m(\mathbf{v}_1, \dots, \mathbf{v}_n)}\ ts)) \longrightarrow_e$

$\qquad [gs \,|\, (lms, [vp \circ_G [\Box \circ_L vp'_M]], methodBody(m) \underline{\circ_M}\ ts)]$

where $vp'_M = ([a_1 \mapsto r_1, \dots, a_n \mapsto r_n], [], [])$

given that $\mathbf{v}_1 =< r_1, v_1 >, \dots, \mathbf{v}_n =< r_n, v_n >$

m represents a classical method and

$T\ m(T_1\ a_1, \dots T_n\ a_n)$ is a header of method m

$$\begin{aligned}
&\text{OP-D}\textsc{o}\text{M}\textsc{ethod}\text{C}\textsc{all}\text{Q} \quad [gs \,|\, (lms, vp, \underline{m(\mathbf{v}_1, \ldots, \mathbf{v}_n)}\ ts)) \quad \longrightarrow_v \\
&\qquad\qquad\qquad\qquad\qquad\qquad [gs' \,|\, (lms, vp, \underline{< \mathsf{none}, \perp >}\ ts)] \\
&\text{where} \quad gs' = ((\rho', L), C) \\
&\qquad\quad \rho' = \Pi^T(\mathcal{E}_m \otimes I_{\bar{d}}(\Pi\rho\Pi^T))\Pi \\
&\text{given that } \mathbf{v}_1 =< r_1, v_1 >, \ldots, \mathbf{v}_n =< r_n, v_n > \\
&\qquad\qquad gs = ((\rho, L), C) \\
&\qquad\qquad v_1 = (GQuantum, l_{r_1}), \ldots, v_n = (GQuantum, l_{r_n}) \\
&\qquad\qquad \mathrm{set}_{[\circlearrowleft]}(l_{r_j}) \cap \mathrm{set}_{[\circlearrowleft]}(l_{r_k}) = \emptyset \text{ for all } 1 \le j, k \le n,\ j \ne k \\
&\qquad\qquad |l_{r_j}|_{[\circlearrowleft]} = |\,\mathrm{set}_{[\circlearrowleft]}(l_{r_j})| \text{ for all } 1 \le j \le n \\
&\qquad\qquad \Pi \text{ is a permutation matrix which places affected quantum} \\
&\qquad\qquad\qquad \text{systems to the head of } \rho \text{ in the order given by } \mathbf{v}_1, \ldots, \mathbf{v}_n \\
&\qquad\qquad m \text{ represents a quantum operator } \mathcal{E}_m
\end{aligned}$$

4.10.10 Returning from a method

When a method is ended, the control is passed back to its caller. The place where the called method was invoked by the caller is marked by the $\circ_M$ symbol. If a method returns no value, it can either end without **return** statement just by evaluating the last statement in the method (handled by OP-RETURNVOIDIMPL) or by explicit **return** statement. In that case the **return** statement pops everything from the term stack until it finds the symbol $\circ_M$ (OP-RETURNVOIDIMPL). When the method returns a value, the return value is evaluated first (OP-RETURNEXPR) and the return value is then left on top of the stack (OP-RETURNVALUE).

OP-RETURNVOID
$$[gs \mid (lms, [vp \circ_G vp_M], \underline{\textbf{return};}\ ts_M \underline{\circ_M}\ ts)] \longrightarrow_v$$
$$[gs \mid (lms, vp, \underline{< \mathsf{none}, \bot >}\ ts)]$$

OP-RETURNVOIDIMPL
$$[gs \mid (lms, [vp \circ_G vp_M], \underline{\circ_M}\ ts)] \longrightarrow_v$$
$$[gs \mid (lms, vp, \underline{< \mathsf{none}, \bot >}\ ts)]$$

OP-RETURNEXPR
$$[gs \mid (lms, [vp \circ_G vp_M], \underline{\textbf{return}\ E;}\ ts_M \underline{\circ_M}\ ts)] \longrightarrow_e$$
$$[gs \mid (lms, vp, \underline{E}\ \underline{\textbf{return}\ \bullet;}\ ts_M \underline{\circ_M}\ ts)$$

OP-RETURNVALUE
$$[gs \mid (lms, [vp \circ_G vp_M], \underline{\textbf{return}\ \mathbf{v};}\ ts_M \underline{\circ_M}\ ts)] \longrightarrow_v$$
$$[gs \mid (lms, vp, \mathbf{v}\ ts)]$$

4.10.11 Forking

Forking is similar to method call, therefore the rule OP-FORKEXPR is similar to the rule OP-METHODCALLEXPR. In the rule OP-DOFORK, a new process is started, values passed as parameters to the forked method are copied to the new process memory and non-duplicable values passed as parameters to the forked method are unmapped from the parent process memory.

OP-FORKEXPR
$$[gs \mid (lms, vp, \underline{\textbf{fork}\ m(\widetilde{\mathbf{v}}, E, \widetilde{E});}\ ts)] \longrightarrow_e$$
$$[gs \mid (lms, vp, \underline{E}\ \underline{\textbf{fork}\ m(\widetilde{\mathbf{v}}, \bullet, \widetilde{E});}\ ts)]$$

OP-DOFORK
$$[gs \mid (lms, vp, \underline{\textbf{fork}\ m(\mathbf{v}_1, \ldots, \mathbf{v}_n);}\ ts)] \longrightarrow_p$$
$$[gs \mid (lms'_1, vp, ts) \parallel (lms'_2, [\blacksquare \circ_G [\square \circ_L vp'_M]], methodBody(m) \underline{\circ_M})]$$

where $vp'_M = ([a_1 \mapsto v_1, \ldots, a_n \mapsto v_n], [], [])$

$lms'_1 = unmap_{nd}(\{r_1, \ldots, r_n\}, lms_1)$

$lms'_2 = lms_1$ restricted to domain $\{r_1, \ldots, r_n\}$

given that $\mathbf{v}_1 =< r_1, v_1 >, \ldots, \mathbf{v}_n =< r_n, v_n >$

$\mathsf{T}\ m(\mathsf{T}_1\ a_1, \ldots \mathsf{T}_n\ a_n)$ is a header of method m

4.10.12 Measurement

Measurement is performed when **measure**$(b, e_1, \ldots, e_n)$ primitive method is invoked. Its first argument b determines measurement basis, the other arguments determine quantum systems that are to be simultaneously measured. Arguments $e_1, \ldots, e_n$ evaluate to internal values $\mathbf{v}_1 =< r_1, v_1 >, \ldots, \mathbf{v}_n =< r_n, v_n >$. Values $v_1, \ldots, v_n$ are either global references to quantum storage $(GQuantum, l_{r_1}), \ldots, (GQuantum, l_{r_n})$ or $\perp$. In the case when $\perp$ is referred, a run-time error must occur, therefore we do not provide any rule for this case.

The condition that all the measured system are physically different can be reformulated as: all the indices in lists $l_{r_1}, \ldots, l_{r_n}$ are mutually different, *ie.* $\mathrm{set}_{[\circlearrowright]}(l_{r_j}) \cap \mathrm{set}_{[\circlearrowright]}(l_{r_k}) = \emptyset$ and $|l_{r_j}|_{[\circlearrowright]} = |\,\mathrm{set}_{[\circlearrowright]}(l_{r_j})|$ for all $1 \leq j, k \leq n,\ j \neq k$.

The list qsi of indices of quantum systems to be measured is given by a concatenation of individual linearized lists: $qsi = l_{r_1} \cdot \ldots \cdot l_{r_n}$, which determines quantum system q_{qsi}. Dimension $d_{q_{qsi}}$ of the quantum system q_l is calculated from the global part $((\rho, L), C)$ of the configuration as

$$d_{q_{qsi}} = \sum_{i=1}^{|qsi|} L_{qsi_i}.$$

We denote d the order of matrix ρ and $\bar{d}$ the dimension of unmeasured part of the system, $\bar{d} = d/d_{q_{qsi}}$. Now we can formulate rules for measurement:

$$\text{OP-MEASUREEXPR} \qquad [gs \,|\, (lms, vp, \underline{\mathbf{measure}(\widetilde{\mathbf{v}}, E, \widetilde{E})}\ ts] \ \longrightarrow_e$$
$$[gs \,|\, (lms, vp, \underline{E}\ \underline{\mathbf{measure}(\widetilde{\mathbf{v}}, \bullet, \widetilde{E})}\ ts)]$$

$$\text{OP-DOMEASURE} \qquad [gs \,|\, (lms, vp, \underline{\mathbf{measure}(\mathbf{v}_b, \mathbf{v}_1, \ldots, \mathbf{v}_n)}\ ts)] \ \longrightarrow_v$$
$$\boxplus_i\, p_i \bullet [gs_i \,|\, (lms, vp, \underline{< \mathsf{none}, \lambda_i >}\ ts)]$$

where $gs_i = ((\rho_i, L), C)$

$$p_i = \mathrm{Tr}\,[\Pi^T(P_i \otimes I_{\bar{d}})\Pi\rho\Pi^T(P_i \otimes I_{\bar{d}})^\dagger\Pi]$$
$$\rho_i = \frac{\Pi^T(P_i \otimes I_{\bar{d}})\Pi\rho\Pi^T(P_i \otimes I_{\bar{d}})^\dagger\Pi}{p_i}$$

given that $\mathbf{v}_1 =< r_1, v_1 >, \ldots, \mathbf{v}_n =< r_n, v_n >$

$v_1 = (GQuantum, l_{r_1}), \ldots, v_n = (GQuantum, l_{r_n})$

$gs = ((\rho, L), C)$

$\mathrm{set}_{[\circlearrowright]}(l_{r_j}) \cap \mathrm{set}_{[\circlearrowright]}(l_{r_k}) = \emptyset$ for all $1 \leq j, k \leq n,\ j \neq k$

$|l_{r_j}|_{[\circlearrowright]} = |\mathrm{set}_{[\circlearrowright]}(l_{r_j})|$ for all $1 \leq j \leq n$

$\sum_i \lambda_i P_i$ is a spectral decomposition of a measurement in the basis given by $\mathbf{v}_b$

Π is a permutation matrix which places measured quantum systems to the head of ρ in the order given by $\mathbf{v}_1, \ldots, \mathbf{v}_n$

4.10.13 Communication

Communication is performed when there is one process sending a value over a channel end and another process waiting to receive a value over a channel end provided that both channel ends belong to the same channel. This condition is equivalent to the condition that both channel ends refer to the same channel. First three rules (OP-SENDEXPR1, OP-SENDEXPR2 and OP-RECVEXPR) are to evaluate statement arguments and the rule OP-SENDRECV performs the communication.

Unique ownership of resources (both quantum and channel) is ensured by unmapping them from the local memory of the sender process using the function $unmap_{nd}$.

$$\text{OP-S}\textsc{end}\text{E}\textsc{xpr}1 \quad [gs \mid (lms, vp, \underline{\mathbf{send}(E_1, E_2);}\ ts)] \longrightarrow_e [gs \mid (lms, vp, \underline{E_1}\ \underline{\mathbf{send}(\bullet, E_2);}\ ts]$$

$$\text{OP-S}\textsc{end}\text{E}\textsc{xpr}2 \quad [gs \mid (lms, vp, \underline{\mathbf{send}(\mathbf{v}_c, E);}\ ts)] \longrightarrow_e [gs \mid (lms, vp, \underline{E}\ \underline{\mathbf{send}(\mathbf{v}_c, \bullet);}\ ts]$$

$$\text{OP-R}\textsc{ecv}\text{E}\textsc{xpr} \quad [gs \mid (lms, vp, \underline{\mathbf{recv}(E)}\ ts)] \longrightarrow_e [gs \mid (lms, vp, \underline{E}\ \underline{\mathbf{recv}(\bullet)}\ ts]$$

$$\begin{aligned}
\text{OP-S}\textsc{end}\text{R}\textsc{ecv} \quad & [gs \mid (lms_1, vp_1, \underline{\mathbf{send}(\mathbf{v}_{c_1}, \mathbf{v}_e);}\ ts_1) \parallel (lms_2, vp_2, \underline{\mathbf{recv}(\mathbf{v}_{c_2})}\ ts_2)] \longrightarrow_p \\
& \qquad [gs \mid (lms'_1, vp_1, ts_1) \parallel (lms'_2, vp_2, \underline{< lr'_2, lv'_2 >}\ ts_2)] \\
\text{where } & lms'_1 = unmap_{nd}(sentRef, lms_1) \\
& lr'_2 = \begin{cases} lr_1 & \text{if } sentRef \in Ref_{nd} \\ \text{none} & \text{otherwise} \end{cases} \\
& lv'_2 = sentVal \\
& lms'_2 = \begin{cases} lms_2[sentRef \mapsto sentVal] & \text{if } sentRef \in Ref_{nd} \\ lms_2 & \text{otherwise} \end{cases} \\
\text{given that } & \mathbf{v}_e =< sentRef, sentVal > \\
& \mathbf{v}_{c_1} =< c_1Ref, c_1Val > \\
& \mathbf{v}_{c_2} =< c_2Ref, c_2Val > \\
& c_1Ref \neq \text{none and } c_2Ref \neq \text{none} \\
& c_1Val = c_2Val \quad \text{(both ends refer to the same channel)}
\end{aligned}$$

5 Conclusion and future work

We have described internal syntax and operational semantics of LanQ programming language. This language can be used for implementation of both quantum algorithms and quantum protocols. An example of a program run can be found in Appendix A.

After proving series of standard lemmas in style of Wright and Felleisen [WF94], it will be possible to use the language to prove correctness of quantum protocols and algorithms. Proving these lemmas is planned for the nearest future.

By-reference usage of variables causes sometimes unwanted behaviour: now it is possible to declare a program where a process sends a qubit and then it attempts to measure it. This is impossible as the qubit is then not owned by the process. Currently, this leads to a stuck configuration. In future, this situation will be handled by runtime errors what will also help in debugging programs and protocols written using LanQ.

The simulator of LanQ is also being developed and will be publicly available.

Acknowledgements

I would like to express my thanks to Simon Gay, Philippe Jorrand, Rajagopal Nagarajan and Nick Papanikolaou for their invaluable comments on LanQ and related discussions. I wish also thank my supervisor, prof. Jozef Gruska, for his guidance.

References

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

[BB84] C.H. Bennett and G. Brassard. Quantum Cryptography: Public Key Distribution and Coin Tossing. In *Proceedings of IEEE International Conference on Computers Systems and Signal Processing*, pages 175–179, India, December 1984. Bangalore.

[BBC$^+$93] 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. *Physical Review Letters*, (70):1895–1899, 1993.

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

[Ben92] C. H. Bennett. Quantum cryptography using any two nonorthogonal states. *Phys. Rev. Lett.*, 68(21):3121–3124, 1992.

[Eke91] Artur K. Ekert. Quantum cryptography based on bell's theorem. *Phys. Rev. Lett.*, 67(6):661–663, 1991.

[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.

[GN06] Simon J. Gay and Rajagopal Nagarajan. Types and typechecking for Communicating Quantum Processes. *Mathematical. Structures in Comp. Sci.*, 16:375–406, 2006.

[JL04] Philippe Jorrand and Marie Lalire. Toward a quantum process algebra. In *CF '04: Proceedings of the 1st conference on Computing frontiers*, pages 111–119, New York, NY, USA, 2004. ACM Press.

[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.

[Lal06] Marie Lalire. *Développement d'une notation alorithmique pour le calcul quantique*. PhD thesis, 2006.

[LGP06] M. Lampis, K. G. Ginis, and N. S. Papaspyrou. Quantum data and control made easier. In Peter Selinger, editor, *Preliminary Proceedings of the 4th International Workshop on Quantum Programming Languages*, pages 73–86, 2006. The final version will be published in Electronic Notes in Theoretical Computer Science.

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

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

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

[Sho94] P. W. Shor. Algorithms for quantum computation: Discrete logarithms and factoring. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*. IEEE Computer Society Press, 1994.

[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.

[vT03] André van Tonder. A Lambda Calculus for Quantum Computation. *quant-ph/0307150*, 2003.

[WF94] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

[WZ82] W.K. Wootters and W.H. Zurek. A single quantum cannot be cloned. *Nature*, 299:802–803, 1982.

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

A Program execution example

The probabilistic nature of measurement of quantum particles allows us to create generator of truly random numbers: Let us have a quantum particle in the state $|\psi\rangle = \frac{1}{2}(|0\rangle + |1\rangle)$. Now we apply a measurement of this particle in the basis $\{|0\rangle, |1\rangle\}$ (so called *standard basis*). The result of the measurement is 0 or 1 with equal probability.

The random number generator can be implemented as shown in the Figure 4.

```
int main() {
        qbit q;
        q = new qbit();
        return measure (StdBasis, q);
}
```

Figure 4: Program example: Random number generator

Before the execution, we must specify the $\mathtt{methodBody}$ function. We have a program containing only a method $\mathtt{main}$, hence the domain of $\mathtt{methodBody}$ function is $\{\mathtt{main}\}$. $\mathtt{methodBody}$ is then specified as:

$\mathtt{methodBody}(\mathtt{main}) =$ { qbit q; q = **new** qbit(); **return measure** ($\mathtt{StdBasis}$, q); }

The execution of the program is shown in Figure 5.

B Concrete syntax

Concrete syntax of LanQ language is shown in the Figure 6. The reserved words of the language are written in **bold** and the identifier names are in *italic*. Grammar is given in nondeterministic extended Backus-Naur form (EBNF). The root of grammar is the nonterminal program.

$$\textbf{start} = [(((\mathbf{1}), ▯), ▯) \mid ((▯, ▯, ▯, ▯), ■, \underline{\mathit{main}()})]$$

$$\downarrow_e \text{OP-DOMETHODCALLCL}$$

$$[(((1), ▯), ▯) \mid ((▯, ▯, ▯, ▯), [■ \circ_G [□ \circ_L □]], \underline{\{\ \mathsf{qbit}\ q;\ q = \textbf{new}\ \mathsf{qbit}();\ \textbf{return measure}\ (\mathtt{StdBasis}, q);\ \}}\ \underline{\circ_M})]$$

$$\downarrow_s \text{OP-BLOCK}$$

$$[(((1), ▯), ▯) \mid ((▯, ▯, ▯, ▯), [■ \circ_G [[□ \circ_L □] \circ_L □]],\ \underline{\mathsf{qbit}\ q;\ q = \textbf{new}\ \mathsf{qbit}();\ \textbf{return measure}\ (\mathtt{StdBasis}, q);}\ \underline{\circ_L}\ \underline{\circ_M})]$$

$$\downarrow_r \text{OP-BLOCKHEAD}$$

$$[(((1), ▯), ▯) \mid ((▯, ▯, ▯, ▯), [■ \circ_G [[□ \circ_L □] \circ_L □]],\ \underline{\mathsf{qbit}\ q;}\ \underline{q = \textbf{new}\ \mathsf{qbit}();\ \textbf{return measure}\ (\mathtt{StdBasis}, q);}\ \underline{\circ_L}\ \underline{\circ_M})]$$

$$\downarrow_s \text{OP-VARDECL}$$

$$[(((1), ▯), ▯) \mid ((▯, ▯, ▯, ▯), [■ \circ_G [[□ \circ_L □] \circ_L [q \mapsto \mathsf{none}]]], \underline{q = \textbf{new}\ \mathsf{qbit}();\ \textbf{return measure}\ (\mathtt{StdBasis}, q);}\ \underline{\circ_L}\ \underline{\circ_M})]$$

$$\downarrow_r \text{OP-BLOCKHEAD}$$

$$[(((1), ▯), ▯) \mid ((▯, ▯, ▯, ▯), [■ \circ_G [[□ \circ_L □] \circ_L [q \mapsto \mathsf{none}]]], \underline{q = \textbf{new}\ \mathsf{qbit}();}\ \underline{\textbf{return measure}\ (\mathtt{StdBasis}, q);}\ \underline{\circ_L}\ \underline{\circ_M})]$$

$$\downarrow_e \text{OP-PROMOEXPR}$$

$$[(((1), ▯), ▯) \mid ((▯, ▯, ▯, ▯), [■ \circ_G [[□ \circ_L □] \circ_L [q \mapsto \mathsf{none}]]], \underline{q = \textbf{new}\ \mathsf{qbit}()}\ \underline{\bullet\ ;}\ \underline{\textbf{return measure}\ (\mathtt{StdBasis}, q);}\ \underline{\circ_L}\ \underline{\circ_M})]$$

$$\downarrow_e \text{OP-ASSIGNEXPR}$$

Figure 5: Program example: Random number generator execution (to be continued)

$$\downarrow_e \text{OP-A}\textsc{ssign}\text{E}\textsc{xpr}$$

$$[(((1),[]),[]) \mid (([],[],[],[]),[\blacksquare \circ_G [[\square \circ_L \square] \circ_L [q \mapsto \mathsf{none}]]], \underline{\textbf{new}\ \mathsf{qbit}()}\ \underline{q = \bullet}\ \underline{\bullet\ ;}\ \underline{\textbf{return measure}\ (\mathit{StdBasis}, q);}\ \underline{\circ_L}\ \underline{\circ_M})]$$

$$\downarrow_v \text{OP-A}\textsc{lloc}\text{Q}$$

$$[((\tfrac{1}{2}\begin{pmatrix}1 & 0\\ 0 & 1\end{pmatrix},[2]),[]) \mid (([],[(\mathit{Quantum},[1]) \mapsto (\mathit{GQuantum},[1])],[],[]),[\blacksquare \circ_G [[\square \circ_L \square] \circ_L [q \mapsto \mathsf{none}]]],$$
$$\underline{<(\mathit{Quantum},[1]),(\mathit{GQuantum},[1])>}\ \underline{q = \bullet}\ \underline{\bullet\ ;}\ \underline{\textbf{return measure}\ (\mathit{StdBasis}, q);}\ \underline{\circ_L}\ \underline{\circ_M})]$$

$$\downarrow_e \text{OP-S}\textsc{ubst}\text{E}$$

$$[((\tfrac{1}{2}\begin{pmatrix}1 & 0\\ 0 & 1\end{pmatrix},[2]),[]) \mid (([],[(\mathit{Quantum},[1]) \mapsto (\mathit{GQuantum},[1])],[],[]),[\blacksquare \circ_G [[\square \circ_L \square] \circ_L [q \mapsto \mathsf{none}]]],$$
$$\underline{q =<(\mathit{Quantum},[1]),(\mathit{GQuantum},[1])>}\ \underline{\bullet\ ;}\ \underline{\textbf{return measure}\ (\mathit{StdBasis}, q);}\ \underline{\circ_L}\ \underline{\circ_M})]$$

$$\downarrow_v \text{OP-A}\textsc{ssign}\text{QV}\textsc{alue}$$

$$[((\tfrac{1}{2}\begin{pmatrix}1 & 0\\ 0 & 1\end{pmatrix},[2]),[]) \mid (([],[(\mathit{Quantum},[1]) \mapsto (\mathit{GQuantum},[1])],[],[]),[\blacksquare \circ_G [[\square \circ_L \square] \circ_L [q \mapsto (\mathit{Quantum},[1])]]],$$
$$\underline{<(\mathit{Quantum},[1]),(\mathit{GQuantum},[1])>}\ \underline{\bullet\ ;}\ \underline{\textbf{return measure}\ (\mathit{StdBasis}, q);}\ \underline{\circ_L}\ \underline{\circ_M})]$$

$$\downarrow_e \text{OP-S}\textsc{ubst}\text{S}$$

$$[((\tfrac{1}{2}\begin{pmatrix}1 & 0\\ 0 & 1\end{pmatrix},[2]),[]) \mid (([],[(\mathit{Quantum},[1]) \mapsto (\mathit{GQuantum},[1])],[],[]),[\blacksquare \circ_G [[\square \circ_L \square] \circ_L [q \mapsto (\mathit{Quantum},[1])]]],$$
$$\underline{<(\mathit{Quantum},[1]),(\mathit{GQuantum},[1])>\ ;}\ \underline{\textbf{return measure}\ (\mathit{StdBasis}, q);}\ \underline{\circ_L}\ \underline{\circ_M})]$$

$$\downarrow_s \text{OP-P}\textsc{romo}\text{F}\textsc{orget}$$

Figure 5 (continued): Program example: Random number generator execution (to be continued)

$$\downarrow_s \text{OP-}\textsc{PromoForget}$$

$$[((\tfrac{1}{2}\begin{pmatrix}1 & 0\\ 0 & 1\end{pmatrix}, [2]), []) \mid (([], [(\mathit{Quantum}, [1]) \mapsto (\mathit{GQuantum}, [1])], [], []), [\blacksquare \circ_G [[\square \circ_L \square] \circ_L [q \mapsto (\mathit{Quantum}, [1])]]],$$
$$\underline{\mathbf{return}\ \mathbf{measure}\ (\mathit{StdBasis}, q);}\ \underline{\circ_L}\ \underline{\circ_M})]$$

$$\downarrow_e \text{OP-}\textsc{ReturnExpr}$$

$$[((\tfrac{1}{2}\begin{pmatrix}1 & 0\\ 0 & 1\end{pmatrix}, [2]), []) \mid (([], [(\mathit{Quantum}, [1]) \mapsto (\mathit{GQuantum}, [1])], [], []), [\blacksquare \circ_G [[\square \circ_L \square] \circ_L [q \mapsto (\mathit{Quantum}, [1])]]],$$
$$\underline{\mathbf{measure}(\mathit{StdBasis}, q)}\ \underline{\mathbf{return}\ \bullet;}\ \underline{\circ_L}\ \underline{\circ_M})]$$

$$\downarrow_e \text{OP-}\textsc{MeasureExpr}$$

$$[((\tfrac{1}{2}\begin{pmatrix}1 & 0\\ 0 & 1\end{pmatrix}, [2]), []) \mid (([], [(\mathit{Quantum}, [1]) \mapsto (\mathit{GQuantum}, [1])], [], []), [\blacksquare \circ_G [[\square \circ_L \square] \circ_L [q \mapsto (\mathit{Quantum}, [1])]]],$$
$$\underline{\mathit{StdBasis}}\ \underline{\mathbf{measure}(\bullet, q)}\ \underline{\mathbf{return}\ \bullet;}\ \underline{\circ_L}\ \underline{\circ_M})]$$

$$\downarrow_v \text{OP-}\textsc{Const}$$

$$[((\tfrac{1}{2}\begin{pmatrix}1 & 0\\ 0 & 1\end{pmatrix}, [2]), []) \mid (([], [(\mathit{Quantum}, [1]) \mapsto (\mathit{GQuantum}, [1])], [], []), [\blacksquare \circ_G [[\square \circ_L \square] \circ_L [q \mapsto (\mathit{Quantum}, [1])]]],$$
$$\underline{< \mathsf{none}, \mathit{StdBasis} >}\ \underline{\mathbf{measure}(\bullet, q)}\ \underline{\mathbf{return}\ \bullet;}\ \underline{\circ_L}\ \underline{\circ_M})]$$

$$\downarrow_e \text{OP-}\textsc{Subst}\text{E}$$

$$[((\tfrac{1}{2}\begin{pmatrix}1 & 0\\ 0 & 1\end{pmatrix}, [2]), []) \mid (([], [(\mathit{Quantum}, [1]) \mapsto (\mathit{GQuantum}, [1])], [], []), [\blacksquare \circ_G [[\square \circ_L \square] \circ_L [q \mapsto (\mathit{Quantum}, [1])]]],$$
$$\underline{\mathbf{measure}(< \mathsf{none}, \mathit{StdBasis} >, q)}\ \underline{\mathbf{return}\ \bullet;}\ \underline{\circ_L}\ \underline{\circ_M})]$$

$$\downarrow_e \text{OP-}\textsc{MeasureExpr}$$

Figure 5 (continued): Program example: Random number generator execution (to be continued)

$$\downarrow_e \text{OP-M{\scriptsize EASURE}E{\scriptsize XPR}}$$

$$[((\tfrac{1}{2}\begin{pmatrix}1 & 0\\ 0 & 1\end{pmatrix}, [2]), []) \mid (([], [(\mathit{Quantum}, [1]) \mapsto (\mathit{GQuantum}, [1])], [], []), [\blacksquare \circ_{\mathrm{G}} [[\square \circ_{\mathrm{L}} \square] \circ_{\mathrm{L}} [\mathsf{q} \mapsto (\mathit{Quantum}, [1])]]],$$
$$\underline{\mathsf{q}}\ \underline{\textbf{measure}(< \mathsf{none}, \mathit{StdBasis} >, \bullet)}\ \underline{\textbf{return}\ \bullet;}\ \underline{\circ_{\mathrm{L}}}\ \underline{\circ_{\mathrm{M}}})]$$

$$\downarrow_v \text{OP-V{\scriptsize AR}}$$

$$[((\tfrac{1}{2}\begin{pmatrix}1 & 0\\ 0 & 1\end{pmatrix}, [2]), []) \mid (([], [(\mathit{Quantum}, [1]) \mapsto (\mathit{GQuantum}, [1])], [], []), [\blacksquare \circ_{\mathrm{G}} [[\square \circ_{\mathrm{L}} \square] \circ_{\mathrm{L}} [\mathsf{q} \mapsto (\mathit{Quantum}, [1])]]],$$
$$\underline{< (\mathit{Quantum}, [1]), (\mathit{GQuantum}, [1]) >}\ \underline{\textbf{measure}(< \mathsf{none}, \mathit{StdBasis} >, \bullet)}\ \underline{\textbf{return}\ \bullet;}\ \underline{\circ_{\mathrm{L}}}\ \underline{\circ_{\mathrm{M}}})]$$

$$\downarrow_e \text{OP-S{\scriptsize UBST}E}$$

$$[((\tfrac{1}{2}\begin{pmatrix}1 & 0\\ 0 & 1\end{pmatrix}, [2]), []) \mid (([], [(\mathit{Quantum}, [1]) \mapsto (\mathit{GQuantum}, [1])], [], []), [\blacksquare \circ_{\mathrm{G}} [[\square \circ_{\mathrm{L}} \square] \circ_{\mathrm{L}} [\mathsf{q} \mapsto (\mathit{Quantum}, [1])]]],$$
$$\underline{\textbf{measure}(< \mathsf{none}, \mathit{StdBasis} >, < (\mathit{Quantum}, [1]), (\mathit{GQuantum}, [1]) >)}\ \underline{\textbf{return}\ \bullet;}\ \underline{\circ_{\mathrm{L}}}\ \underline{\circ_{\mathrm{M}}})]$$

$$\downarrow_v \text{OP-D{\scriptsize O}M{\scriptsize EASURE}}$$

$$0.5 \bullet [((\begin{pmatrix}1 & 0\\ 0 & 0\end{pmatrix}, [2]), []) \mid (([], [(\mathit{Quantum}, [1]) \mapsto (\mathit{GQuantum}, [1])], [], []), [\blacksquare \circ_{\mathrm{G}} [[\square \circ_{\mathrm{L}} \square] \circ_{\mathrm{L}} [\mathsf{q} \mapsto (\mathit{Quantum}, [1])]]],$$
$$\underline{< \mathsf{none}, 0 >}\ \underline{\textbf{return}\ \bullet;}\ \underline{\circ_{\mathrm{L}}}\ \underline{\circ_{\mathrm{M}}})]$$
$$\boxplus\, 0.5 \bullet [((\begin{pmatrix}0 & 0\\ 0 & 1\end{pmatrix}, [2]), []) \mid (([], [(\mathit{Quantum}, [1]) \mapsto (\mathit{GQuantum}, [1])], [], []), [\blacksquare \circ_{\mathrm{G}} [[\square \circ_{\mathrm{L}} \square] \circ_{\mathrm{L}} [\mathsf{q} \mapsto (\mathit{Quantum}, [1])]]],$$
$$\underline{< \mathsf{none}, 1 >}\ \underline{\textbf{return}\ \bullet;}\ \underline{\circ_{\mathrm{L}}}\ \underline{\circ_{\mathrm{M}}})]$$

$$^{0.5}\swarrow \quad \text{NP-P{\scriptsize ROB}E{\scriptsize VOL}} \quad \searrow^{0.5}$$

Evolution of measurement branch 0 Evolution of measurement branch 1

Figure 5 (continued): Program example: Random number generator execution (to be continued)

We continue showing the program evolution of the branch where the measurement returned the value 0 only. The other measurement branch evolves obviously the same way, the only difference is in the measured value and the global quantum state.

$$\downarrow^{0.5} \text{NP-}\textsc{ProbEvol}$$

$$[((\begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}, [2]), []) \mid (([], [(\mathit{Quantum}, [1]) \mapsto (\mathit{GQuantum}, [1])], [], []), [\blacksquare \circ_G [[\square \circ_L \square] \circ_L [\mathsf{q} \mapsto (\mathit{Quantum}, [1])]]], \underline{<\mathsf{none}, 0>}\ \underline{\textbf{return}\ \bullet;}\ \underline{\circ_L}\ \underline{\circ_M})]$$

$$\downarrow_e \text{OP-}\textsc{SubstS}$$

$$[((\begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}, [2]), []) \mid (([], [(\mathit{Quantum}, [1]) \mapsto (\mathit{GQuantum}, [1])], [], []), [\blacksquare \circ_G [[\square \circ_L \square] \circ_L [\mathsf{q} \mapsto (\mathit{Quantum}, [1])]]], \underline{\textbf{return}\ <\mathsf{none}, 0>;}\ \underline{\circ_L}\ \underline{\circ_M})]$$

$$\downarrow_v \text{OP-}\textsc{ReturnValue}$$

$$[((\begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}, [2]), []) \mid (([], [(\mathit{Quantum}, [1]) \mapsto (\mathit{GQuantum}, [1])], [], []), [\blacksquare \circ_G [[\square \circ_L \square] \circ_L [\mathsf{q} \mapsto (\mathit{Quantum}, [1])]]], \underline{<\mathsf{none}, 0>})]$$

Figure 5 (continued): Program example: Random number generator execution

Code

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

Block structure

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

Program flow

fork	::=	**fork** methodCall **;**
return	::=	**return** (expr)? **;**

Conditionals and loops

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

Communication

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

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.*

Figure 6: Concrete syntax