Operational Semantics and Type Soundness of Quantum Programming Language LanQ

Hynek Mlnařík*
Faculty of Informatics, Masaryk University
Brno, Czech Republic
hmlnarik@mail.muni.cz

June 14, 2007

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 and a proof of type soundness of the noncommunicating part of the language. We provide an example run of a quantum random number generator.

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

Contents

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 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 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. In the present paper, we define its syntax, operational semantics, and prove type soundness of the noncommunicating part of the language.

The paper is structured as follows: we start with an example of an program written in LanQ in Section 2. We then formally define its concrete (Section 3) and internal syntax (Section 4). Then basic concepts used later in the paper are defined in Section 6, followed by typing system in Section 5.1. In Section 7, we define the operational semantics of the language

and prove its type soundness in Section 8. 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 implementation of a well-known multiparty quantum protocol – teleportation [BBC+93]. Teleportation can be written as the program shown in 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 takes no parameters and it returns no value what can be seen from the word void in front of the method name. The method *angela*() has to be invoked with two parameters – one end of a channel that can be used to transmit values of type int, and one qubit (*ie.* a quantum bit). It also returns no value. The method *bert*() takes a channel end and a qubit, and returns a value of type int.

The method *main*() declares variables $\psi_A, \psi_B, \psi_{EPR}, c, c_0$ and c_1 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 two-qubit compound system $\psi_A \otimes \psi_B$. Channel c capable of transmitting integers is declared on the next line. The ends of the channel are named c_0 and c_1.

On next lines, the method **main** invokes method *createEPR*() which creates an EPR-pair, and stores the returned reference to the created pair into the variable ψ_{EPR}. After that, a new channel is allocated and assigned to the variable c. This also modifies the channel end variables c_0 and c_1. The next command makes the running process split into two. One of the processes continues its run and invokes the method *angela*(). The second process starts its run from the method *bert*().

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

The method *bert*() receives one channel end and one qubit as arguments. After declaring a 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 on qubit *stto*. Then, it invokes a method *doSomethingElse*() and passes the variable *stto* as an argument to this method. Finally, it returns the value of the variable i to the caller.

3 Concrete syntax

In this section, we introduce concrete syntax of LanQ programs. This syntax is used to write programs by a programmer. Semantics is defined using internal syntax which is described later (see Section 4).

The syntax is shown in Figure 3. Reserved words of the language are written in **bold** and the identifier names are in CAPITALS. Grammar is given in nondeterministic extended Backus-Naur form (EBNF). The root of grammar is the nonterminal *program*.

For the sake of clarity, the concrete grammar nonterminals names are long and descriptive to indicate their meaning. We describe meaning of the most important nonterminals here: *program* (words derived from this nonterminal represent LanQ *programs*), *code* (*statements*), *pExpr* (*promotable expressions*, *ie.* expressions that can act as statements), *methodCall* (*method calls*), *methodParams* (*method parameters*), *assignment* (*assignments*), *measurement* (*measurements*), *expr* (*expressions*), *indivExpr* (*individual expressions*, *ie.* expressions not containing any operators), *op* (*operators*), *method* (*method declarations*), *block* (*blocks* of code), *seq* (*block-forming statements*, *ie.* statements that can be used in blocks), and *varDeclaration* (*variable declarations*). The other nonterminals are auxiliary and their meaning is obvious.

Definition 3.1. *Let m be a method declaration. We call the part of m which was derived from nonterminal methodHeader a* method header*, and the part of m which was derived from nonterminal block a* method body*.*

In the following example, a method named $mName$ is declared. The parts of the method declaration are annotated on the right side.

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

Figure 2: Declaration of a method named $mName$

4 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. To define operational semantics, we need a representation for the program execution – a syntax that allows us to evaluate a program by means of rewriting of program terms. The rewriting rules are presented later in Section 7 where the operational semantics is defined.

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	::=	+ | − | ⊗ | ...

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** | **bool** | ...
nonDupType	::=	**channelEnd**[*nonVoidType*] | *channelType* | *qType*
channelType	::=	**channel**[*nonVoidType*]
qType	::=	*qBasicType*(⊗*qType*)?
qBasicType	::=	**qbit** | **qtrit** | ...

Figure 3: Concrete syntax

The internal syntax is defined in Figure 4. The syntax is similar to the concrete one while not containing declarative parts of the concrete syntax and being abbreviated. In the internal syntax, we define the following basic syntactic entities: *numbers* (N), *lists* (L), *recursive lists* (RL), *references* (R), *constants* (C), *identifiers* (I), *types* (T), and *internal values* ($\mathbf{v}$).

Promotable expressions (PE) are expressions that can act as statements when postfixed by semicolon. *Expressions* (E) can evaluate to an internal value. The syntactic classes of *variable declarations* (VD) and *statements* (S) can together create a block. Therefore they are together called *block-forming elementary statements* (Be). A *block-forming statement* (B) is built from zero or more such block-forming elementary statements.

Remark 4.1. *For the sake of clarity, we use the following notation in the rule body. We denote by $\bar{S}$ an abbreviation of BNF rule body "$(S)*$", and by $\widetilde{E}$ an abbreviation of "$(E\ (\mathbf{,}\ E)*)?$".*

$$
\begin{array}{lcl}
N & ::= & 0 \mid 1 \mid \ldots \\
L & ::= & [] \mid [\widetilde{N}] \\
RL & ::= & L \mid [\widetilde{RL}] \\
R & ::= & \mathsf{none} \mid \mathbf{(Classical,}N\mathbf{)} \mid \mathbf{(Quantum,}RL\mathbf{)} \mid \mathbf{(Channel,}N\mathbf{)} \mid \\
 & & \mathbf{(ChannelEnd_0,}N\mathbf{)} \mid \mathbf{(ChannelEnd_1,}N\mathbf{)} \mid \mathbf{(GQuantum,}L\mathbf{)} \mid \mathbf{(GChannel,}N\mathbf{)} \\
C & ::= & R \mid \mathbf{true} \mid \mathbf{false} \mid \bot \mid \ldots \\
I & ::= & x \mid y \mid z \mid \ldots \mid + \mid - \mid \ldots \\
T & ::= & \mathsf{void} \mid \mathsf{int} \mid \mathsf{qbit} \mid \mathsf{channel}[T] \mid \mathsf{channelEnd}[T] \mid T \otimes T \mid \ldots \\
\mathbf{v} & ::= & \langle\!\langle R, C \rangle\!\rangle_T \\
 & & \\
PE & ::= & \mathbf{new}\ T\mathbf{()} \mid I = E \mid I(\widetilde{E}) \mid \mathbf{measure}(\widetilde{E}) \mid \mathbf{recv}(E) \\
E & ::= & I \mid \mathbf{v} \mid (E) \mid PE \\
VD & ::= & T\ \widetilde{I}\mathbf{;} \mid \mathsf{channel}[T]\ I\ \mathbf{withends}[I\mathbf{,}I]\mathbf{;} \mid I\ \mathbf{aliasfor}\ [\widetilde{I}]\mathbf{;} \\
S & ::= & \mathbf{;} \mid PE\mathbf{;} \mid \{B\} \mid \mathbf{if}\ (E)\ S\ \mathbf{else}\ S \mid \mathbf{while}\ (E)\ S \mid \\
 & & \mathbf{return;} \mid \mathbf{return}\ E\mathbf{;} \mid \mathbf{fork}\ I(\widetilde{E})\mathbf{;} \mid \mathbf{send}(E\mathbf{,}E)\mathbf{;} \\
Be & ::= & VD \mid S \\
B & ::= & \overline{Be}
\end{array}
$$

Figure 4: Internal syntax

Configuration syntax specifies formal notation of process configuration which is described in Subsection 7.3.

If a statement or an expression contains a subexpression, this subexpression is evaluated separately and the evaluation result is substituted in place of the subexpression. For this reason, we introduce a concept of a *hole* ($\bullet$) which stands for the awaited result of subexpression evaluation. We call a term not containing a hole a *closed term*.

The terms containing a hole are defined by nonterminals Sc and Ec which represent partially evaluated statements and expressions, respectively, whose subexpression is being evaluated. In other words, they represent *evaluation contexts*. We also define syntactic entities for *runtime errors* ($RTErr$) and *term stack elements* ($StkEl$).

Before a method can be invoked to be run, we must transform its body to the internal representation. 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 ;**,
- In internal representation, all constants C are represented by internal values $\langle\!\langle \mathsf{none}, C \rangle\!\rangle_{\mathsf{T}}$ where T is the type of the constant C,
- In internal representation, all operators are written in the prefix notation and seen as method calls, *ie.* $E\ \odot\ F$ is converted to $\odot\mathbf{(}E\mathbf{,}F\mathbf{)}$.

$$
\begin{array}{lcl}
Ec & ::= & I = \bullet \mid I(\widetilde{\mathbf{v}}, \bullet, \widetilde{E}) \mid \mathbf{measure}(\widetilde{\mathbf{v}}, \bullet, \widetilde{E}) \mid \mathbf{recv}(\bullet) \\
Sc & ::= & \bullet; \mid \mathbf{if}\ (\bullet)\ S\ \mathbf{else}\ S \mid \mathbf{fork}\ I(\widetilde{\mathbf{v}}, \bullet, \widetilde{E}); \mid \mathbf{send}(\bullet, E); \mid \mathbf{return}\ \bullet; \\
RTErr & ::= & \mathbf{UV} \mid \mathbf{OQV} \mid \mathbf{ISQV} \\
\\
StkEl & ::= & B \mid E \mid Ec \mid Sc \mid RTErr \mid \circ_{\mathbf{L}} \mid \circ_{\mathbf{M}} \\
LMS & ::= & \text{local memory state as described in Subsection 7.3} \\
VP & ::= & \text{variable properties as described in Subsection 7.3} \\
\\
LS & ::= & (LMS, VP, \overline{StkEl}) \\
P & ::= & LS \mid LS \parallel P \\
GS & ::= & ((DM, L), [I \overset{\frown}{\models\!\!=\!\!\dashv} I]) \quad \text{where } DM \text{ is a density matrix} \\
Sys & ::= & [GS \mid P] \\
Univ & ::= & p \bullet Sys \mid p \bullet Sys \boxplus Univ
\end{array}
$$

Figure 5: Configuration syntax

There is obviously an algorithm which rewrites any method body derived using the concrete syntax to the internal representation.

An example of a block written using the concrete syntax and its internal representation is shown in Figure 6.

```
{
    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 6: Block derived using concrete syntax (a) and the same block converted to internal syntax (b).

5 Typing

5.1 Typing rules

LanQ is a typed language. This feature enables us to detect errors arising from incorrect usage of methods, variables and constants at compile time.

First, we define basic types used in LanQ: void (a type with only one value: $\perp$), int (a type of integers), bool (a type of truth values **true** and **false**), $\mathsf{Q_n}$ (a type of references to n-dimensional quantum systems), Ref (a type of references, defined later in Subsection 7.1), RTErr (a type of runtime errors), and MeasurementBasis (a type of measurement bases).[1] If T is a type, then channel[T] is a type of references to a channel capable of transmitting values

[1] The type system can be indeed extended when needed.

of type T, and $\mathsf{channelEnd[T]}$ is a type of references to ends of such a channel. Further, let $\mathsf{S_1}, ..., \mathsf{S_n}, \mathsf{S}$ be types for $n \geq 0$. Then a *method type* T is defined to be the type $\mathsf{S_1}, \mathsf{S_2}, ..., \mathsf{S_n} \rightarrow \mathsf{S}$. We call types $\mathsf{S_1}, \ldots, \mathsf{S_n}$ *argument types* and the type S a *return type*.

Definition 5.1. *We define a set Types of types of classical values. We denote by* val(T) *a set of values of type* T *and define a set Values as a set of values of all types:*

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

After parsing a program, the method declarations are stored in a triplet (M_T, M_H, M_B) of partial functions. We call this triplet a *method typing context* where:

- $M_T(m)$ which returns the method type for a method m (the method type is straightforwardly determined from the method header),
- $M_H(m)$ returns the method header for a method m, and
- $M_B(m)$ which returns the method body represented using internal syntax for a method m.

We provide typechecking rules in Figures 7, 8, 9 and 10. These rules use a *typing context* which is a pair $(M; \Gamma)$ consisting of:

- M is a method typing context,
- Γ is a *variable typing context* – a partial mapping $\Gamma : Names \rightharpoonup Types$ that assigns a type to a variable name. We write a variable typing context Γ as $\Gamma = a_1 : \mathsf{T_1}, \ldots, a_n : \mathsf{T_n}$ meaning that the type T_i is assigned to the variable a_i.

  The extension of a context Γ by a variable b of type T_b is written as $\Gamma, b : \mathsf{T}_b$. It is undefined if $\Gamma(b)$ is defined and $\Gamma(b) \neq \mathsf{T}_b$. Otherwise it is defined as:

$$(\Gamma, b : \mathsf{T}_b)(x) = \begin{cases} \mathsf{T}_b & \text{if } x = b, \\ \Gamma(x) & \text{otherwise.} \end{cases}$$

Let P be a program whose internal representation is stored in a method typing context (M_T, M_H, M_B). We call this program *well-typed* if the premise of the rule T-Program is satisfied for this method typing context. This check is passed iff all the declared methods satisfy the typing rule T-Method.

Typechecking of a method in the rule T-Method is a little more complicated. The reason is that we require any method m whose return type is not void to return a value of appropriate type in all possible control paths which the evaluation of this method can take. The return type is for the sake of typechecking stored in the formal variable $@retVal$.

This can be checked at compile time. We split this requirement into two:

(1) during evaluation, the method m body always reaches a **return** statement (or invokes a runtime error or diverges), and

(2) any value returned by a **return** statement during evaluation of the method m body is of appropriate type. This is checked by typing rules T-ReturnVoid and T-ReturnExpr in cooperation with T-Method.

For formal definition of the condition (1), we define a predicate *RetOk*.

T-PROGRAM
$$\frac{\forall m \in \mathrm{dom}(M_T):\ (M_T, M_H, M_B) \vdash_T M_H(m) : M_T(m)}{\vdash (M_T, M_H, M_B)}$$

T-METHOD
$$\frac{\begin{array}{l} T = \mathsf{void} \vee \mathit{RetOk}(M_B(m)), \\ (M_T, M_H, M_B); a_1 : T_1, \ldots, a_n : T_n, @retVal : T \vdash_T M_B(m) : \mathsf{void} \end{array}}{(M_T, M_H, M_B) \vdash_T T\ m(T_1\ a_1, \ldots, T_n\ a_n) : T_1, \ldots, T_n \rightarrow T}$$

Figure 7: Typing rules for program and method declaration

T-VARDECL
$$\frac{M; \Gamma, I_0 : T, \ldots, I_n : T \vdash_T B : \mathsf{T}}{M; \Gamma \vdash_T T\ I_0, \ldots, I_n; B : \mathsf{T}}$$

T-VARDECLCHE
$$\frac{M; \Gamma, I_0 : \mathsf{channel}[T], I_1 : \mathsf{channelEnd}[T], I_2 : \mathsf{channelEnd}[T] \vdash_T B : \mathsf{T}}{M; \Gamma \vdash_T \mathsf{channel}[T]\ I_0\ \textbf{withends}\ [I_1, I_2]; B : \mathsf{T}}$$

T-VARDECLALF
$$\frac{\begin{array}{l} \forall i \in \{1, \ldots, n\} : M; \Gamma \ \vdash_T I_i : \mathsf{T_i} \text{ where } \mathsf{T_i} \text{ is a quantum type}, \\ M; \Gamma, I_0 : \bigotimes_{\mathsf{i=1}}^{\mathsf{n}} \mathsf{T_i} \ \vdash_T B : \mathsf{T} \end{array}}{M; \Gamma \vdash_T I_0\ \ \textbf{aliasfor}\ [I_1, \ldots, I_n]; B : \mathsf{T}}$$

Figure 8: Typing rules for variable declarations

T-SKIP
$$\frac{}{M; \Gamma \vdash_T \textbf{;} : \mathsf{void}}$$

T-PROMOEXPR
$$\frac{M; \Gamma \vdash_T PE : \mathsf{T}}{M; \Gamma \vdash_T PE\textbf{;} : \mathsf{void}}$$

T-BLOCK
$$\frac{M; \Gamma \vdash_T B : \mathsf{void}}{M; \Gamma \vdash_T \textbf{\{}B\textbf{\}} : \mathsf{void}}$$

T-BLOCKHEAD
$$\frac{M; \Gamma \vdash_T Be_0 : \mathsf{void} \quad M; \Gamma \vdash_T Be_1\ \ldots\ Be_n : \mathsf{void}}{M; \Gamma \vdash_T Be_0\ Be_1\ \ldots\ Be_n : \mathsf{void}} \quad \text{where } Be_0 \neq VD$$

T-IF
$$\frac{M; \Gamma \vdash_T E : \mathsf{bool} \quad M; \Gamma \vdash_T S_0 : \mathsf{void} \quad M; \Gamma \vdash_T S_1 : \mathsf{void}}{M; \Gamma \vdash_T \textbf{if}\ (E)\ S_0\ \textbf{else}\ S_1 : \mathsf{void}}$$

T-WHILE
$$\frac{M; \Gamma \vdash_T E : \mathsf{bool} \quad M; \Gamma \vdash_T S : \mathsf{void}}{M; \Gamma \vdash_T \textbf{while}\ (E)\ S : \mathsf{void}}$$

T-RETURNVOID
$$\frac{}{M; \Gamma, @retVal : \mathsf{void} \vdash_T \textbf{return;}\ : \mathsf{void}}$$

T-RETURNEXPR
$$\frac{M; \Gamma, @retVal : \mathsf{T} \vdash_T E : \mathsf{T}}{M; \Gamma \vdash_T \textbf{return}\ E\textbf{;}\ : \mathsf{void}}$$

T-FORK
$$\frac{M; \Gamma \vdash_T I(\widetilde{E}) : \mathsf{T} \quad I \text{ is a classical method}}{M; \Gamma \vdash_T \textbf{fork}\ I(\widetilde{E})\textbf{;}\ : \mathsf{void}}$$

T-SEND
$$\frac{M; \Gamma \vdash_T E_0 : \mathsf{channelEnd[T]} \quad M; \Gamma \vdash_T E_1 : \mathsf{T}}{M; \Gamma \vdash_T \textbf{send}(E_0, E_1)\textbf{;}\ : \mathsf{void}}$$

Figure 9: Typing rules for statements

T-VAR
$$\frac{}{M;\Gamma, I : \mathsf{T} \vdash_T I : \mathsf{T}}$$

T-VALUE
$$\frac{}{M;\Gamma \vdash_T \langle\!\langle R, C \rangle\!\rangle_{\mathsf{T}} : \mathsf{T}}$$

T-BRACKET
$$\frac{M;\Gamma \vdash_T E : \mathsf{T}}{M;\Gamma \vdash_T (E) : \mathsf{T}}$$

T-ALLOC
$$\frac{\mathsf{T} \text{ is either a quantum type } (\mathsf{Q_d}) \text{ or a channel type } (\mathsf{channel[T]})}{M;\Gamma \vdash_T \textbf{new } \mathsf{T}() : \mathsf{T}}$$

T-ASSIGN
$$\frac{M;\Gamma, I : \mathsf{T} \vdash_T E : \mathsf{T}}{M;\Gamma \vdash_T I = E : \mathsf{T}}$$

T-METHODCALL
$$\frac{M_T(I) = \mathsf{S}_0, \ldots, \mathsf{S}_n \to \mathsf{T} \quad M;\Gamma \vdash_T E_0 : \mathsf{S}_0 \quad \ldots \quad M;\Gamma \vdash_T E_n : \mathsf{S}_n}{M;\Gamma \vdash_T I(E_0, \ldots, E_n) : \mathsf{T}}$$
where $M = (M_T, M_H, M_B)$

T-MEASUREMENT
$$\frac{\begin{array}{c} M;\Gamma \quad \vdash_T E_0 : \mathsf{MeasurementBasis}, \\ \forall i \in \{1, \ldots, n\} : M;\Gamma \quad \vdash_T E_i : \mathsf{T_i} \text{ where } \mathsf{T_i} \text{ is a quantum type} \end{array}}{M;\Gamma \vdash_T \textbf{measure}(E_0, E_1, \ldots, E_n) : \mathsf{int}}$$

T-RECV
$$\frac{M;\Gamma \vdash_T E : \mathsf{channelEnd[T]}}{M;\Gamma \vdash_T \textbf{recv}(E) : \mathsf{T}}$$

Figure 10: Typing rules for expressions

Definition 5.2. *Let B be a block-forming statement. We define a predicate RetOk as:*

$$RetOk(B) \quad = \quad \begin{cases} \textbf{true} & \textit{if } B = \textbf{return } E\textbf{;} \\ RetOk(S_t) \wedge RetOk(S_e) & \textit{if } B = \textbf{if } (E)\ S_t \textbf{ else } S_e \\ \bigvee_{Be_i} RetOk(Be_i) & \textit{if } B = Be_0\ Be_1\ \ldots\ Be_n \\ RetOk(B') & \textit{if } B = \{\ B'\ \} \\ \textbf{false} & \textit{otherwise} \end{cases}$$

This predicate does not handle the case $B = \textbf{while } (E)\ S$ because evaluation of the condition E is undecidable at compile-time. Hence even for the straightforwardly always-terminating case $B = \textbf{while (true) return } 1\textbf{;}$, $RetOk(B)$ is not satisfied. Therefore the predicate is only approximate.

Later we prove a lemma stating that if the predicate *RetOk* is satisfied on B then any control path of evaluation of B reaches a **return;** or **return** E**;** statement, or a runtime error, or diverges (see Lemma 8.15). Thus, if B is a method m body and $RetOk(B)$ is satisfied, the evaluation of method can either reach some **return** statement, lead to a runtime error, or diverge.

Definition 5.3. *We call a method m* well-typed *if the premises of rule* T-METHOD *are satisfied for this method.*

Remark 5.4. *Note that if a method m is well-typed and its return type is not* void *then its body contains only* **return** E**;** *statements, ie. no* **return;** *statements.*

The rest of the typing rules is usual: The rules for typechecking variable declarations in Figure 8 check the block-forming statement can be typechecked with the variable context extended with the newly declared variables.

We formally regard all statements to be of type void what is seen in the typechecking rules in Figure 9. These rules are quite usual up to the rule T-Fork. This rule requires that the method, which should be a new process run from, is classical. This is natural requirement as running a new process, which is by its nature a classical object from a quantum operator, would be a nonsense.

The typechecking rules for expressions shown in Figure 10 are designed as usual.

6 Basic concepts

Before we continue with formal definition of the semantics, we must define several useful functions and structures. First we define notation used in the rest of the article.

6.1 Notation

Notation 6.1. *Let S be a set, $\perp \notin S$. Then $S_\perp = S \cup \{\perp\}$. We denote a set of natural numbers with zero $\mathbb{N} \cup \{0\}$ by $\mathbb{N}_0$.*

Definition 6.2. *Let S be a set. An S-*list *$s = [s_1, \ldots, s_n]$ is a list where $n \in \mathbb{N}_0$ and $s_1, \ldots, s_n \in S$. Set of all finite S-lists $\{s \mid s$ is a finite S-list$\}$ is denoted by $S_{[]}$.*

Definition 6.3. *Let $m, n \in \mathbb{N}_0$. Let $L = [l_1, \ldots, l_n], K = [k_1, \ldots, k_m]$ be lists. Then $|L|$ is a* length *of a list L, $|L| = n$. Concatenation of lists L and K is defined as $L \cdot K = [l_1, \ldots, l_n, k_1, \ldots, k_m]$. Set of list L elements is defined as $\mathrm{set}(L) = \{l_1, \ldots, l_n\}$.*

6.2 Reference-related concepts

We use the following specially formed lists for storing references to quantum systems.

Definition 6.4. *Let S be a set. We define a* recursive *S-*list *recursively as:*

- *Any S-list $[s_1, \ldots, s_k]$ is a recursive S-list,*
- *A list $[e_1, \ldots, e_m]$ is a recursive S-list for any $m \in \mathbb{N}_0$ if $e_1, \ldots, e_m$ are recursive S-lists.*

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

Recursive S-lists are used for the representation of quantum system references in the following way:

A reference to a quantum system, be it compound or not, is specified by a recursive $\mathbb{N}$-list. Quantum systems are stored in indexed registers in the quantum memory, one quantum system per one register. The (unique) index is assigned to a quantum system when it is allocated. The reference to the system with index n is a recursive $\mathbb{N}$-list $[n]$.

Let us have two quantum systems ϕ and ψ whose indices are 1 and 2, respectively. The references to these quantum systems are $r_\phi = [1]$ and $r_\psi = [2]$, respectively. A reference to a compound system ρ consisting of the two quantum systems ϕ and ψ is then a recursive $\mathbb{N}$-list $r_\rho = [r_\phi, r_\psi] = [[1], [2]]$. Note that the structure of ρ, *ie.* that it consists of *two* systems, corresponds to the structure of the reference r_ρ – it is built up from *two* elements.

Notation 6.5. *The set of all finite recursive S-lists $\{s \mid s$ is a finite recursive S-list$\}$ is denoted by $S_{[\circlearrowright]}$.*

Recursive S-lists allow us to nicely capture quantum system structure in the reference. However, when working with referred quantum systems, *eg.* applying some unitary operator, we do not want to bother with the structure – we only need a list of indices of the affected quantum systems. To get such a *linearized* list out of the structured one, we define the following function:

Definition 6.6. *For a recursive S-list $s \in S_{[\circlearrowright]}$, we define a function* linearize : $S_{[\circlearrowright]} \to S_{[]}$ *which converts a recursive S-list into an S-list:*

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

If a reference R contains a special element $\perp$ which denotes a reference to nonexistent quantum system, we consider the reference R itself to refer to nonexistent quantum system. Hence linearization of such a reference returns $\perp$:

Definition 6.7. *For a set $S_\perp$, we define a function* $\text{linearize}_\perp : (S_\perp)_{[\circlearrowright]} \to (S_\perp)_{[]} \cup \{\perp\}$ *as:*

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

Definition 6.8. *Let S be a set. We define a function* $\text{set}_{[\circlearrowright]} : S_{[\circlearrowright]} \to S$ *for getting the set of recursive list items regardless of the structure as* $\text{set}_{[\circlearrowright]}(l) \stackrel{\text{def}}{=} \text{set}(\text{linearize}(l))$. *We also define function* $|-|_{[\circlearrowright]} : S_{[\circlearrowright]} \to \mathbb{N}$ *for getting length of a recursive list regardless of the structure as* $|l|_{[\circlearrowright]} \stackrel{\text{def}}{=} |\,\text{linearize}(l)|$.

6.3 Variable-related concepts

We use partial functions to capture variable properties, *eg.* mapping a variable name to a place in memory where the variable value is stored, or a mapping to the variable type. We define several useful functions for handling these partial functions describing variables.

Adding a new variable to a set of known variables is represented by extending the domain of the appropriate partial function with the new variable name. We call this function an *update*. Sometimes we only want to update a variable property if the updated variable is already contained in the domain of the updated function, *eg.* change a memory reference referred by the variable. We call such an operation an *replacement*. In general, we define these functions in the following way:

Definition 6.9. *Let $f : X \rightharpoonup Y$ be a partial function. For $x \in X, y \in Y$, we define* replacement $f[x \mapsto y] : X \rightharpoonup Y$ *and* update $f[x \mapsto y]_+ : X \rightharpoonup Y$ *as:*

$$f[x \mapsto y](z) = \begin{cases} y & \textit{if } x = z \textit{ and } f(x) \textit{ is defined,} \\ f(z) & \textit{otherwise,} \end{cases}$$

and

$$f[x \mapsto y]_+(z) = \begin{cases} y & \textit{if } x = z, \\ f(z) & \textit{otherwise.} \end{cases}$$

Note that the $f[x \mapsto y](x)$ is defined iff $f(x)$ is defined while $f[x \mapsto y]_+(x)$ is defined even if $f(x)$ is not defined.[2]

We need to store different variable properties: variable type, names of channel ends corresponding to given channel *etc.* Each property is represented by a separate partial mapping. Hence variable properties are described by a tuple of such partial functions.

Definition 6.10. *A* partial function tuple *is a tuple $f = (f_0, \ldots, f_n)$ where $f_0, \ldots, f_n$ are partial functions.*

[2] The + sign in function index means "add the mapping from x to y even if it was not defined yet".

We will need to capture variable scope during a method evaluation. For this reason, we define concepts of *list of partial function tuples* and *list of lists of partial function tuples*. Their usage is in more detail explained in Subsection 7.2. We also extend update and replacement functions to lists of partial function tuples and lists of lists of partial function tuples.

Definition 6.11. *We define* list of partial function tuples *recursively as:*

- $\square$ *is an (empty) list of partial function tuples,*
- $[K \circ_L f]$ *is a list of partial function tuples if* K *is a list of partial function tuples and* f *is a partial function tuple. The symbol* $\circ_L$ *serves as an element separator only.*[3]

Definition 6.12. *For a list of partial function tuples* K*, we define a* replacement $K[x \mapsto y]_i$ *and an* update $K[x \mapsto y]_{+,i}$ *of outermost* x *in an* i*-th partial function of a list of partial function tuples* K *as:*

$$
\begin{aligned}
[K \circ_L (f_0, \ldots, f_n)][x \mapsto y]_i &\stackrel{\text{def}}{=} \begin{cases} [K \circ_L (f_0, \ldots, f_i[x \mapsto y], \ldots, f_n)] & \textit{if } f_i(x) \textit{ is defined} \\ [K[x \mapsto y]_i \circ_L (f_0, \ldots, f_n)] & \textit{otherwise} \end{cases} \\
\square[x \mapsto y]_i &\stackrel{\text{def}}{=} \square \\
\\
[K \circ_L (f_0, \ldots, f_n)][x \mapsto y]_{+,i} &\stackrel{\text{def}}{=} [K \circ_L (f_0, \ldots, f_i[x \mapsto y]_+, \ldots, f_n)] \\
\square[x \mapsto y]_{+,i} &\stackrel{\text{def}}{=} \square
\end{aligned}
$$

Definition 6.13. *We define a* list of lists of partial function tuples *recursively as:*

- $\blacksquare$ *is an (empty) list of lists of partial function tuples,*
- $[L_1 \circ_G K]$ *is a list of lists of partial function tuples if* L_1 *is a list of lists of partial function tuples and* K *is a list of partial function tuples. The symbol* $\circ_G$ *serves as an element separator only.*[4]

Definition 6.14. *For a list of lists of partial function tuples* L*, 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:*

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

Last, we define a coalesce[5] function $*$ of two partial functions $f, g : A \rightharpoonup B$. Coalesce $g * f$ is a partial function which for $(g * f)(x)$ results into $f(x)$ if $f(x)$ is defined, otherwise to $g(x)$:

[3] L in $\circ_L$ stands for *local*.

[4] G in $\circ_G$ stands for *global*.

[5] The name *coalesce* is given because this function is similar to the COALESCE function defined in SQL-92 standard.

Definition 6.15. *Let $f, g : A \rightharpoonup B$ be partial functions. We define a* coalesce of g and f $g * f : A \rightharpoonup B$ *as:*

$$(g * f)(x) = \begin{cases} f(x) & \text{if } f(x) \text{ is defined} \\ g(x) & \text{otherwise.} \end{cases}$$

Definition 6.16. *We define Names to be a set of all identifier names.*

7 Operational semantics

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

7.1 Memory model

In this subsection, we describe the memory model used in LanQ implementation.

Our model abstract machine uses a memory to store values. As we work both with duplicable and nonduplicable data, we have two types of memory: *system* and *local*. All processes manage their own memory – a *local process memory* where duplicable values are stored. System manages the system memory where nonduplicable resources are stored. Processes cannot access the system memory directly, they work with resources by means of references to the system memory. This is transparent to the programmer. Memory model is depicted in Figure 11.

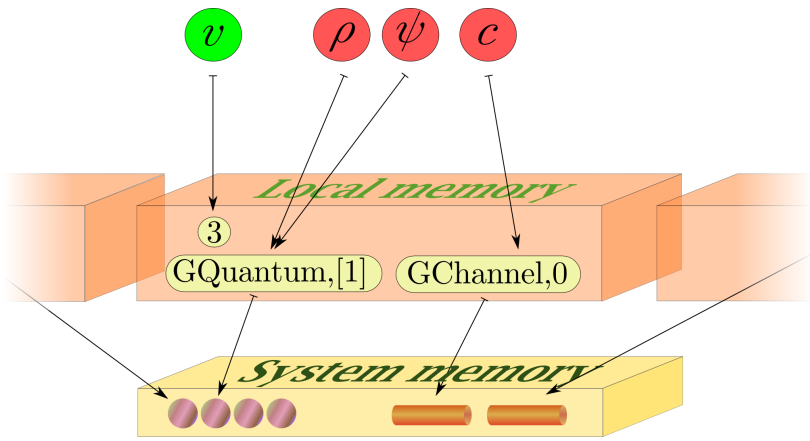


Figure 11: Memory model of LanQ. Duplicable variables are shown in a green circle, nonduplicable ones in a red circle. The variables v (integer), ρ (quantum system), ψ (quantum system) and c (channel) belonging to one process refer to places in its local memory shown as yellow containers. The containers refer to values – a value 3 in the case of variable v container and references to the system memory in the rest of the cases.

A memory reference specifies a position of a value in the memory. We define none to be a special reference that refers to no value. We distinguish *global* and *local* references[6]:

- References to system channel memory: $Ref_{GCh} = \{\bot\} \cup (\{GChannel\} \times \mathbb{N}_{\bot})$,
- References to system quantum memory: $Ref_{GQ} = \{\bot\} \cup (\{GQuantum\} \times \mathbb{N}_{[]})$,
- References to local classical value memory: $Ref_{Cl} = \{\mathsf{none}\} \cup (\{Classical\} \times \mathbb{N}_{\bot})$,

[6]Global references are references to a system memory, local references are references to a local process memory.

- References to local quantum systems reference memory:
  $Ref_Q = \{\mathsf{none}\} \cup (\{Quantum\} \times (\mathbb{N}_\perp)_{[\circlearrowleft]})$,
- References to local channel reference memory: $Ref_{Ch} = \{\mathsf{none}\} \cup (\{Channel\} \times \mathbb{N})$,
- References to local channel end reference memory:
  $Ref_{ChE} = \{\mathsf{none}\} \cup (\{ChannelEnd_0, ChannelEnd_1\} \times \mathbb{N})$.

We define a set of global references $Ref_G = Ref_{GCh} \cup Ref_{GQ}$, a set of local references $Ref_L = Ref_{Cl} \cup Ref_Q \cup Ref_{Ch} \cup Ref_{ChE}$, and a set of all references $Ref = Ref_G \cup Ref_L$. A set of references to nonduplicable values is denoted by $Ref_{nd} = Ref_Q \cup Ref_{Ch} \cup Ref_{ChE}$.

Definition 7.1. *A* memory reference *is an element from the set Ref. We define* Ref *to be the type of memory references.*

Remark 7.2. *Note that a memory reference is a classical value, therefore* $\mathsf{Ref} \in Types$.

7.2 Variable properties storage

In this subsection, we informally introduce structure used for handling variable properties.

Variable properties is a structure where properties of variables neccessary for correct handling the variables are stored while respecting their scope: The actually running method has access only to variables declared in this method, it cannot access any variable from the caller method. Moreover, the validity of a variable is limited to the block in which the variable is declared.

The properties of a variable are formally described later in Subsection 7.3. They are represented by partial functions which, given a variable name, return:

- A reference to local process memory where the value of the variable is stored,
- Variable names representing individual ends of the channel (if the given variable represents a channel),
- List of variable names representing quantum systems which are subsystems of the compound quantum system (if the given variable represents a compound quantum system),
- A type of the given variable.

Therefore we have a quadruple of partial functions that represent variable properties: $f = (f_{var}, f_{ch}, f_{qa}, f_{type})$. Indeed, this quadruple is not enough for handling variable scope.

Respecting a variable scope is achieved by using lists of partial function tuples and lists of such lists:

- *A variable can be accessed only from within the block where it was declared.* This is ensured by using a list of partial function 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 currently running method are accessible to this method.* This is ensured by using a list of lists of partial function tuples (separated by $\circ_G$) where a new list of partial function tuples is appended when a method is invoked, and removed when a method finishes.

We show the manipulation with a variable properties structure on an example. Consider the following method a:

```
  int a(int c)
1 {
2     bool b;
3     b = true;
4     if (b) {
5         int i;
          ⋮
6     }
7     return 3 + c;
8 }
```

We show the variable properties construction as the individual lines of the method are executed. As the formal notation of the variable properties is not well-readable, we also provide the reader with its graphical representation. The representation uses the following notation:

- A variable properties tuple $(f_{var}, f_{ch}, f_{qa}, f_{type})$ is represented as: $\left(\begin{array}{c} f_{var} \\ \hline f_{ch} \\ \hline f_{qa} \\ \hline f_{type} \end{array}\right)$,
- A list of variable properties $[K \circ_L f]$ is represented as: $\boxed{\begin{array}{c} f \\ K \end{array}}$,
- A list of lists of variable properties $[L_1 \circ_G K]$ is represented as: $\boxed{\begin{array}{c} K \\ L_1 \end{array}}$.

We assume that the original variable properties were vp_G right before calling the method a. When the method a is called, a list of variable properties $[\Box \circ_L ([c \mapsto r_c], [], [], [c \mapsto \mathsf{int}, @retVal \mapsto \mathsf{int}])]$ is appended to vp_G:

$$[vp_G \ \circ_G \ [\Box \circ_L ([c \mapsto r_c], [], [], [c \mapsto \mathsf{int}, @retVal \mapsto \mathsf{int}])]]$$

(see Figure 12(a)). In this appended list, method parameters values are passed to the called method; in our case, the method parameter c refers to the memory as set by the reference r_c.

On line 1, a new block is started, therefore a new empty variable properties tuple $\Diamond$ is appended:

$$[vp_G \circ_G [[\Box \circ_L ([c \mapsto r_c], [], [], [c \mapsto \mathsf{int}, @retVal \mapsto \mathsf{int}])] \ \circ_L \ \Diamond]]$$

(see Figure 12(b)). On the next line, a variable b is declared, hence the head element of the inner list is modified:

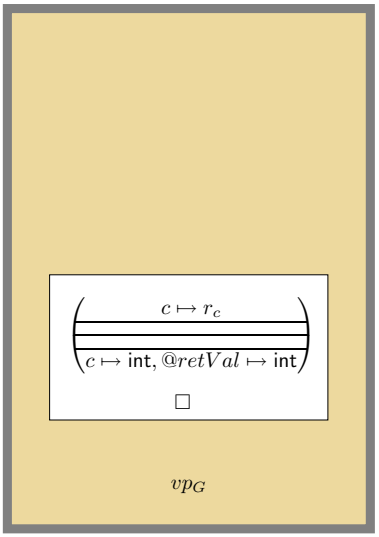
$$[vp_G \circ_G [[\Box \circ_L ([c \mapsto r_c], [], [], [c \mapsto \mathsf{int}, @retVal \mapsto \mathsf{int}])] \circ_L ([b \mapsto \mathsf{none}], [], [], [b \mapsto \mathsf{bool}])]]$$

(see Figure 12(c)). On line 3, b is assigned a value **true** which is stored in the memory in a place referred by a reference r_b. This modifies f_{var} element of the appropriate variable properties tuple:

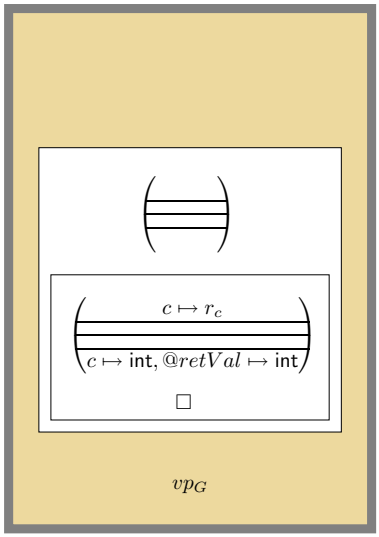
$$[vp_G \circ_G [[\Box \circ_L ([c \mapsto r_c], [], [], [c \mapsto \mathsf{int}, @retVal \mapsto \mathsf{int}])] \ \circ_L ([b \mapsto r_b], [], [], [b \mapsto \mathsf{bool}])]]$$

(see Figure 12(d)). On line 4, a new block is started, therefore again a new empty variable properties tuple $\Diamond$ is appended:

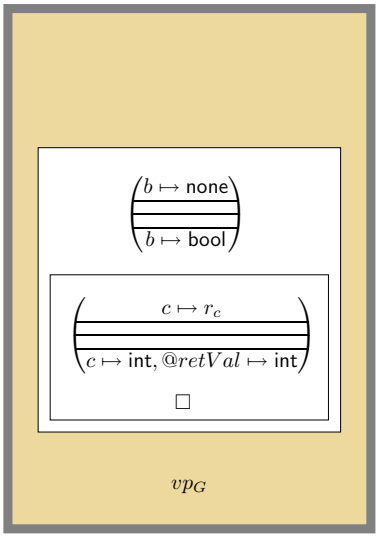
$$[vp_G \circ_G [[[\Box \circ_L ([c \mapsto r_c], [], [], [c \mapsto \mathsf{int}, @retVal \mapsto \mathsf{int}])] \ \circ_L ([b \mapsto r_b], [], [], [b \mapsto \mathsf{bool}])] \circ_L \Diamond]]$$



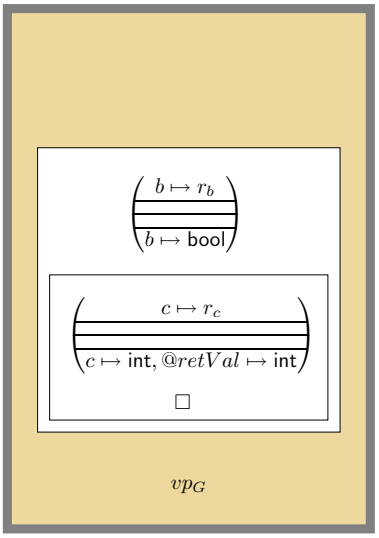
(a) Before execution of line 1



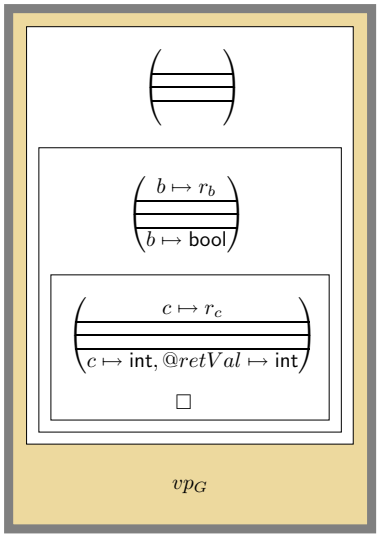
(b) After execution of line 1



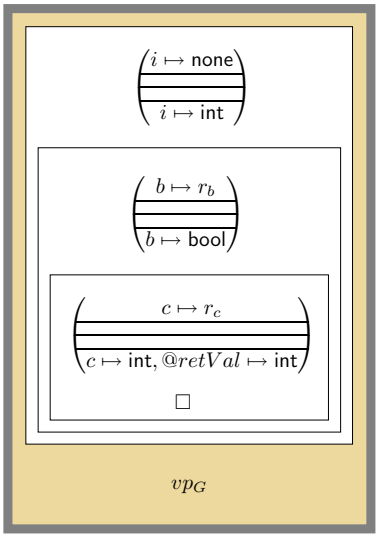
(c) After execution of line 2



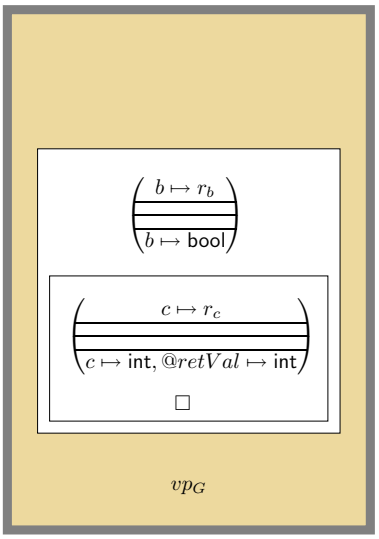
(d) After execution of line 3



(e) After execution of line 4



(f) After execution of line 5



(g) After execution of line 6

vp_G

(h) After execution of line 7

Figure 12: Variable properties stack construction when invoking the method a

(see Figure 12(e)). On line 5, a new integer variable i is declared what is reflected in the inner list head variable properties tuple:

$$[vp_G \circ_G [[[\Box \circ_L ([c \mapsto r_c], [], [], [c \mapsto \mathsf{int}, @retVal \mapsto \mathsf{int}])] \circ_L ([b \mapsto r_b], [], [], [b \mapsto \mathsf{bool}])] \circ_L ([i \mapsto \mathsf{none}], [], [], [i \mapsto \mathsf{int}])]]$$

(see Figure 12(f)). On line 6, the block ends, hence the appropriate variable properties tuple is discarded:

$$[vp_G \circ_G [[\Box \circ_L ([c \mapsto r_c], [], [], [c \mapsto \mathsf{int}, @retVal \mapsto \mathsf{int}])] \circ_L ([b \mapsto r_b], [], [], [b \mapsto \mathsf{bool}])]]$$

(see Figure 12(g)). Finally on line 7, the method execution ends, hence all local variable properties tuples are discarded and the original variable properties structure is restored:

$$vp_G$$

(see Figure 12(h)).

7.3 Configuration

In this subsection, we formally define abstract machine configuration which is later used for the definition of LanQ operational semantics.

A *configuration* of the abstract machine used for the definition of LanQ operational semantics is composed of two basic parts – *global* and *local*. The global part of the configuration stores information about resources – a quantum state of the whole system and a relation between channels and their ends. The local part of the configuration stores information about individual processes – the state of their local memory, variables, and terms to be evaluated.

A configuration C describing n processes running in parallel is written as:

$$C = [gs \,|\, ls_1 \parallel \cdots \parallel ls_n]$$

where gs is the global part of the configuration and ls_j represents the local process configuration of j-th process.

The components of the abstract machine configuration are formally defined as follows:

- Global part of the configuration: a pair (Q, C) where:
  - Q describes the quantum part of the configuration.
    In the present paper, we represent the quantum state of the system by a pair (ρ, L) of a finite density matrix ρ and a finite list L of natural numbers. The list L represents dimensions of individual quantum subsystems. The order of the list elements is given by order of quantum system allocations.
  - C represents the 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 and c_1 represent individual ends of one channel.
- Local part of the configuration: it defines state of one process, hence we call it a *local process configuration*. It is a triplet (lms, vp, ts) where:
  - *Local memory state* lms is a quadruple of partial functions, $lms = (lms_{Cl}, lms_Q, lms_{Ch}, lms_{ChE})$ which stores the state of classical memory and references to non-duplicable resources available to the process:
    - $lms_{Cl} : Ref_{Cl} \rightharpoonup Values$ is a partial function which returns a (classical) value stored at the given position in memory. The set of all such partial functions is denoted by LMS_{Cl},

* $lms_Q : Ref_Q \rightharpoonup Ref_{GQ}$ returns a global reference to quantum systems given by the local quantum reference. The set of all such partial functions is denoted by LMS_Q,
* $lms_{Ch} : Ref_{Ch} \rightharpoonup Ref_{GCh}$ returns a global reference to the channel given by the local channel reference. The set of all such partial functions is denoted by LMS_{Ch},
* $lms_{ChE} : Ref_{ChE} \rightharpoonup Ref_{GCh}$ returns a global reference to the channel corresponding to the given local channel end reference. The set of all such partial functions is denoted by LMS_{ChE}.

To simplify the notation, we regard lms itself as a partial function. Note that $Ref_{GQ} \subseteq Values$ and $Ref_{GCh} \subseteq Values$. Now we can define $lms = (lms_{Cl}, lms_Q, lms_{Ch}, lms_{ChE}) : Ref_L \rightharpoonup Values$ where:

$$lms(r) = \begin{cases} lms_{Cl}(r) & \text{if } r \in Ref_{Cl}, \\ lms_Q(r) & \text{if } r \in Ref_Q, \\ lms_{Ch}(r) & \text{if } r \in Ref_{Ch}, \\ lms_{ChE}(r) & \text{if } r \in Ref_{ChE}, \\ \perp & \text{if } r = \mathsf{none}. \end{cases}$$

The set of all such quadruples lms is denoted by LMS.

– *Variable properties vp* represent various properties of variables while respecting variable scope. They are stored as a list of lists of partial function tuples $f = (f_{var}, f_{ch}, f_{qa}, f_{type})$ where:
  * $f_{var} : Names \rightharpoonup Ref_L$ maps a variable name to a local reference,
  * $f_{ch} : Names \rightharpoonup Names \times Names$ maps a channel variable name to variable names representing ends of the channel,
  * $f_{qa} : Names \rightharpoonup Names_{[]}$ maps a variable name representing a quantum system to variable names that represent its subsystems,
  * $f_{type} : Names \rightharpoonup Types$ maps a variable name to the variable type.

  We call the quadruple f a *variable properties tuple*.

  We define $VarProp_L$ to be a set of all finite lists of such partial function tuples f. These lists are built to reflect the block structure of a method as described in Subsection 7.2.

  We define $VarProp$ to be a set of all finite lists of lists of such partial function tuples f. These lists are built to reflect the method calls as described in Subsection 7.2.

  We define an *empty variable properties tuple* as a partial function tuple $\Diamond$:

$$\Diamond = (f_{var}, f_{ch}, f_{qa}, f_{type})$$

  where $\mathrm{dom}(f_{var}) = \mathrm{dom}(f_{ch}) = \mathrm{dom}(f_{qa}) = \mathrm{dom}(f_{type}) = \emptyset$.

– *Term stack ts*: stack of terms to be evaluated. For the sake of readability, we use a notation where individual stack items are underlined. An empty term stack is denoted by ε.

A configuration can evolve by a probabilistic transition to a mixture of configurations, so called *mixed configuration*. A mixed configuration is written as:

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

It represents configurations of q different computational branches, each of them running with probability p_i. A configuration is a special case of a mixed configuration where $q = 1$ and $p_1 = 1$.

7.4 Variable properties handling functions

In this subsection, we define functions for variable properties handling.

First we define functions for retrieving information about variables using variable properties. The defined functions are designed so that they only work with variables accessible from the actually running method. We achieve this by inspecting the structure of the variable properties. If the variable properties given as one of the function arguments are represented by a list of lists of partial function tuples $L = [L_1 \circ_G K]$ then we consider only its second element – the list of partial function tuples K. We do not consider the variable properties from L_1 as they are inaccessible to the current method (as explained in Subsection 7.2).

We then walk through the obtained list of partial function tuples $[K \circ_L f]$. We attempt to get the requested information about requested variable using appropriate partial function from the variable properties tuple. If the requested information about the variable cannot be obtained from the actual tuple, we repeat this procedure with the list of partial function tuples K. This procedure is designed so that it respects block scope of variables (as in more detail explained in Subsection 7.2).

Definition 7.3. *We define a partial function $varRef : Names \times VarProp \rightarrow Ref_L$ for getting a local reference from a variable name and variable properties as:*

$$varRef(x, [L_1 \circ_G K]) \stackrel{\text{def}}{=} varRef_L(x, K)$$

where $varRef_L : Names \times VarProp_L \rightharpoonup Ref_L$ is a partial function for getting a local reference from a variable name and a list of partial function tuples:

$$varRef_L(x, [K \circ_L (f_{var}, f_{ch}, f_{qa}, f_{type})]) \stackrel{\text{def}}{=} \begin{cases} f_{var}(x) & \text{if } f_{var}(x) \text{ is defined} \\ varRef_L(x, K) & \text{otherwise} \end{cases}$$

Definition 7.4. *We define a partial function $chanEnds : Names \times VarProp \rightharpoonup Names \times Names$ for getting variable names that represent individual ends of given channel from a name of the channel and variable properties:*

$$chanEnds(x, [L_1 \circ_G K]) \stackrel{\text{def}}{=} chanEnds_L(x, K)$$

where $chanEnds_L : Names \times VarProp_L \rightharpoonup Names \times Names$ is a partial function for getting variable names that represent individual ends of given channel from a name of the channel and a list of partial function tuples:

$$chanEnds_L(x, [K \circ_L (f_{var}, f_{ch}, f_{qa}, f_{type})]) \stackrel{\text{def}}{=} \begin{cases} f_{ch}(x) & \text{if } f_{ch}(x) \text{ is defined} \\ chanEnds_L(x, K) & \text{otherwise} \end{cases}$$

Definition 7.5. *We define a partial function $aliasSubsyst : Names \times VarProp \rightharpoonup Names_{[]}$ for getting a list of variable names that represent individual parts of a compound system from a name of the compound system and variable properties:*

$$aliasSubsyst(x, [L_1 \circ_G K]) \stackrel{\text{def}}{=} aliasSubsyst_L(x, K)$$

where $aliasSubsyst_L : Names \times VarProp_L \rightharpoonup Names_{[]}$ is a partial 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:

$$aliasSubsyst_L(x, [K \circ_L (f_{var}, f_{ch}, f_{qa}, f_{type})]) \stackrel{\text{def}}{=} \begin{cases} f_{qa}(x) & \text{if } f_{qa}(x) \text{ is defined} \\ aliasSubsyst_L(x, K) & \text{otherwise} \end{cases}$$

Definition 7.6. *We define a partial function* $typeOf : Names \times VarProp \rightharpoonup Types$ *for getting a variable type from a name of a variable and variable properties:*

$$typeOf(x, [L_1 \circ_G K]) \stackrel{\text{def}}{=} typeOf_L(x, K)$$

where $typeOf_L : Names \times VarProp_L \rightharpoonup Types$ *is a partial function for getting a type from a name of the variable and a list of partial function tuples:*

$$typeOf_L(x, [K \circ_L (f_{var}, f_{ch}, f_{qa}, f_{type})]) \stackrel{\text{def}}{=} \begin{cases} f_{type}(x) & \text{if } f_{type}(x) \text{ is defined} \\ typeOf_L(x, K) & \text{otherwise} \end{cases}$$

7.5 Local memory handling functions

Next we define functions for local memory state handling.

LanQ allows the programmer to create multiple processes. These processes can communicate with each other, namely a process can send some resource, *ie.* a quantum system or a channel, to another process. In that case, the language must assure that the sent resource becomes unavailable to the sending process.

For this reason we define a function $unmap_{nd}$[7] which invalidates a reference to resource in given local memory state. By invalidation we mean setting the appropriate local reference to the sent resource to point to $\perp$ in the local memory state.

This function can be split into three functions: unmapping a memory reference to a quantum system (this is done by the function $unmap_Q$[8]), unmapping a memory reference to a channel ($unmap_{Ch}$[9]), and unmapping a memory reference to a channel end ($unmap_{ChE}$[10]).

The function $unmap_Q$ is designed to obey the following rule: When unmapping a reference to a quantum system ρ then any memory reference which refers to any part of ρ is unmapped too.

The function $unmap_{Ch}$ is designed to obey the following rule: When unmapping a reference to a channel c then any memory reference to its ends is unmapped too. The reason is that when a process sends away a channel, it also loses control over both its ends.

The function $unmap_{ChE}$ is designed to obey the following rule: When unmapping a reference to a channel end c then we unmap any memory reference to the corresponding channel. The justification is that when a process sends away a part of a channel, it also loses control over the whole channel.

Definition 7.7. *We define a function* $unmap_{nd} : Ref_L \times LMS \to LMS$ *for unmapping a reference to a non-duplicable value from the local memory:*

$$unmap_{nd}((refType, n), lms) \stackrel{\text{def}}{=} \begin{cases} unmap_Q(n, lms) & \text{if } refType = Quantum, \\ unmap_{Ch}(n, lms) & \text{if } refType = Channel, \\ unmap_{ChE}(i, n, lms) & \text{if } refType = ChannelEnd_i, \\ lms & \text{otherwise.} \end{cases}$$

where

- *Function* $unmap_Q : (\mathbb{N}_\perp)_{[\circlearrowright]} \times LMS \to LMS$ *is defined as:*

$$unmap_Q(n, (lms_{Cl}, lms_Q, lms_{Ch}, lms_{ChE})) = (lms_{Cl}, lms'_Q, lms_{Ch}, lms_{ChE}).$$

[7] The name $unmap_{nd}$ should be read as "unmap (a reference to a) *n*on*d*uplicable (value)".
[8] The name $unmap_Q$ should be read as "unmap (a reference to a) *q*uantum (value)".
[9] The name $unmap_{Ch}$ should be read as "unmap (a reference to a) *ch*annel (value)".
[10] The name $unmap_{ChE}$ should be read as "unmap (a reference to a) *ch*annel *e*nd (value)".

where lms'_Q is defined as:

$$lms'_Q((Quantum, l)) \stackrel{\text{def}}{=} \begin{cases} lms_Q((Quantum, l)) & \text{if } \mathrm{set}_{[\circlearrowright]}(n) \cap \mathrm{set}_{[\circlearrowright]}(l) \subseteq \{\bot\}, \\ \bot & \text{otherwise,} \end{cases}$$

- *Function $unmap_{Ch} : \mathbb{N} \times LMS \to LMS$ is defined as:*

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

$$\begin{aligned} \text{where} \quad & lms'_{Ch} \stackrel{\text{def}}{=} lms_{Ch}[(Channel, n) \mapsto \bot], \\ & lms'_{ChE} \stackrel{\text{def}}{=} lms_{ChE}[(ChannelEnd_0, n) \mapsto \bot, (ChannelEnd_1, n) \mapsto \bot], \end{aligned}$$

- *Function $unmap_{ChE} : \{0, 1\} \times \mathbb{N} \times LMS \to LMS$ is defined as:*

$$unmap_{ChE}(i, n, (lms_{Cl}, lms_Q, lms_{Ch}, lms_{ChE})) = (lms_{Cl}, lms_Q, lms'_{Ch}, lms'_{ChE}).$$

$$\begin{aligned} \text{where} \quad & lms'_{Ch} \stackrel{\text{def}}{=} lms_{Ch}[(Channel, n) \mapsto \bot], \\ & lms'_{ChE} \stackrel{\text{def}}{=} lms_{ChE}[(ChannelEnd_i, n) \mapsto \bot]. \end{aligned}$$

We extend the function $unmap_{nd}$ so that we can also use any set of references as the first argument.

Definition 7.8. *For any $R = \{r_1, \ldots, r_k\} \subseteq Ref_L$ we define:*

$$unmap_{nd}(R, lms) \stackrel{\text{def}}{=} unmap_{nd}(r_1, unmap_{nd}(r_2, \ldots unmap_{nd}(r_k, lms) \ldots)).$$

Next, we define a function for updating updating a local memory state $lms = (lms_{Cl}, lms_Q, lms_{Ch}, lms_{ChE})$. We use the existing concept of partial function update and extend this concept to local memory states. The extended function updates appropriate element of the quadruple according to memory reference type:

Definition 7.9. *Let $lms = (lms_{Cl}, lms_Q, lms_{Ch}, lms_{ChE})$ be a local memory state. For $r \in Ref_L$ and $v \in Values$, we define* local memory state update *$lms[r \mapsto v]_+$ as:*

$$lms[r \mapsto v]_+ \stackrel{\text{def}}{=} \begin{cases} (lms_{Cl}[r \mapsto v]_+, lms_Q, lms_{Ch}, lms_{ChE}) & \text{if } r \in Ref_{Cl}, \\ (lms_{Cl}, lms_Q[r \mapsto v]_+, lms_{Ch}, lms_{ChE}) & \text{if } r \in Ref_Q, \\ (lms_{Cl}, lms_Q, lms_{Ch}[r \mapsto v]_+, lms_{ChE}) & \text{if } r \in Ref_{Ch}, \\ (lms_{Cl}, lms_Q, lms_{Ch}, lms_{ChE}[r \mapsto v]_+) & \text{if } r \in Ref_{ChE}. \end{cases}$$

7.6 Functions for handling aliasfor constructs

Quantum algorithms often use multipartite quantum systems. LanQ allows definition of compound quantum systems using **aliasfor** construct: if $q_0, \ldots, q_n$ are quantum variables then the declaration:

$$q \textbf{ aliasfor } [q_0, \ldots, q_n];$$

declares a new quantum variable q which specifies a quantum system composed from subsystems referred by the variables $q_0, \ldots, q_n$. In this subsection, we define functions needed to handle this construct.

To simplify the description of the functions in the following text, we first define two concepts: We call a quantum variable which was declared using the **aliasfor** construct an *aliased quantum variable*. The quantum variables not declared using the **aliasfor** construct are called *proper quantum variables*.

In the following text, we require that all quantum variables that any aliased quantum variable is composed of are *proper quantum variables*. This requirement is later imposed by the semantic rule OP-VarDeclAlF.

Handling the **aliasfor** construct is a little complicated. Two cases must be handled when assigning a reference to a quantum system to a quantum variable q:

(1) The variable q is a proper quantum variable. Hence it can be used in several aliased quantum variables. Then this variable (1a) and all the aliased quantum variables which use this variable as their subsystem (1b) must be updated,

(2) The variable q is an aliased quantum variable. Then all its subsystems must be appropriately modified. However, the subsystems can be also used in several other aliased quantum variables as subsystems. Then all these aliased quantum variables must be updated too.

We define several auxiliary functions which help us with handling the assignment to a quantum variable. These functions modify variable properties and a local memory state parts of the local process configuration. Therefore all these auxiliary functions take the unmodified variable properties vp and local memory state lms parts as arguments and yield the modified ones. Moreover, all these functions take the name $name$ of the quantum variable being assigned and the assigned reference ref as its additional arguments.

The simplest case is the case (1a). For this case, we define a function $f_{assignQSystemDirect}$ which performs the following operation:

- If the assigned reference $ref = (Quantum, q)$ is invalid (this is checked by the condition $\text{linearize}_\perp(q) = \perp$), the reference ref is unmapped from the local process memory.

  Otherwise the reference ref is set to map from the local process memory to a global reference to the corresponding quantum systems in the system memory,

- The assigned quantum variable is mapped to the reference ref.

Definition 7.10. *We define a function* $f_{assignQSystemDirect} : LMS_Q \times VarProp \times Names \times Ref_Q \to LMS_Q \times VarProp$ *as:*

$$f_{assignQSystemDirect}(lms_Q, vp, name, ref) \stackrel{\text{def}}{=} (lms_{Q,ret}, vp_{ret})$$

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

To update all the aliased quantum variables which use the assigned proper quantum variable as their subsystems (the case (1b)), we first define an auxiliary function $f_{assignQSystemInAlias}$ which updates one subsystem reference in the aliased quantum variable. This function takes one more argument – the index $index$ of the updated subsystem. It then proceeds as follows:

- It takes the original reference of the aliased quantum variable $varRef(name)$ and modifies its $index$-th element to point to the newly assigned system (specified by the recursive $\mathbb{N}$-list q). Individual elements of the recursive $\mathbb{N}$-list specifying the new reference are denoted by v'_j,

- It unmaps the original reference from the local process memory,

- If any of the systems in the newly assigned reference is invalid (checked by the condition $\exists j : lms_Q((Quantum, v'_j)) = \bot$), the newly assigned reference is unmapped from the local process memory.

  Otherwise the reference ref is set to map from the local process memory to a global reference to the corresponding quantum systems in the system memory,

- The assigned quantum variable is mapped to the new reference $(Quantum, [v'_1, \ldots, v'_k])$.

Definition 7.11. *We define a function* $f_{assignQSystemInAlias} : LMS_Q \times VarProp \times Names \times Ref_Q \times \mathbb{N} \to LMS_Q \times VarProp$ *as:*

$$f_{assignQSystemInAlias}(lms_Q, vp, name, ref, index) \stackrel{\text{def}}{=} (lms_{Q,ret}, vp_{ret})$$

$$\begin{aligned}
\textit{where}\quad lms_{Q,ret} &= lms_Q[(Quantum, [v_1, \ldots, v_k]) \mapsto \bot][(Quantum, [v'_1, \ldots, v'_k]) \mapsto gq] \\
vp_{ret} &= vp[name \mapsto (Quantum, [v'_1, \ldots, v'_k])]_{var} \\
v'_i &= \begin{cases} q & \textit{if } i = index \\ v_i & \textit{otherwise} \end{cases} \\
\textit{given that}\quad ref &= (Quantum, q) \\
varRef(name, vp) &= (Quantum, [v_1, \ldots, v_k]) \\
gq &= \begin{cases} \bot & \textit{if } \exists j : lms_Q((Quantum, v'_j)) = \bot \\ (GQuantum, \text{linearize}_\bot([v'_1, \ldots, v'_k])) & \textit{otherwise} \end{cases}
\end{aligned}$$

Before we define a function that handles the whole case (1), we must define one more auxiliary function $localAliasedVars$ that returns all the aliased quantum variables available to the currently running method.

Definition 7.12. *We define a function* $localAliasedVars : VarProp \to \mathcal{P}(Names)$ *as:*

$$\begin{aligned}
localAliasedVars([L_1 \circ_G K]) &\stackrel{\text{def}}{=} localAliasedVars_L(K) \\
localAliasedVars(\blacksquare) &\stackrel{\text{def}}{=} \emptyset
\end{aligned}$$

where $localAliasedVars_L : VarProp_L \to \mathcal{P}(Names)$ *is a function for getting a set of all names of variables representing compound systems in the local variable properties list from a list of partial function tuples:*

$$\begin{aligned}
localAliasedVars_L([K \circ_L (f_{var}, f_{ch}, f_{qa}, f_{type})]) &\stackrel{\text{def}}{=} dom(f_{qa}) \cup localAliasedVars_L(K) \\
localAliasedVars_L(\square) &\stackrel{\text{def}}{=} \emptyset
\end{aligned}$$

Now we are ready to define a function $f_{assignQSystem}$ that performs assignment to one *proper quantum variable* while correctly adjusting all aliased quantum variables which use the variable being assigned (case (1)). The function operates as follows:

- It first uses the function $f_{assignQSystemDirect}$ to perform the assignment to the *proper quantum variable*,

- Then it adjusts all the aliased quantum variables which use the variable being assigned (the set of such variables is denoted as AQV) using the function $f_{assignQSystemInAlias}$.

Definition 7.13. *We define a function* $f_{assignQSystem} : LMS_Q \times VarProp \times Names \times Ref_Q \to LMS_Q \times VarProp$ *as:*

$$f_{assignQSystem}(lms_Q, vp, name, ref) \stackrel{\text{def}}{=} (lms_{Q,ret}, vp_{ret})$$

$$\begin{aligned}
\textit{where}\quad & (lms_{Q,0}, vp_0) = f_{assignQSystemDirect}(lms_Q, vp, name, ref),\\
& (lms_{Q,i}, vp_i) = f_{assignQSystemInAlias}(lms_{Q,i-1}, vp_{i-1}, qcs_i, ref, l_i) \forall i : 1 \leq i \leq k,\\
& lms_{Q,ret} = lms_{Q,k},\ vp_{ret} = vp_k,\\
\textit{given that}\quad & AQV = \{aqv \in localAliasedVars(vp) |\ name \in \mathrm{set}(aliasSubsyst(aqv, vp)\},\\
& AQV \textit{ is indexed by numbers } i \in \mathbb{N} : 1 \leq i \leq k,\\
& aqv_i \in AQV,\\
& aliasSubsyst(aqv_i, vp) = [aqv_{i,1}, \ldots, aqv_{i,m_i}],\\
& aqv_{i,l_i} = name.
\end{aligned}$$

Last, we define a function $f_{assignQAlias}$ that performs an assignment to an aliased quantum variable. This function operates very straightforwardly – it takes each proper quantum variables which the aliased quantum variable is composed of and applies the function $f_{assignQSystem}$ onto it. We assume that the number of subsystems of the aliased quantum variable corresponds to the structure of the assigned reference (the length of the $\mathbb{N}$-list in the reference).

Definition 7.14. *We define a function* $f_{assignQAlias} : LMS_Q \times VarProp \times Names \times Ref_Q \rightarrow LMS_Q \times VarProp$ *as:*

$$f_{assignQAlias}(lms_Q, vp, name, ref) \stackrel{\text{def}}{=} (lms_{Q,ret}, vp_{ret})$$

$$\begin{aligned}
\textit{where}\quad & lms_{Q,0} = lms_Q,\ vp_0 = vp,\\
& (lms_{Q,i}, vp_i) = f_{assignQSystem}(lms_{Q,i-1}, vp_{i-1}, q_i, (Quantum, v_i)) \textit{ for all } 1 \leq i \leq k,\\
& lms_{Q,ret} = lms_{Q,k},\ vp_{ret} = vp_k,\\
\textit{given that}\quad & aliasSubsyst(name, vp) = [q_1, \ldots, q_k],\\
& ref = (Quantum, [v_1, \ldots, v_k]).
\end{aligned}$$

7.7 Internal values

Expressions evaluate to references and values which in turn are possibly references to global memory. Operational semantics uses both values and references so we define *internal value* to be a triplet $(ref, val, \mathsf{T}) \in Ref_L \times Values \times Types$ written as $\langle\langle ref, val \rangle\rangle_{\mathsf{T}}$.

7.8 Transitions

We define operational semantics in terms of the following relations:

- $\longrightarrow_v$ – Transitions of expressions to internal values,
- $\longrightarrow_e$ – Transitions of expressions to expressions – the order of evaluation is encoded here,
- $\longrightarrow_s$ – Transitions of statements,
- $\longrightarrow_{ret}$ – Transitions of **return** statements,
- $\longrightarrow_{rte}$ – Transitions of runtime errors,
- $\longrightarrow_r$ – Transitions of statements to statements, used to rewrite an abbreviated statement to the unabbreviated form,
- $\longrightarrow_p$ – Transitions of processes,
- $\stackrel{p}{\longrightarrow}$ – This defines probabilistic transitions of processes, $0 \leq p \leq 1$ is the probability of the transition.

We define relation $\longrightarrow$ as:

$$\longrightarrow = \longrightarrow_v \cup \longrightarrow_e \cup \longrightarrow_s \cup \longrightarrow_{ret} \cup \longrightarrow_{rte} \cup \longrightarrow_r \cup \longrightarrow_p .$$

The relations $\longrightarrow_v, \longrightarrow_e, \longrightarrow_s, \longrightarrow_{ret}, \longrightarrow_{rte}, \longrightarrow_r$ and $\xrightarrow{p}$ define deterministic and probabilistic single process evolution. Nondeterminism is introduced by parallel evolution of processes – a choice which process gets evolved is nondeterministic. However, there is no nondeterminism in the evolution of individual processes even when they are run in parallel with other processes. This is an improvement over existing quantum process algebras [LJ04, GN04].

In these algebras, there is a nondeterminism arising from resource sharing. Although there is no nondeterminism arising from quantum resource control, there is one arising from channel resources: it is possible for three or more processes to share one channel. When these processes use the channel simultaneously, the resulting behaviour is nondeterministic.[11] We avoid this type of nondeterminism by using channel ends for communication instead of the channels themselves 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 is 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, the errors arising from invalid typing can be avoided. The formal proof that a language does not suffer from typing errors lies in proving standard lemmata in the style of Wright and Felleisen [WF94]. For LanQ, this is done in Section 8.

However, there exist some unavoidable cases caused by the by-reference handling of variables. For example, a process P can send away a qubit referred by variable ψ. If P later tries to measure a qubit referred by ψ, there is none. Such cases are handled by runtime errors described in the next subsection.

7.9 Runtime errors

There exist errors that cannot be recognized during compile time and can occur during the run time. For that reason. we define special stack symbols representing such *runtime errors*:

- **UV**: an error representing an *u*ninitialized *v*ariable usage. An example method invoking this error is in Figure 13(a) (U is a unitary operation there). In this example, the variable q is sent away, hence not initialized. The attempt to perform $U(q)$ therefore invokes a runtime error **UV**.

- **OQV**: an error representing *o*verlapping *q*uantum *v*ariable usage. An example method invoking this error is shown in Figure 13(b) (U is a two-qubit unitary operation there). In this example, variables p and q refer to the same quantum system. An attempt to perform $U(p, q)$ therefore invokes a runtime error **OQV**.

- **ISQV**: an error representing an assignment to an *i*ncompatibly *s*tructured *q*uantum *v*ariable. An example method invoking this error is shown in Figure 13(c). In this example, variables p and q refer to qubit systems, r refers to a system composed of the two qubits. An attempt to assign one four-dimensional quantum system to r fails

[11] The nondeterministic behaviour can be used to simply catch *eg.* server environment serving requests from multiple clients where it is used to resolve which request came from which client.

as this assignment must also appropriately set the two systems p and q. Hence this assignment invokes a runtime error **ISQV**.

```
void ex1(channelEnd[qbit] c) {
    qbit q;
    q = new qbit();
    send(c, q);
    U(q);
}
```

(a)

```
void ex2() {
    qbit p, q;
    q = new qbit();
    p = q;
    U(p, q);
}
```

(b)

```
void ex3() {
    qbit p, q;
    r aliasfor [p, q];
    r = new Q4();
}
```

(c)

Figure 13: Example methods invoking runtime errors.

7.10 Processes and configurations

In this subsection, we define special configurations, processes and relations between them.

We define a special configuration **start** (a starting configuration for any LanQ program execution) and a set $\mathbf{0}_c$ of silent local process configurations as:

$$\mathbf{start} \stackrel{\text{def}}{=} [(((1), []), []) \mid (([], [], [], []), \blacksquare, \underline{main()})]$$

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

The *terminal configuration* is defined as

$$[gs \mid (lms_1, vp_1, v_1) \parallel \cdots \parallel (lms_n, vp_n, v_n)]$$

where gs is a global part of the configuration, and for all i, $lms_i \in LMS$, $vp_i \in VarProp$, and v_i is either ε, runtime error $RTErr$, or an internal value $\mathbf{v}$. If some of v_i is a runtime error, then we call this configuration *errorneous*.

7.10.1 Structural congruence

In this subsection, we define structurally congruent processes. A process is fully characterized by a local process configuration, therefore the relation is defined on these local process configurations.

Any process is structurally congruent to a process running in parallel with a silent process (rule SC-Nil). Order of processes in the configuration does not matter (SC-Comm) as well as grouping of processes within the configuration (SC-Assoc).

SC-Nil	$P \parallel 0 \equiv P$ for any $0 \in \mathbf{0_c}$		
SC-Comm	$P \parallel Q \equiv Q \parallel P$	SC-Assoc	$(P \parallel Q) \parallel R \equiv P \parallel (Q \parallel R)$

7.10.2 Nondeterminism and parallelism

In this subsection, we define behaviour related to nondeterminism and parallelism.

The rule NP-PropagProb states that evolution of a process P leaves all other processes running in parallel with P in their original state while propagating the probability distribution on configurations to the top level. We can exchange congruent processes without any impact on the resulting behaviour (rule NP-Cong). A probabilistic configuration consisting of two or more probabilistic alternatives must resolve a probabilistic choice (rule NP-ProbEvol).

NP-PropagProb	$\dfrac{[gs \mid P] \longrightarrow \boxplus_i\ p_i \bullet [gs_i \mid P_i]}{[gs \mid P \parallel Q] \longrightarrow \boxplus_i\ p_i \bullet [gs_i \mid P_i \parallel Q]}$
NP-Cong	$\dfrac{[gs \mid P] \longrightarrow \boxplus_i\ p_i \bullet [gs_i \mid P_i] \quad P \equiv P' \quad P_i \equiv P'_i \text{ for all } i}{[gs \mid P'] \longrightarrow \boxplus_i\ p_i \bullet [gs_i \mid P'_i]}$
NP-ProbEvol	$\boxplus_{i=1}^{q}\ p_i \bullet [gs_i \mid P_i] \xrightarrow{p_i} [gs_i \mid P_i] \qquad \text{for } q > 1$

7.11 Evaluation

In this subsection, we define the transition rules of individual processes.

7.11.1 Basic rules

The first four rules define configuration change on a skip statement (rule OP-Skip), a 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-Var	$[gs \mid (lms, vp, \underline{x}\ ts)]$	$\longrightarrow_v$ $[gs \mid (lms, vp, \underline{\langle\!\langle ref, lms(ref)\rangle\!\rangle_{typeOf(x,vp)}}\ ts)]$ where $ref = varRef(x, vp)$
OP-Bracket	$[gs \mid (lms, vp, \underline{(E)}\ ts)]$	$\longrightarrow_r\ [gs \mid (lms, vp, \underline{E}\ ts)]$
OP-BlockHead	$[gs \mid (lms, vp, \underline{\overline{Be}}\ ts)]$	$\longrightarrow_r\ [gs \mid (lms, vp, \underline{head(\overline{Be})}\ \underline{tail(\overline{Be})}\ ts)]$
OP-SubstE	$[gs \mid (lms, vp, \underline{\mathbf{v}}\ \underline{Ec}\ ts)]$	$\longrightarrow_e\ [gs \mid (lms, vp, \underline{Ec[\mathbf{v}]}\ ts)]$
OP-SubstS	$[gs \mid (lms, vp, \underline{\mathbf{v}}\ \underline{Sc}\ ts)]$	$\longrightarrow_e\ [gs \mid (lms, vp, \underline{Sc[\mathbf{v}]}\ ts)]$

7.11.2 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).

OP-PromoExpr	$[gs \mid (lms, vp, \underline{PE\mathbf{;}}\ ts)]$	$\longrightarrow_e\ [gs \mid (lms, vp, \underline{PE\ \bullet\mathbf{;}}\ ts)]$
OP-PromoForget	$[gs \mid (lms, vp, \underline{\mathbf{v;}}\ ts)]$	$\longrightarrow_s\ [gs \mid (lms, vp, ts)]$

7.11.3 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 $\mathsf{Q_d}$, *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.

OP-AllocQ

$$
\begin{aligned}
&[gs \,|\, (lms, vp, \underline{\textbf{new}\ \mathsf{Q_d}()}\ ts)] \quad \longrightarrow_v \\
&\quad [gs' \,|\, (lms', vp, \underline{\langle\!\langle (Quantum, [l]), (GQuantum, [l]) \rangle\!\rangle_{\mathsf{Q_d}}}\ ts)] \\
&\quad \text{where } l = |L| \\
&\qquad gs' = ((\rho \otimes (\tfrac{1}{d} I_d), L \cdot [d]), C) \\
&\qquad lms' = (lms_{Cl}, lms'_Q, lms_{Ch}, lms_{ChE}) \\
&\qquad lms'_Q = lms_Q[(Quantum, [l]) \mapsto (GQuantum, [l])] \\
&\quad \text{given that } gs = ((\rho, L), C) \\
&\qquad\qquad lms = (lms_{Cl}, lms_Q, lms_{Ch}, lms_{ChE})
\end{aligned}
$$

OP-AllocC

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

7.11.4 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** $[q_0, \ldots, q_n]$ is used where $q_0, \ldots, q_n$ are names of quantum variables. Some of them can again be compound systems. To deal with this feature, all variables from $\{q_0, \ldots, q_n\}$ 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

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

OP-VarDecl

$$[gs \mid (lms, [vp_G \circ_G [vp \circ_L vp_L]], \underline{T\ x;}\ ts)] \longrightarrow_s [gs \mid (lms, [vp_G \circ_G [vp \circ_L vp'_L]], ts)]$$
$$\text{where } vp'_L = vp_L[x \mapsto \mathsf{none}]_{+,var}[x \mapsto T]_{+,type}$$

OP-VarDeclChE

$$[gs \mid (lms, [vp_G \circ_G [vp \circ_L vp_L]], \underline{\mathsf{channel[T]}\ c\ \textbf{withends}[c_0,c_1];}\ ts)] \longrightarrow_s [gs \mid (lms, [vp_G \circ_G [vp \circ_L vp'_L]], ts)]$$
$$\text{where } vp'_L = vp_L[c \mapsto \mathsf{none}]_{+,var}[c_0 \mapsto \mathsf{none}]_{+,var}[c_1 \mapsto \mathsf{none}]_{+,var}[c \mapsto (c_0, c_1)]_{+,ch}$$
$$[c \mapsto \mathsf{channel[T]}]_{+,type}[c_0 \mapsto \mathsf{channelEnd[T]}]_{+,type}[c_1 \mapsto \mathsf{channelEnd[T]}]_{+,type}$$

OP-VarDeclAlF

$$[gs \mid (lms, [vp_G \circ_G [vp \circ_L vp_L]], \underline{q\ \textbf{aliasfor}\ [q_0, \ldots, q_n];}\ ts)] \longrightarrow_s [gs \mid (lms, [vp_G \circ_G [vp \circ_L vp'_L]], ts)]$$
$$\text{where } vp'_L = vp_L[q \mapsto (Quantum, [l_0, \ldots, l_n])]_{+,var}[q \mapsto [q'_0, \ldots, q'_n]]_{+,qa}[q \mapsto \bigotimes_{i=0}^{n} typeOf(q_n, vp)]_{+,type}$$
$$\text{given that } l_i = \begin{cases} l & \text{if } varRef_L(q_i, [vp \circ_L vp_L]) = (Quantum, l) \\ \bot & \text{otherwise} \end{cases}$$
$$q'_i = \begin{cases} p_0, \ldots, p_k & \text{if } aliasSubsyst_L(q_i, [vp \circ_L vp'_L]) = [p_0, \ldots, p_k] \\ q_i & \text{otherwise} \end{cases}$$
$$varRef_L(q_i, [vp \circ_L vp'_L]) \text{ is defined for } 0 \leq i \leq n$$

7.11.5 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). If the structure of assigned system is not compatible with the structure of the assigned variable then a runtime error **ISQV** occurs (rule OP-AssignQAValueBad).

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

OP-AssignNewValue $[gs \,|\, (lms, vp, \underline{x = \mathbf{v}}\ ts)] \longrightarrow_v [gs \,|\, (lms', vp', \underline{\langle\!\langle lr', lv \rangle\!\rangle_{\mathsf{T}}}\ ts)]$

where $lr' = (Classical, nc)$
$lms'_{Cl} = lms_{Cl}[lr' \mapsto lv]_+$
$vp' = vp[x \mapsto lr']_{var}$

given that $\mathbf{v} = \langle\!\langle lr, lv \rangle\!\rangle_{\mathsf{T}}$
$nc \in \mathbb{N}_0$ is such that $lms((Classical, nc))$ is not defined
$lms' = lms[lr' \mapsto lv]_+$
$lr = \mathsf{none} \wedge lv \neq \bot$

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

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

given that $\mathbf{v} = \langle\!\langle lr, lv \rangle\!\rangle_{\mathsf{T}}$
$lr = (Quantum, q)$ 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 $[gs \,|\, (lms, vp, \underline{x = \mathbf{v}}\ ts)] \longrightarrow_v [gs \,|\, (lms', vp', \underline{\mathbf{v}}\ ts)]$

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

given that $\mathbf{v} = \langle\!\langle lr, lv \rangle\!\rangle_{\mathsf{T}}$
$lr = (Quantum, q)$ and $aliasSubsyst(x, vp)$ is defined
$lms = (lms_{Cl}, lms_Q, lms_{Ch}, lms_{ChE})$
$lms' = (lms_{Cl}, lms'_Q, lms_{Ch}, lms_{ChE})$
$typeOf(q) = \bigotimes_{i=0}^{n} Q_i, \mathsf{T} = \bigotimes_{j=0}^{m} T_j$
and $m = n$ and $\forall i : Q_i \cong T_i$

OP-AssignQAValueBad $[gs \,|\, (lms, vp, \underline{x = \mathbf{v}}\ ts)] \longrightarrow_{rte} [gs \,|\, (lms, vp, \mathbf{ISQV})]$

given that $typeOf(q) = \bigotimes_{i=0}^{n} Q_i, \mathsf{T} = \bigotimes_{j=0}^{m} T_j$
and $m \neq n$ or $\exists i : Q_i \ncong T_i$

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

$$\text{where } vp' = \begin{cases} vp[x \mapsto lr]_{var}[x_0 \mapsto (ChannelEnd_0, i)]_{var}[x_1 \mapsto (ChannelEnd_1, i)]_{var} \\ \qquad\qquad \text{if } lr = (Channel, i) \text{ and } chanEnds(x, vp) = (x_0, x_1) \\ vp[x \mapsto lr]_{var} \qquad \text{otherwise} \end{cases}$$

given that $\mathbf{v} = \langle\!\langle lr, lv \rangle\!\rangle_{\mathsf{T}}$
$(lr = \mathsf{none} \wedge lv = \bot) \vee (lr \neq (Quantum, q))$

7.11.6 Block

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

OP-BLOCK $[gs \,|\, (lms, [vp_G \circ_G vp], \underline{\{B\}}\ ts)] \longrightarrow_s$
$[gs|(lms, [vp_G \circ_G [vp \circ_L \square]], \underline{B}\ \underline{\circ_{\mathbf{L}}}\ ts)]$

OP-BLOCKEND $[gs \,|\, (lms, [vp_G \circ_G [vp \circ_L vp_L]], \underline{\circ_{\mathbf{L}}}\ ts)] \longrightarrow_s$
$[gs|(lms, [vp_G \circ_G vp], ts)]$

7.11.7 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).

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

OP-IFTRUE $[gs \,|\, (lms, vp, \underline{\textbf{if } (\mathbf{v})\ S_1 \textbf{ else } S_2}\ ts)] \longrightarrow_s [gs \,|\, (lms, vp, \underline{S_1}\ ts)]$
if $\mathbf{v} = \langle\!\langle r, \textbf{true} \rangle\!\rangle_{\mathsf{bool}}$

OP-IFFALSE $[gs \,|\, (lms, vp, \underline{\textbf{if } (\mathbf{v})\ S_1 \textbf{ else } S_2}\ ts)] \longrightarrow_s [gs \,|\, (lms, vp, \underline{S_2}\ ts)]$
if $\mathbf{v} = \langle\!\langle r, \textbf{false} \rangle\!\rangle_{\mathsf{bool}}$

7.11.8 Conditional cycle – while

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

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

7.11.9 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 M_B taken from method typing context.

If m represents a quantum operator $\mathcal{E}_m$, the operator $\mathcal{E}_m$ is applied to a quantum subsystem specified by the parameters $\mathbf{v}_1 = \langle\!\langle r_1, v_1 \rangle\!\rangle_{\mathsf{T}_1}, \ldots, @\mathbf{v}_n = \langle\!\langle r_n, v_n \rangle\!\rangle_{\mathsf{T}_n}$. 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 **UV** occurs (OP-METHODCALLQUNINIT).

The condition that all manipulated quantum system are physically different can be reformulated as: all the indices in lists $l_{r_1}, \ldots, l_{r_n}$ are mutually different, *ie.* $\mathrm{set}_{[\circlearrowleft]}(l_{r_j}) \cap \mathrm{set}_{[\circlearrowleft]}(l_{r_k}) = \emptyset$ and $|l_{r_j}|_{[\circlearrowleft]} = |\,\mathrm{set}_{[\circlearrowleft]}(l_{r_j})|$ for all $1 \leq j, k \leq n,\ j \neq k$. If this condition is not satisfied, runtime error **OQV** is invoked (OP-METHODCALLQOVERLAP).

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

OP-DoMethodCallCl $\quad [gs \,|\, (lms, vp_G, \underline{m(\mathbf{v}_1, \ldots, \mathbf{v}_n)}\ ts) \longrightarrow_e [gs \,|\, (lms', [vp_G \circ_G [\Box \circ_L vp']], M_B(m) \underline{\circ_{\mathbf{M}}} ts)]$

where $vp' = ([a_1 \mapsto r'_1, \ldots, a_n \mapsto r'_n], [], [], [a_1 \mapsto \mathsf{T}_1, \ldots, a_n \mapsto \mathsf{T}_n, @retVal \mapsto \mathsf{T}])$
$lms' = lms[r'_1 \mapsto v_1]_+ \ldots [r'_n \mapsto v_n]_+$

given that $\mathbf{v}_1 = \langle\!\langle r_1, v_1 \rangle\!\rangle_{\mathsf{T}_1}, \ldots, \mathbf{v}_n = \langle\!\langle r_n, v_n \rangle\!\rangle_{\mathsf{T}_n}$

$$r'_i = \begin{cases} (Classical, nc_i) & \text{if } r_i = \mathsf{none} \\ r_i & \text{otherwise} \end{cases}$$

where all nc_i are mutually different natural numbers
such that $lms((Classical, nc_i))$ is not defined
m represents a classical method and
$\mathsf{T}\ m(\mathsf{T}_1\ a_1, \ldots \mathsf{T}_n\ a_n)$ is a header of method m

OP-MethodCallQUninit $\quad [gs \,|\, (lms, vp, \underline{m(\mathbf{v}_1, \ldots, \mathbf{v}_n)}\ ts)] \longrightarrow_{rte} [gs \,|\, (lms, vp, \mathbf{UV})]$

given that $\mathbf{v}_1 = \langle\!\langle r_1, v_1 \rangle\!\rangle_{\mathsf{T}_1}, \ldots, \mathbf{v}_n = \langle\!\langle r_n, v_n \rangle\!\rangle_{\mathsf{T}_n}$
$\exists i : v_i = \bot$

OP-MethodCallQOverlap $\quad [gs \,|\, (lms, vp, \underline{m(\mathbf{v}_1, \ldots, \mathbf{v}_n)}\ ts)] \longrightarrow_{rte} [gs \,|\, (lms, vp, \mathbf{OQV})]$

given that $\mathbf{v}_1 = \langle\!\langle r_1, v_1 \rangle\!\rangle_{\mathsf{T}_1}, \ldots, \mathbf{v}_n = \langle\!\langle r_n, v_n \rangle\!\rangle_{\mathsf{T}_n}$
$v_1 = (GQuantum, l_{r_1}), \ldots, v_n = (GQuantum, l_{r_n})$
$\mathrm{set}_{[\circlearrowright]}(l_{r_j}) \cap \mathrm{set}_{[\circlearrowright]}(l_{r_k}) \neq \emptyset$ for some $1 \leq j, k \leq n,\ j \neq k$
or
$|l_{r_j}|_{[\circlearrowright]} \neq |\,\mathrm{set}_{[\circlearrowright]}(l_{r_j})|$ for some $1 \leq j \leq n$

OP-DoMethodCallQ $[gs \,|\, (lms, vp, \underline{m(\mathbf{v}_1, \ldots, \mathbf{v}_n)}\ ts)] \longrightarrow_v$
$[gs' \,|\, (lms, vp, \underline{\langle\!\langle \mathsf{none}, \bot \rangle\!\rangle_{\mathsf{void}}}\ ts)]$

where $gs' = ((\rho', L), C)$
$\rho' = \Pi^T(\mathcal{E}_m \otimes I_{\bar{d}}(\Pi\rho\Pi^T))\Pi$

given that $\mathbf{v}_1 = \langle\!\langle r_1, v_1 \rangle\!\rangle_{\mathsf{T}_1}, \ldots, \mathbf{v}_n = \langle\!\langle r_n, v_n \rangle\!\rangle_{\mathsf{T}_n}$
$gs = ((\rho, L), C)$
$v_1 = (GQuantum, l_{r_1}), \ldots, v_n = (GQuantum, l_{r_n})$
$\mathrm{set}_{[\circlearrowright]}(l_{r_j}) \cap \mathrm{set}_{[\circlearrowright]}(l_{r_k}) = \emptyset$ for all $1 \leq j, k \leq n,\ j \neq k$
$|l_{r_j}|_{[\circlearrowright]} = |\,\mathrm{set}_{[\circlearrowright]}(l_{r_j})|$ for all $1 \leq j \leq n$
Π is a permutation matrix which places affected quantum systems to the head of ρ in the order given by $\mathbf{v}_1, \ldots, \mathbf{v}_n$
m represents a quantum operator $\mathcal{E}_m$

7.11.10 Returning from a method

When a method evaluation finishes, the control is passed back to its caller. The place where the called method was invoked by the caller is marked by the $\circ_{\mathbf{M}}$ symbol (see OP-DoMethodCallCl). 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 $\circ_{\mathbf{M}}$ (OP-ReturnVoid). When the method returns a value, the return value is evaluated first (OP-ReturnExpr) and then the return value is then left on top of the stack after popping all symbols up to $\circ_{\mathbf{M}}$ from the stack (OP-ReturnValue).

OP-ReturnVoid $[gs \,|\, (lms, [vp_G \circ_G vp], \underline{\mathbf{return;}}\ ts_M\, \underline{\circ_{\mathbf{M}}}\, ts)] \longrightarrow_{ret}$
$[gs \,|\, (lms, vp_G, \underline{\langle\!\langle \mathsf{none}, \bot \rangle\!\rangle_{\mathsf{void}}}\ ts)]$
where ts_M does not contain $\circ_{\mathbf{M}}$

OP-ReturnVoidImpl $[gs \,|\, (lms, [vp_G \circ_G vp],\ \underline{\circ_{\mathbf{M}}}\, ts)] \longrightarrow_s$
$[gs \,|\, (lms, vp_G, \underline{\langle\!\langle \mathsf{none}, \bot \rangle\!\rangle_{\mathsf{void}}}\ ts)]$

OP-ReturnExpr $[gs \,|\, (lms, vp, \underline{\mathbf{return}\ E\mathbf{;}}\ ts)] \longrightarrow_e$
$[gs \,|\, (lms, vp, \underline{E}\ \underline{\mathbf{return}\ \bullet\mathbf{;}}\ ts)$

OP-ReturnValue $[gs \,|\, (lms, [vp_G \circ_G vp], \underline{\mathbf{return\ v;}}\ ts_M\, \underline{\circ_{\mathbf{M}}}\, ts)] \longrightarrow_{ret}$
$[gs \,|\, (lms, vp_G, \underline{\mathbf{v}}\ ts)]$
where ts_M does not contain $\circ_{\mathbf{M}}$

7.11.11 Forking

Forking creates a new process which is started from given method. As **fork** contains a method call construct, the rule OP-ForkExpr for evaluation of arguments 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. The non-duplicable values passed as parameters to the forked method are unmapped from the parent process memory.

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

OP-DoFork $[gs \,|\, (lms, vp, \underline{\textbf{fork}\ m(\mathbf{v}_1, \ldots, \mathbf{v}_n)\textbf{;}}\ ts)] \longrightarrow_p$
$[gs \,|\, (lms'_1, vp, ts) \parallel (lms'_2, \blacksquare, \underline{m(\mathbf{v}'_1, \ldots, \mathbf{v}'_n)})]$

where $lms'_1 = unmap_{nd}(\{r_1, \ldots, r_n\}, lms_1)$
$lms'_2 = ([], [], [], [])[r'_1 \mapsto v_1]_+ \ldots [r'_n \mapsto v_n]_+$
$\mathbf{v}'_1 = \langle\!\langle r'_1, v_1 \rangle\!\rangle_{\mathsf{T}_1}, \ldots, \mathbf{v}'_n = \langle\!\langle r'_n, v_n \rangle\!\rangle_{\mathsf{T}_n}$

given that $\mathbf{v}_1 = \langle\!\langle r_1, v_1 \rangle\!\rangle_{\mathsf{T}_1}, \ldots, \mathbf{v}_n = \langle\!\langle r_n, v_n \rangle\!\rangle_{\mathsf{T}_n}$

$$r'_i = \begin{cases} (Classical, nc_i) & \text{if } r_i = \mathsf{none} \\ r_i & \text{otherwise} \end{cases}$$

where all nc_i are mutually different natural numbers such that $lms_1((Classical, nc_i))$ is not defined

---

7.11.12 Measurement

Measurement is performed when **measure**$(b, e_1, \ldots, e_n)$ primitive method is evaluated. 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 $\mathbf{v}_1 = \langle\!\langle r_1, v_1 \rangle\!\rangle_{\mathsf{T}_1}, \ldots, \mathbf{v}_n = \langle\!\langle r_n, v_n \rangle\!\rangle_{\mathsf{T}_n}$. 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 **UV**occurs (OP-MeasureUninit).

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 $1 \leq j, k \leq n,\ j \neq k$. If this condition is not satisfied, runtime error **OQV** is invoked (OP-MeasureOverlap).

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, $\bar{d} = d/d_{q_{qsi}}$.

In quantum mechanics, the possible results of the measurement are eigenvalues of the observable. We assign to each eigenvalue an index in a list of all eigenvalues. This index is returned as a result of evaluated **measure** expression. Indeed, it is possible that two or more eigenvalues are the same (they are called *degenerate* eigenvalues). In this case, the obtained result is the first index of corresponding eigenvalue in the list. The list is indexed from zero.

Now we can formulate rules for measurement:

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

OP-MeasureUninit
$$[gs \,|\, (lms, vp, \underline{\mathbf{measure}(\mathbf{v}_b, \mathbf{v}_1, \ldots, \mathbf{v}_n)}\ ts)] \longrightarrow_{rte} [gs \,|\, (lms, vp, \mathbf{UV})]$$
given that $\mathbf{v}_1 = \langle\!\langle r_1, v_1 \rangle\!\rangle_{\mathsf{T}_1}, \ldots, \mathbf{v}_n = \langle\!\langle r_n, v_n \rangle\!\rangle_{\mathsf{T}_n}$
$\exists i : v_i = \bot$

OP-MeasureOverlap
$$[gs \,|\, (lms, vp, \underline{\mathbf{measure}(\mathbf{v}_b, \mathbf{v}_1, \ldots, \mathbf{v}_n)}\ ts)] \longrightarrow_{rte} [gs \,|\, (lms, vp, \mathbf{OQV})]$$
given that $\mathbf{v}_1 = \langle\!\langle r_1, v_1 \rangle\!\rangle_{\mathsf{T}_1}, \ldots, \mathbf{v}_n = \langle\!\langle r_n, v_n \rangle\!\rangle_{\mathsf{T}_n}$
$v_1 = (GQuantum, l_{r_1}), \ldots, v_n = (GQuantum, l_{r_n})$
$\mathrm{set}_{[\circlearrowright]}(l_{r_j}) \cap \mathrm{set}_{[\circlearrowright]}(l_{r_k}) \neq \emptyset$ for some $1 \leq j, k \leq n,\ j \neq k$
or
$|l_{r_j}|_{[\circlearrowright]} \neq |\,\mathrm{set}_{[\circlearrowright]}(l_{r_j})|$ for some $1 \leq j \leq n$

OP-DoMeasure
$$[gs \,|\, (lms, vp, \underline{\mathbf{measure}(\mathbf{v}_b, \mathbf{v}_1, \ldots, \mathbf{v}_n)}\ ts)] \longrightarrow_v \boxplus_i\ p_i \bullet [gs_i \,|\, (lms, vp, \underline{\langle\!\langle \mathsf{none}, fdi(i) \rangle\!\rangle_{\mathsf{int}}}\ ts)]$$
where $gs_i = ((\rho_i, L), C)$
$p_i = \mathrm{Tr}\ [\Pi^T (P_i \otimes I_{\bar{d}}) \Pi \rho \Pi^T (P_i \otimes I_{\bar{d}})^\dagger \Pi]$
$$\rho_i = \frac{\Pi^T (P_i \otimes I_{\bar{d}}) \Pi \rho \Pi^T (P_i \otimes I_{\bar{d}})^\dagger \Pi}{p_i}$$
$fdi(i)$ is the first index of i-th eigenvalue in the list of observable eigenvalues (for the case of degenerate eigenvalues)
given that $\mathbf{v}_1 = \langle\!\langle r_1, v_1 \rangle\!\rangle_{\mathsf{T}_1}, \ldots, \mathbf{v}_n = \langle\!\langle r_n, v_n \rangle\!\rangle_{\mathsf{T}_n}$
$v_1 = (GQuantum, l_{r_1}), \ldots, v_n = (GQuantum, l_{r_n})$
$gs = ((\rho, L), C)$
$\mathrm{set}_{[\circlearrowright]}(l_{r_j}) \cap \mathrm{set}_{[\circlearrowright]}(l_{r_k}) = \emptyset$ 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 $\mathbf{v}_1, \ldots, \mathbf{v}_n$

7.11.13 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. When either the sending channel end or the sent value is undefined, or a receiving process attempts to receive from uninitialized channel end, a runtime error **UV** occurs (rules OP-SendUninit and OP-RecvUninit).

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

OP-SendExpr1 $[gs \mid (lms, vp, \underline{\mathbf{send}(E_1, E_2);}\ ts)] \longrightarrow_e$
$[gs \mid (lms, vp, \underline{E_1}\ \underline{\mathbf{send}(\bullet, E_2);}\ ts]$

OP-SendExpr2 $[gs \mid (lms, vp, \underline{\mathbf{send}(\mathbf{v}_c, E);}\ ts)] \longrightarrow_e$
$[gs \mid (lms, vp, \underline{E}\ \underline{\mathbf{send}(\mathbf{v}_c, \bullet);}\ ts]$

OP-SendUninit $[gs \mid (lms, vp, \underline{\mathbf{send}(\mathbf{v}_1, \mathbf{v}_2);}\ ts)] \longrightarrow_{rte} [gs \mid (lms, vp, \mathbf{UV}]$
given that $\mathbf{v}_1 = \langle\!\langle r_1, v_1 \rangle\!\rangle_{\mathsf{T}_1}$ and $v_1 = \bot$ or $\mathbf{v}_2 = \langle\!\langle r_2, v_2 \rangle\!\rangle_{\mathsf{T}_2}$ and $v_2 = \bot$)

OP-RecvExpr $[gs \mid (lms, vp, \underline{\mathbf{recv}(E)}\ ts)] \longrightarrow_e [gs \mid (lms, vp, \underline{E}\ \underline{\mathbf{recv}(\bullet)}\ ts]$

OP-RecvUninit $[gs \mid (lms, vp, \underline{\mathbf{recv}(\mathbf{v})}\ ts)] \longrightarrow_{rte} [gs \mid (lms, vp, \mathbf{UV}]$
given that $\mathbf{v} = \langle\!\langle r, v \rangle\!\rangle_{\mathsf{T}}$ and $v = \bot$

OP-SendRecv $[gs \mid (lms_1, vp_1, \underline{\mathbf{send}(\mathbf{v}_{c_1}, \mathbf{v}_e);}\ ts_1) \parallel (lms_2, vp_2, \underline{\mathbf{recv}(\mathbf{v}_{c_2})}\ ts_2)] \longrightarrow_p$
$[gs \mid (lms'_1, vp_1, ts_1) \parallel (lms'_2, vp_2, \underline{\langle\!\langle lr'_2, lv'_2 \rangle\!\rangle_{\mathsf{T}}}\ ts_2)]$

where $lms'_1 = unmap_{nd}(sentRef, lms_1)$

$$lr'_2 = \begin{cases} lr_1 & \text{if } sentRef \in Ref_{nd} \\ \mathsf{none} & \text{otherwise} \end{cases}$$

$$lv'_2 = sentVal$$

$$lms'_2 = \begin{cases} lms_2[sentRef \mapsto sentVal] & \text{if } sentRef \in Ref_{nd} \\ lms_2 & \text{otherwise} \end{cases}$$

given that $\mathbf{v}_e = \langle\!\langle sentRef, sentVal \rangle\!\rangle_{\mathsf{T}}$
$\mathbf{v}_{c_1} = \langle\!\langle c_1Ref, c_1Val \rangle\!\rangle_{\mathsf{T}}$
$\mathbf{v}_{c_2} = \langle\!\langle c_2Ref, c_2Val \rangle\!\rangle_{\mathsf{T}}$
$c_1Ref \neq \mathsf{none}$ and $c_2Ref \neq \mathsf{none}$
$c_1Val = c_2Val$ (both ends refer to the same channel)

7.12 Rule index

$\longrightarrow_e$: OP-AssignExpr, OP-DoMethodCallCl, OP-ForkExpr, OP-IfExpr, OP-MeasureExpr, OP-MethodCallExpr, OP-PromoExpr, OP-RecvExpr, OP-ReturnExpr, OP-SendExpr1, OP-SendExpr2, OP-SubstE, OP-SubstS

$\longrightarrow_p$: OP-DoFork, OP-SendRecv

$\longrightarrow_r$: OP-BlockHead, OP-Bracket, OP-VarDeclMulti, OP-While

$\longrightarrow_{ret}$: OP-ReturnValue, OP-ReturnVoid

$\longrightarrow_{rte}$: OP-AssignQAValueBad, OP-MeasureUninit, OP-MeasureOverlap, OP-MethodCallQUninit, OP-MethodCallQOverlap, OP-RecvUninit, OP-SendUninit

$\longrightarrow_s$: OP-Block, OP-BlockEnd, OP-IfFalse, OP-IfTrue, OP-PromoForget, OP-ReturnVoidImpl, OP-Skip, OP-VarDecl, OP-VarDeclAlF, OP-VarDeclChE

$\longrightarrow_v$: OP-AllocC, OP-AllocQ, OP-AssignNewValue, OP-AssignQAValue, OP-AssignQValue, OP-AssignValue, OP-DoMeasure, OP-DoMethodCallQ, OP-Var

8 Type soundness

In this section, we prove type soundness (see *eg.* [WF94]) for LanQ.

To prove type soundness (also known as Subject reduction) theorem, we first define typing on configurations. Then in series of progress lemmata proofs, we prove that any typable configuration can be reduced by some semantic rule. After this proof, we prove series of type preservation lemmata stating that if a configuration gets reduced to another configuration, either a runtime error occurs or the type of the configuration is preserved. These lemmata straightforwardly imply type soundness of the language.

However, we cannot prove the type soundness property for the unrestricted language because it is possible that a program gets to a stuck configuration during the evaluation. This can happen because the **send** and **recv** constructs are blocking and synchronizing actions. Consider a process which attempts to send a value over a channel where no other process is receiving the values from the other end of the channel. Then no semantic rule can be applied to the sending process. Symmetrically, it is indeed possible to define a process which attempts to receive a value from a channel where no other process sends a value over this channel. Such a process also cannot evolve, therefore the evaluation can get to a stuck configuration.

In other words, we can prove type soundness only for the noncommunicating part of the language. Nevertheless, if to each **send** statement there is a corresponding **recv** expression, then it can be proved that the evaluation of a well-typed configuration never gets stuck, hence type soundness can be proved for the unrestricted language.

8.1 Typing of configurations

To prove type soundness, we follow the approach of [BPP03]. Before proving type preservation, we define typing of configurations in this subsection in Figures 14 and 15.

Any configuration $C = [gs \mid ls_1 \parallel \cdots \parallel ls_n]$ is assigned a type T, written as $C : T$ which is a cartesian product of types of local process configurations $ls_1, \ldots, ls_n$. If the type of ls_i is T_i then $T = (T_1, \ldots, T_n)$ (see the typing rule T-Config).

We call a configuration which is assigned a type a *well-typed configuration*.

The typing rules provide rules for well-formedness of configurations, hence also for local process configurations. This indeed means that structure of variable properties and a term stack are tightly connected: To any block end mark $\circ_{\mathbf{L}}$ on the term stack, the variable properties must contain a nonempty list of variable properties $[vp \circ_L vp_L] \in VarProp_L$ (see rule TC-BlockEnd); to any method call mark $\circ_{\mathbf{M}}$ on the term stack, the variable properties must contain a nonempty list of lists of variable properties $[vp_G \circ_G vp] \in VarProp$ (see rules TC-RetHole, TC-RetExpr, TC-RetVoid and TC-RetImpl).

Typing of local process configurations as depicted in Figure 14 needs a deeper explanation. Contrary to usual typing of configurations known *eg.* from λ-calculi where one configuration contains only one expression, we deal with the situation where there are many "expressions" in one configuration. These expressions are in our case term stack elements and they altogether form a term stack of a local process configuration.

The type of a local process configuration is defined as $\sigma \to \tau$ for types σ, τ. The type σ specifies the type of the hole in the top term stack element, τ specifies the type of the result value. When there is no hole in the top term stack element, σ is void.

The top term stack element TE can contain at most one hole. As the hole type σ is always known and the variable typing context can be deduced from variable properties, the type τ' of TE is derivable from the typing rules defined in Section 5.

If TE is the only term stack element in the local process configuration, its type determines the type of the result value τ. Otherwise, the type τ' is the type of a hole in the term stack

TC-EMPTY
$$\frac{}{M;\Gamma \vdash_C (lms, \blacksquare, \varepsilon) : \tau \to \tau}$$

TC-RUNTIME
$$\frac{}{M;\Gamma \vdash_C (lms, vp, \underline{RTErr}) : \sigma \to \tau}$$

TC-EXPRHOLE
$$\frac{M;\Gamma, vpContext(vp), \bullet : \sigma \vdash_T Ec : \tau' \quad M;\Gamma \vdash_C (lms, vp, ts) : \tau' \to \tau}{M;\Gamma \vdash_C (lms, vp, \underline{Ec}\ ts) : \sigma \to \tau}$$

TC-STATHOLE
$$\frac{M;\Gamma, vpContext(vp), \bullet : \sigma \vdash_T Sc : \mathsf{void} \quad M;\Gamma \vdash_C (lms, vp, ts) : \mathsf{void} \to \tau}{M;\Gamma \vdash_C (lms, vp, \underline{Sc}\ ts) : \sigma \to \tau}$$
if $Sc \neq$ **return** $\bullet$ **;**

TC-RETHOLE
$$\frac{M;\Gamma \vdash_C (lms, vp_G, ts) : typeOf(@retVal, [vp_G \circ_G vp]) \to \tau}{M;\Gamma \vdash_C (lms, [vp_G \circ_G vp], \underline{\textbf{return}\ \bullet\textbf{;}} \ldots \underline{\circ_{\mathbf{M}}}\ ts) : \sigma \to \tau}$$

TC-EXPRCLO
$$\frac{M;\Gamma, vpContext(vp) \vdash_T E : \tau' \quad M;\Gamma \vdash_C (lms, vp, ts) : \tau' \to \tau}{M;\Gamma \vdash_C (lms, vp, \underline{E}\ ts) : \sigma \to \tau}$$

TC-STATCLO
$$\frac{M;\Gamma, vpContext(vp) \vdash_T S : \mathsf{void} \quad M;\Gamma \vdash_C (lms, vp, ts) : \mathsf{void} \to \tau}{M;\Gamma \vdash_C (lms, vp, \underline{S}\ ts) : \sigma \to \tau}$$
if $S \neq$ **return** E**;**, $S \neq$ **return;**

TC-RETEXPR
$$\frac{\begin{array}{c} M;\Gamma, vpContext([vp_G \circ_G vp]) \vdash_T E : typeOf(@retVal, [vp_G \circ_G vp]), \\ M;\Gamma \vdash_C (lms, vp_G, ts) : typeOf(@retVal, [vp_G \circ_G vp]) \to \tau \end{array}}{M;\Gamma \vdash_C (lms, [vp_G \circ_G vp], \underline{\textbf{return}\ E\textbf{;}} \ldots \underline{\circ_{\mathbf{M}}}\ ts) : \sigma \to \tau}$$

TC-RETVOID
$$\frac{M;\Gamma \vdash_C (lms, vp_G, ts) : \mathsf{void} \to \tau}{M;\Gamma \vdash_C (lms, [vp_G \circ_G vp], \underline{\textbf{return;}} \ldots \underline{\circ_{\mathbf{M}}}\ ts) : \sigma \to \tau}$$

TC-RETIMPL
$$\frac{M;\Gamma \vdash_C (lms, vp_G, ts) : typeOf(@retVal, [vp_G \circ_G vp]) \to \tau}{M;\Gamma \vdash_C (lms, [vp_G \circ_G vp], \underline{\circ_{\mathbf{M}}}\ ts) : \sigma \to \tau}$$

TC-BLOCKHEAD
$$\frac{M;\Gamma \vdash_C (lms, vp, \underline{Be_0}\ \underline{Be_1\ \ldots\ Be_n}\ ts) : \mathsf{void} \to \tau}{M;\Gamma \vdash_C (lms, vp, \underline{Be_0\ Be_1\ \ldots\ Be_n}\ ts) : \sigma \to \tau}$$

TC-BLOCKEND
$$\frac{M;\Gamma \vdash_C (lms, [vp_G \circ_G vp], ts) : \mathsf{void} \to \tau}{M;\Gamma \vdash_C (lms, [vp_G \circ_G [vp \circ_L vp_L]], \underline{\circ_{\mathbf{L}}}\ ts) : \mathsf{void} \to \tau}$$

TC-VARDECLMULTI
$$\frac{M;\Gamma \vdash_C (lms, vp, \underline{T\ I_0\textbf{;}}\ \underline{T\ I_1, \ldots, I_n\textbf{;}}\ ts) : \mathsf{void} \to \tau}{M;\Gamma \vdash_C (lms, vp, \underline{T\ I_0, I_1, \ldots, I_n\textbf{;}}\ ts) : \sigma \to \tau}$$

TC-VARDECLONE
$$\frac{\begin{array}{c} varRef(I, vp) \text{ is undefined} \\ M;\Gamma \vdash_C (lms, vp', ts) : \mathsf{void} \to \tau \end{array}}{M;\Gamma \vdash_C (lms, vp, \underline{T\ I\textbf{;}}\ ts) : \sigma \to \tau}$$
where $vp' = vp[I \mapsto \mathsf{T}]_{+,type}$

TC-VARDECLCHE
$$\frac{\begin{array}{c} varRef(I_i, vp) \text{ is undefined for } i = 0, ..2, \\ I_0, I_1, I_2 \text{ are mutually different,} \\ M;\Gamma \vdash_C (lms, vp', ts) : \mathsf{void} \to \tau \end{array}}{M;\Gamma \vdash_C (lms, vp, \underline{\mathsf{channel}[T]\ I_0\ \textbf{withends}[I_1,I_2]\textbf{;}}\ ts) : \sigma \to \tau}$$
where $vp' = vp[c \mapsto \mathsf{channel}[\mathsf{T}]]_{+,type}[c_0 \mapsto \mathsf{channelEnd}[\mathsf{T}]]_{+,type}[c_1 \mapsto \mathsf{channelEnd}[\mathsf{T}]]_{+,type}$

TC-VARDECLALF
$$\frac{\begin{array}{c} varRef(I_0, vp) \text{ is undefined,} \\ M;\Gamma \vdash_T I_i : \mathsf{T_i} \quad \mathsf{T_i} \text{ is a quantum type for } i = 1, ..n, \\ M;\Gamma \vdash_C (lms, vp', ts) : \mathsf{void} \to \tau \end{array}}{M;\Gamma \vdash_C (lms, vp, \underline{I_0\ \textbf{aliasfor}\ [I_1, \ldots, I_n]\textbf{;}}\ ts) : \sigma \to \tau}$$
where $vp' = vp[I_0 \mapsto \bigotimes_{\mathsf{i}=1}^{n} \mathsf{T_i}]_{+,type}$

Figure 14: Typing rules for local process configurations

T-MixedConf
$$\frac{\forall i \in 1,\ldots,q:\ M;\Gamma \vdash [GS_i \,|\, LS_{i,1} \parallel \cdots \parallel LS_{i,n}] : \tau_i}{M;\Gamma \vdash \boxplus_{i=1}^{q} p_i \bullet [GS_i \,|\, LS_{i,1} \parallel \cdots \parallel LS_{i,n}]}$$

T-Config
$$\frac{\forall i \in 1,\ldots,n:\ M;\Gamma \vdash_C LS_i : \mathsf{void} \rightarrow \tau_i}{M;\Gamma \vdash [GS \,|\, LS_1 \parallel \cdots \parallel LS_n] : \prod_{i=1}^{n} \tau_i}$$

Figure 15: Typing rules for configurations

element right beneath TE and we can continue with typing the local process configuration where TE is popped from the term stack.

To derive variable typing context from local process configuration variable properties, we define a function $vpContext$. This function is defined as follows:

Definition 8.1. *We define a function $vpContext : VarProp \rightarrow (Names \rightharpoonup Types)$ which creates a variable typing context from variable properties as:*

$$\begin{aligned} vpContext([L_1 \circ_G K]) &\stackrel{\text{def}}{=} vpContext_L(K) \\ vpContext(\blacksquare) &\stackrel{\text{def}}{=} [] \end{aligned}$$

where $vpContext_L : VarProp_L \rightarrow (Names \rightharpoonup Types)$ is a function for getting a variable typing context from local variable properties defined as:

$$\begin{aligned} vpContext_L([L_1 \circ_L (f_{var}, f_{ch}, f_{qa}, f_{type})]) &\stackrel{\text{def}}{=} vpContext_L(L_1) * f_{type} \\ vpContext_L(\square) &\stackrel{\text{def}}{=} [] \end{aligned}$$

8.2 Progress

In this subsection, we prove series of progress lemmata, *ie.* assertions claiming that any well-typed configuration which is not terminal can be reduced by some semantic rule. It also follows from the proof that the choice of the semantic rule is unique, hence the semantics is deterministic.

Lemma 8.2 (Progress Lemma for probabilism)**.** *If $C_0 = \boxplus_{i=1}^{q} p_i \bullet [gs_i \,|\, ls_{i,0}]$, $q > 1$ and $\vdash C_0 : \tau$ then there exists a configuration C_1 such that $C_0 \xrightarrow{p} C_1$.*

Proof. Such a mixed configuration C_0 is reduced by the rule NP-ProbEvol. □

Lemma 8.3 (Progress Lemma for local processes)**.** *If $C_0 = [gs_0 \,|\, (lms_0, vp_0, \underline{TE}\ ts_0)]$ is not terminal, $TE \neq$ **recv(v)**, $TE \neq$ **send(v$_1$,v$_2$);**, and $\vdash C_0 : \tau$ then there exists a mixed configuration C_1 such that $C_0 \longrightarrow C_1$.*

Proof. By case analysis of all possibilities of the top term stack element TE:

Case $TE =$ new $\mathsf{T}()$: As C_0 is well-typed, we know that T is either Q_d or $\mathsf{channel}[\mathsf{T}]$ (from the rule T-Alloc). If $\mathsf{T} = \mathsf{Q}_d$, then the configuration C_0 is reduced by the rule OP-AllocQ. Otherwise $\mathsf{T} = \mathsf{channel}[\mathsf{T}]$ and the configuration C_0 is reduced by the rule OP-AllocC.

Case $TE = I = E$: As C_0 is well typed, we know that types of I and E match (from the rule T-Assign). If $E = \mathbf{v} = \langle\!\langle lr, lv \rangle\!\rangle_{\mathsf{T}}$ then one of the following rules is applied:

- OP-AssignNewValue if $lv \neq \bot$ and $lr = \mathsf{none}$,

- OP-AssignQValue if $lr = (Quantum, q)$ and $aliasSubsyst(I, vp_0)$ is not defined,
- OP-AssignQAValue if $lr = (Quantum, q)$ and $aliasSubsyst(I, vp_0)$ is defined,
- OP-AssignValue otherwise.

Otherwise, the configuration C_0 is reduced by the rule OP-AssignExpr.

Case $TE = I(\widetilde{E})$**:** As C_0 is well typed, we know that I denotes either a quantum operator or a classical method (from the rule T-MethodCall). If $\widetilde{E} = \widetilde{\mathbf{v}}$ then one of the following rules is applied:

- OP-DoMethodCallCl if I represents a classical method,
- OP-DoMethodCallQ if I represents a quantum operator.

Otherwise the configuration C_0 is reduced by the rule OP-MethodCallExpr.

Case $TE = \mathbf{measure}(\widetilde{E})$**:** If $\widetilde{E} = \widetilde{\mathbf{v}}$ then the configuration C_0 is reduced by the rule OP-DoMeasure, otherwise it is reduced by the rule OP-MeasureExpr.

Case $TE = \mathbf{recv}(E)$**:** As E cannot be $\mathbf{v}$ (from assumptions) the configuration C_0 is reduced by the rule OP-RecvExpr.

Case $TE = I$**:** As C_0 is well typed, we know that $varRef(I, vp_0)$ is defined. Configuration C_0 is reduced by the rule OP-Var.

Case $TE = \mathbf{v}$**:** If $|ts_0| = 1$ then C_0 is a terminal configuration. Otherwise let UT be the first symbol under the top element of the stack. As the configuration is well-typed, we know that UT must be a symbol containing a hole. Now:

Case $UT = Ec$**:** Configuration C_0 is reduced by the rule OP-SubstE.

Case $UT = Sc$**:** Configuration C_0 is reduced by the rule OP-SubstS.

Case $TE = (E)$**:** Configuration C_0 is reduced by the rule OP-Bracket.

Case $TE = T\ \widetilde{I}$**;:** If $TE = T\ I$; then the configuration C_0 is reduced by the rule OP-VarDecl. Otherwise it is reduced by the rule OP-VarDeclMulti.

Case $TE = \mathsf{channel}[\mathsf{T}]\ I$ **withends**$[I,I]$**;:** Configuration C_0 is reduced by the rule OP-VarDeclChE.

Case $TE = q$ **aliasfor** $[q_0, \ldots, q_n]$**;:** Configuration C_0 is reduced by the rule OP-VarDeclAlF.

Case $TE =$ **;:** Configuration C_0 is reduced by the rule OP-Skip.

Case $TE = PE$**;:** If $PE = \mathbf{v}$ then the configuration C_0 is reduced by the rule OP-PromoForget, otherwise it is reduced by OP-PromoExpr.

Case $TE = \circ_L$**:** Configuration C_0 is reduced by the rule OP-BlockEnd.

Case $TE = \overline{Be}$**:** Configuration C_0 is reduced by the rule OP-BlockHead.

Case $TE = \{B\}$**:** Configuration C_0 is reduced by the rule OP-Block.

Case $TE =$ **if** $(E)\ S_1$ **else** S_2**:** If $E = \mathbf{v}$, then the configuration is reduced by the rule OP-IfTrue (OP-IfFalse) when $\mathbf{v}$ is **true** (**false**). Otherwise, the configuration C_0 is reduced by the rule OP-IfExpr.

Case $TE = \textbf{while}\ (E)\ S$**:** Configuration C_0 is reduced by the rule OP-WHILE.

Case $TE = \circ_{\mathbf{M}}$**:** Configuration C_0 is reduced by the rule OP-RETURNVOIDIMPL.

Case $TE = \textbf{return;}$**:** Configuration C_0 is reduced by the rule OP-RETURNVOID.

Case $TE = \textbf{return}\ E\textbf{;}$**:** If $E = \mathbf{v}$ then the configuration C_0 is reduced by the rule OP-RETURNVALUE, otherwise it is reduced by OP-RETURNEXPR.

Case $TE = \textbf{fork}\ m(\widetilde{E})\textbf{;}$**:** If $\widetilde{E} = \widetilde{\mathbf{v}}$ then the configuration C_0 is reduced by the rule OP-DOFORK. Otherwise it is reduced by the rule OP-FORKEXPR.

Case $TE = \textbf{send}(E_1,E_2)\textbf{;}$**:** If $E_1 \neq \mathbf{v}$ then the configuration C_0 is reduced by the rule OP-SENDEXPR1. If $E_2 \neq \mathbf{v}$, the configuration C_0 is reduced by the rule OP-SENDEXPR2. Case $E_1 = \mathbf{v}_1$ and $E_2 = \mathbf{v}_2$ is prohibited by the assumptions.

□

Lemma 8.4 (Progress Lemma for communication)**.** *If* $C_0 = [gs_0 \,|\, (lms_0, vp_0, \underline{\textbf{recv(v)}}\ ts_0) \parallel (lms_1, vp_1, \underline{\textbf{send(v}_1\textbf{,v}_2\textbf{);}}\ ts_1) \parallel ls_2 \parallel \cdots \parallel ls_n] : \tau$ *then there exists a configuration* C_1 *such that* $C_0 \longrightarrow C_1$*.*

Proof. Such a configuration is not terminal because of the elements $\textbf{recv(v)}$ and $\textbf{send(v}_1\textbf{,v}_2\textbf{);}$ on tops of the process term stacks. It is reduced by the rule OP-SENDRECV. □

Corollary 8.5. *It is possible that a process evolves to a stuck configuration. This is the case when one process attempts to send/receive a value over a channel end and there is no matching process receiving/sending over the corresponding channel end.*

8.3 Type preservation

8.3.1 Evaluation theorems

In the definition of LanQ semantics, we assumed that an expression always evaluates to a value (or diverges or invokes a runtime error). This assumption was used in all rules which manipulate a subexpression *Sub* of a statement or an expression: The subexpression *Sub* is pushed onto the top of the term stack and the place of the awaited result in the original statement/expression is marked with a hole • (see *eg.* the rule OP-ASSIGNEXPR). We expect that if the evaluation correctly finishes, the subexpression evaluates to an internal value that in the subsequent step replaces the hole.

However, we have not yet shown that the subexpression evaluation accomplishes this assertion. We have to prove that the statement/expression awaiting the result of the subexpression evaluation is not modified before the evaluation of the subexpression yields an internal value (unless a runtime error occurs).

This will be shown in this subsection. We will use the following semantic predicates on configurations:

- *ExpOk* which is defined for configurations where the top term stack element of the first local process configuration is some expression E, and
- *BFOk* and *StkRetOk* which is defined for configurations where the top term stack element of the first local process configuration is some block-forming statement$B$.

These predicates satisfiability is determined by the future evolution of the configurations in question.

We will further define inductive *syntactic* predicates $ExpOk_i$, $BFOk_i$ and show the connection between these syntactic predicates and the semantic ones.

The defined predicates are then used in the proof of the statement that if the syntactic predicate $RetOk$ defined in Section 5.1 is satisfied for a block-forming statement B then any control path of evaluation of B reaches a **return;** or **return** E**;** statement, or a runtime error, or diverges (see Lemma 8.15). This proof is later used in proving type preservation lemma for $\longrightarrow_s$ (see Lemma 8.21, proof of the case OP-ReturnVoidImpl).

Definition 8.6. *For an expression E, we define a predicate $ExpOk(E)$: $ExpOk(E)$ is satisfied iff for any configuration $C_0 = [gs_0 \,|\, (lms_0, vp_0, \underline{E}\ ts_0)] : \tau$, the evaluation of C_0 either diverges or reaches one of the following configurations:*

- $[gs_n \,|\, (lms_{0,n}, vp_{0,n}, \underline{\mathbf{v}}\ ts_0) \parallel \cdots \parallel (lms_{k,n}, vp_{k,n}, ts_{k,n})], k \geq 0$, *or*
- $[gs_n \,|\, (lms_{0,n}, vp_{0,n}, \underline{RTErr}) \parallel \cdots \parallel (lms_{k,n}, vp_{k,n}, ts_{k,n})], k \geq 0$.

Definition 8.7. *For a block-forming statement B, we define a predicate $BFOk(B)$: $BFOk(E)$ is satisfied iff for any configuration $C_0 = [gs_0 \,|\, (lms_0, vp_0, \underline{B}\ ts_0)] : \tau$, the evaluation either diverges or reaches one of the following configurations:*

- $[gs_n \,|\, (lms_{0,n}, vp_{0,n}, ts_0) \parallel \cdots \parallel (lms_{k,n}, vp_{k,n}, ts_{k,n})], k \geq 0$, *or*
- $[gs_n \,|\, (lms_{0,n}, vp_{0,n}, \underline{\mathbf{return;}}\ ts_{0,n}\ ts_0) \parallel \cdots \parallel (lms_{k,n}, vp_{k,n}, ts_{k,n})], k \geq 0$, $ts_{0,n}$ *does not contain* $\circ_{\mathbf{M}}$, *or*
- $[gs_n \,|\, (lms_{0,n}, vp_{0,n}, \underline{\mathbf{return\ v;}}\ ts_{0,n}\ ts_0) \parallel \cdots \parallel (lms_{k,n}, vp_{k,n}, ts_{k,n})], k \geq 0$, $ts_{0,n}$ *does not contain* $\circ_{\mathbf{M}}$, *or*
- $[gs_n \,|\, (lms_{0,n}, vp_{0,n}, \underline{RTErr}) \parallel \cdots \parallel (lms_{k,n}, vp_{k,n}, ts_{k,n})], k \geq 0$.

We further define a predicate $StkRetOk(B)$: $StkRetOk(E)$ is satisfied iff for any configuration $C_0 = [gs_0 \,|\, (lms_0, vp_0, \underline{B}\ \underline{\circ_{\mathbf{M}}}\ ts_0)] : \tau$, the evaluation either diverges or reaches one of the following configurations:

- $[gs_n \,|\, (lms_{0,n}, vp_{0,n}, \underline{\mathbf{v}}\ ts_0) \parallel \cdots \parallel (lms_{k,n}, vp_{k,n}, ts_{k,n})], k \geq 0$, *or*
- $[gs_n \,|\, (lms_{0,n}, vp_{0,n}, \underline{RTErr}) \parallel \cdots \parallel (lms_{k,n}, vp_{k,n}, ts_{k,n})], k \geq 0$.

Lemma 8.8. *Let B be a block-forming statement such that $BFOk(B)$. Then $StkRetOk(B)$.*

Proof. The evaluation starts from the following configuration:

$$[gs_0 \,|\, (lms_0, vp_0, \underline{B}\ \underline{\circ_{\mathbf{M}}}\ ts_0)]$$

From $BFOk(B)$ we know that evolution of this configuration:

- Diverges or gets to a configuration:

  $$[gs_n \,|\, (lms_n, vp_n, \underline{RTErr}) \parallel \cdots \parallel (lms_{k,n}, vp_{k,n}, ts_{k,n})]$$

  therefore the lemma holds.

- Gets to a configuration:

  $$[gs_{n'} \,|\, (lms_{n'}, vp_{n'}, \underline{\mathbf{return;}}\ ts_{n'}\ \underline{\circ_{\mathbf{M}}}\ ts_0) \parallel \cdots \parallel (lms_{k,n'}, vp_{k,n'}, ts_{k,n'})]$$

  By application of OP-ReturnVoid, the 0-th process gets to a configuration:

  $$[gs_{n'} \,|\, (lms_{n'}, vp_{n'}, \underline{\langle\!\langle \mathsf{none}, \bot \rangle\!\rangle_{\mathsf{void}}}\ ts_0) \parallel \cdots \parallel (lms_{k,n'}, vp_{k,n'}, ts_{k,n'})]$$

  therefore the lemma holds.

- Gets to a configuration:

$$[gs_{n'} \,|\, (lms_{n'}, vp_{n'}, \underline{\textbf{return v;}}\ ts_{n'} \ \underline{\circ_{\mathbf{M}}}\ ts_0) \parallel \cdots \parallel (lms_{k,n'}, vp_{k,n'}, ts_{k,n'})]$$

By application of OP-ReturnValue, the 0-th process gets to a configuration:

$$[gs_{n'} \,|\, (lms_{n'}, vp_{n'}, \underline{\mathbf{v}}\ ts_0) \parallel \cdots \parallel (lms_{k,n'}, vp_{k,n'}, ts_{k,n'})]$$

therefore the lemma holds.

□

We want to prove that for any expression E the predicate $ExpOk(E)$ is satisfied. We do this by defining inductive predicates $ExpOk_i$ and $BFOk_i$ with respect to the structure of E and B, respectively. The dependency is the following ($a \leftarrow b$ means a is dependent on b):

$$ExpOk_0 \longleftarrow BFOk_0 \longleftarrow ExpOk_1 \leftarrow \cdots \leftarrow ExpOk_i \longleftarrow BFOk_i \leftarrow \cdots$$

Definition 8.9. *Let E be an expression, B be a block-forming statement. We define predicates $ExpOk_0(E)$ and $BFOk_0(B)$:*

$$\begin{aligned} ExpOk_0(E) &\iff E \text{ does not contain } I(\widetilde{E}) \text{ as its subexpression.} \\ BFOk_0(B) &\iff B \text{ contains only such subexpressions } E \text{ which satisfy } ExpOk_0(E). \end{aligned}$$

For any $i \in \mathbb{N}_0$, we further define predicates $ExpOk_{i+1}(E)$ and $BFOk_{i+1}(B)$:

$$\begin{aligned} ExpOk_{i+1}(E) &\iff \text{For any subexpression } E_0 \text{ of } E,\ ExpOk_i(E_0) \text{ is satisfied} \\ &\qquad \text{or } E_0 = I(\widetilde{E}) \text{ and } BFOk_i(M_B(I)) \text{ is satisfied.} \\ BFOk_{i+1}(B) &\iff B \text{ contains only such subexpressions } E \text{ which satisfy } ExpOk_{i+1}(E). \end{aligned}$$

Lemma 8.10. *Let E be an expression such that $ExpOk_0(E)$. Then $ExpOk(E)$ is satisfied.*

Proof. By induction on the structure of E. Let $C_0 = [gs_0 \,|\, (lms_0, vp_0, \underline{E}\ ts_0)] : \tau$ be any configuration. Base case:

Case $E = \mathbf{v}$: $ExpOk(\mathbf{v})$ trivially.

Case $E = I$: By application of OP-Var we reach configuration $[gs \,|\, (lms, vp, \underline{\mathbf{v}}\ ts_0)]$, hence $ExpOk(E)$.

Case $E =$ new T(): By application of OP-AllocC/OP-AllocQ reaches configuration $[gs \,|\, (lms, vp, \underline{\mathbf{v}}\ ts_0)]$, hence $ExpOk(E)$.

Case $E = I = \mathbf{v}$: By application of OP-AssignNewValue/OP-AssignQAValue/OP-AssignQValue/OP-AssignValue we reach configuration $[gs \,|\, (lms, vp, \underline{\mathbf{v}}\ ts_0)]$, hence $ExpOk(E)$; by application of OP-AssignQAValueBad we reach configuration $[gs \,|\, (lms, vp, \underline{\mathbf{ISQV}})]$, hence $ExpOk(E)$.

Case $E =$ recv($\mathbf{v}$): By application of OP-SendRecv we reach configuration $[gs \,|\, (lms, vp, \underline{\mathbf{v}}\ \ ts_0)]$, hence $ExpOk(E)$; by application of OP-RecvUninit we reach configuration $[gs \,|\, (lms, vp, \underline{\mathbf{UV}})]$, hence $ExpOk(E)$.

Case $E =$ measure($\widetilde{\mathbf{v}}$): By subsequent application of rules OP-DoMeasure and NP-ProbEvol we reach configuration $[gs \,|\, (lms, vp, \underline{\mathbf{v}}\ \ ts_0)]$, hence $ExpOk(E)$; by application of OP-MeasureUninit we reach configuration $[gs \,|\, (lms, vp, \underline{\mathbf{UV}})]$, hence $ExpOk(E)$; by application of OP-MeasureOverlap we reach configuration $[gs \,|\, (lms, vp, \underline{\mathbf{OQV}})]$, hence $ExpOk(E)$.

Next we assume that the lemma holds for all subexpressions E_0 of E (inductive hypotesis, IH). Then:

Case $E = (E_0)$: By application of OP-BRACKET we reach configuration $[gs \,|\, (lms, vp, \underline{E_0}\ ts_0)]$ for which the theorem holds by the inductive hypothesis. Hence $ExpOk(E)$.

Case $E = I = E_0$: We get the following evolution:

$$\begin{array}{lcl}
[gs_0 \,|\, (lms_0, vp_0, \underline{I = E_0}\ ts_0)] & \xrightarrow{\text{OP-AssignExpr}} & [gs_0 \,|\, (lms_0, vp_0, \underline{E_0}\ \underline{I = \bullet}\ ts_0)] \\
 & \xrightarrow{IH}{}^{*} & [gs_{n'} \,|\, (lms_{n'}, vp_{n'}, \underline{\mathbf{v}}\ \underline{I = \bullet}\ ts_0)] \\
 & \xrightarrow{\text{OP-SubstE}} & [gs_{n'} \,|\, (lms_{n'}, vp_{n'}, \underline{I = \mathbf{v}}\ ts_0)]
\end{array}$$

The last configuration is one of the base cases for which the lemma holds. Hence $ExpOk(E)$.

Case $E = \mathbf{recv}(E_0)$: We get the following evolution:

$$\begin{array}{lcl}
[gs_0 \,|\, (lms_0, vp_0, \underline{\mathbf{recv}(E_0)}\ ts_0)] & \xrightarrow{\text{OP-RecvExpr}} & [gs_0 \,|\, (lms_0, vp_0, \underline{E_0}\ \underline{\mathbf{recv}(\bullet)}\ ts_0)] \\
 & \xrightarrow{IH}{}^{*} & [gs_{n'} \,|\, (lms_{n'}, vp_{n'}, \underline{\mathbf{v}}\ \underline{\mathbf{recv}(\bullet)}\ ts_0)] \\
 & \xrightarrow{\text{OP-SubstE}} & [gs_{n'} \,|\, (lms_{n'}, vp_{n'}, \underline{\mathbf{recv}(\mathbf{v})}\ ts_0)]
\end{array}$$

The last configuration is one of the base cases for which the lemma holds. Hence $ExpOk(E)$.

Case $E = \mathbf{measure}(E_0, \ldots, E_e)$: We get the following evolution:

$$\begin{array}{lcl}
\multicolumn{3}{l}{[gs_0 \,|\, (lms_0, vp_0, \underline{\mathbf{measure}(E_0, \ldots, E_e)}\ ts_0)]} \\
 & \xrightarrow{\text{OP-MeasureExpr}} & [gs_0 \,|\, (lms_0, vp_0, \underline{E_0}\ \underline{\mathbf{measure}(\bullet, \ldots, E_e)}\ ts_0)] \\
 & \xrightarrow{IH}{}^{*} & [gs_{n'} \,|\, (lms_{n'}, vp_{n'}, \underline{\mathbf{v}_0}\ \underline{\mathbf{measure}(\bullet, \ldots, E_e)}\ ts_0)] \\
 & \xrightarrow{\text{OP-SubstE}} & [gs_{n'} \,|\, (lms_{n'}, vp_{n'}, \underline{\mathbf{measure}(\mathbf{v}_0, \ldots, E_e)}\ ts_0)] \\
 & \vdots & \\
 & \xrightarrow{\text{OP-MeasureExpr}} & [gs_{n''} \,|\, (lms_{n''}, vp_{n''}, \underline{E_e}\ \underline{\mathbf{measure}(\mathbf{v}_0, \ldots, \bullet)}\ ts_0)] \\
 & \xrightarrow{IH}{}^{*} & [gs_{n'''} \,|\, (lms_{n'''}, vp_{n'''}, \underline{\mathbf{v}_e}\ \underline{\mathbf{measure}(\mathbf{v}_0, \ldots, \bullet)}\ ts_0)] \\
 & \xrightarrow{\text{OP-SubstE}} & [gs_{n'''} \,|\, (lms_{n'''}, vp_{n'''}, \underline{\mathbf{measure}(\mathbf{v}_0, \ldots, \mathbf{v}_e)}\ ts_0)]
\end{array}$$

The last configuration is one of the base cases for which the lemma holds. Hence $ExpOk(E)$.

□

Lemma 8.11. *Let B be a block-forming statement such that $BFOk_0(B)$. Then $BFOk(B)$ is satisfied.*

Proof. By induction on the structure of B. Let $C_0 = [gs_0 \,|\, (lms_0, vp_0, \underline{B}\ ts_0)] : \tau$ be any configuration. Base case:

Case $B = \mathbf{return\ v;}$: $BFOk(\mathbf{return\ v;})$ trivially.

Case $B = \mathbf{return}\ E\mathbf{;}$: We get the following evolution:

$$\begin{array}{lcl}
\multicolumn{3}{l}{[gs_a \,|\, (lms_a, vp_a, \underline{\mathbf{return}\ E\mathbf{;}}\ ts_0)]} \\
 & \xrightarrow{\text{OP-ReturnExpr}} & [gs_a, |, (lms_a, vp_a, \underline{E}\ \underline{\mathbf{return}\ \bullet\mathbf{;}}\ ts_0)] \\
 & \xrightarrow{\text{Lemma 8.10}}{}^{*} & [gs_{n'} \,|\, (lms_{n'}, vp_{n'}, \underline{RTErr})]\ \textit{ie.}\ \text{the lemma holds} \\
 & \textit{or} & [gs_{n'} \,|\, (lms_{n'}, vp_{n'}, \underline{\mathbf{v}}\ \underline{\mathbf{return}\ \bullet\mathbf{;}}\ ts_0)] \\
 & \xrightarrow{\text{OP-SubstS}} & [gs_{n'} \,|\, (lms_{n'}, vp_{n'}, \underline{\mathbf{return\ v;}}\ ts_0)]
\end{array}$$

In the last step we see that $BFOk(B)$.

Case $B = \mathbf{return;}$**:** $BFOk(\mathbf{return;})$ trivially.

Case $B = \mathbf{;}$**:** By application of OP-Skip we reach configuration $[gs_a \,|\, (lms_a, vp_a, ts_0)]$, hence $BFOk(B)$.

Case $B = \mathbf{v;}$**:** By application of OP-PromoForget we reach configuration $[gs_a \,|\, (lms_a, vp_a, ts_0)]$, hence $BFOk(B)$.

Case $B = PE\mathbf{;}$**:** We get the following evolution:

$$
\begin{array}{ll}
[gs_a \,|\, (lms_a, vp_a, \underline{PE\mathbf{;}}\ ts_0)] & \\
\xrightarrow{\text{OP-PromoExpr}} & [gs_a, |, (lms_a, vp_a, \underline{PE\ \bullet\mathbf{;}}\ ts_0)] \\
\xrightarrow{\text{Lemma 8.10}}{}^{*} & [gs_{n'} \,|\, (lms_{n'}, vp_{n'}, \underline{RTErr})] \ \textit{ie.}\ \text{the lemma holds} \\
\textit{or} & [gs_{n'} \,|\, (lms_{n'}, vp_{n'}, \underline{\mathbf{v}\ \bullet\mathbf{;}}\ ts_0)] \\
\xrightarrow{\text{OP-SubstS}} & [gs_{n'} \,|\, (lms_{n'}, vp_{n'}, \underline{\mathbf{v;}}\ ts_0)]
\end{array}
$$

The last configuration is exactly the previous case for which the lemma holds.

Case $B = \circ_{\mathbf{L}}$**:** By application of OP-BlockEnd we reach configuration $[gs_a \,|\, (lms_a, vp_a, ts_0)]$, hence $BFOk(B)$.

Case $B = \mathbf{fork}\ I(\mathbf{v}_0, \ldots, \mathbf{v}_e)\mathbf{;}$**:** By application of OP-DoFork we reach configuration $[gs_a \,|\, (lms_{0,a}, vp_{0,a}, ts_0) \parallel (lms_{1,a}, \blacksquare, I(\mathbf{v}_0, \ldots, \mathbf{v}_e)]$, hence $BFOk(B)$.

Case $B = \mathbf{fork}\ I(E_0, \ldots, E_e)\mathbf{;}$**:** We get the following evolution:

$$
\begin{array}{ll}
[gs_a \,|\, (lms_a, vp_a, \underline{\mathbf{fork}\ I(E_0, \ldots, E_e)\mathbf{;}}\ ts_0)] & \\
\xrightarrow{\text{OP-ForkExpr}} & [gs_a, |, (lms_a, vp_a, \underline{E_0}\ \underline{\mathbf{fork}\ I(\bullet, \ldots, E_e)\mathbf{;}}\ ts_0)] \\
\xrightarrow{\text{Lemma 8.10}}{}^{*} & [gs_{n'} \,|\, (lms_{n'}, vp_{n'}, \underline{RTErr})] \ \textit{ie.}\ \text{the lemma holds} \\
\textit{or} & [gs_{n'} \,|\, (lms_{n'}, vp_{n'}, \underline{\mathbf{v}_0}\ \underline{\mathbf{fork}\ I(\bullet, \ldots, E_e)\mathbf{;}}\ ts_0)] \\
\xrightarrow{\text{OP-SubstS}} & [gs_{n'} \,|\, (lms_{n'}, vp_{n'}, \underline{\mathbf{fork}\ I(\mathbf{v}_0, \ldots, E_e)\mathbf{;}}\ ts_0)] \\
\vdots & \\
\xrightarrow{\text{OP-ForkExpr}} & [gs_{a'}, |, (lms_{a'}, vp_{a'}, \underline{E_e}\ \underline{\mathbf{fork}\ I(\mathbf{v}_0, \ldots, \bullet)\mathbf{;}}\ ts_0)] \\
\xrightarrow{\text{Lemma 8.10}}{}^{*} & [gs_{n''} \,|\, (lms_{n''}, vp_{n''}, \underline{RTErr})] \ \textit{ie.}\ \text{the lemma holds} \\
\textit{or} & [gs_{n'''} \,|\, (lms_{n'''}, vp_{n'''}, \underline{\mathbf{v}_e}\ \underline{\mathbf{fork}\ I(\mathbf{v}_0, \ldots, \bullet)\mathbf{;}}\ ts_0)] \\
\xrightarrow{\text{OP-SubstS}} & [gs_{n'''} \,|\, (lms_{n'''}, vp_{n'''}, \underline{\mathbf{fork}\ I(\mathbf{v}_0, \ldots, \mathbf{v}_e)\mathbf{;}}\ ts_0)]
\end{array}
$$

The last configuration is exactly the previous case for which the lemma holds.

Case $B = \mathbf{send}(\mathbf{v}_c, \mathbf{v}_v)\mathbf{;}$**:** By application of OP-SendRecv we reach configuration $[gs_a \,|\, (lms_{0,a}, vp_{0,a}, ts_0)]$, hence $BFOk(B)$; by application of OP-SendUninit we reach configuration $[gs_a \,|\, (lms_{0,a}, vp_{0,a}, \mathbf{UV})]$, hence $BFOk(B)$.

Case $B = \mathbf{send}(E_c, E_v)\mathbf{;}$**:** We get the following evolution:

$$
\begin{array}{ll}
[gs_a \,|\, (lms_a, vp_a, \underline{\mathbf{send}(E_c, E_v)\mathbf{;}}\ ts_0)] & \\
\xrightarrow{\text{OP-SendExpr1}} & [gs_a, |, (lms_a, vp_a, \underline{E_c}\ \underline{\mathbf{send}(\bullet, E_v)\mathbf{;}}\ ts_0)] \\
\xrightarrow{\text{Lemma 8.10}}{}^{*} & [gs_{n'} \,|\, (lms_{n'}, vp_{n'}, \underline{RTErr})] \ \textit{ie.}\ \text{the lemma holds} \\
\textit{or} & [gs_{n'} \,|\, (lms_{n'}, vp_{n'}, \underline{\mathbf{v}_c}\ \underline{\mathbf{send}(\bullet, E_v)\mathbf{;}}\ ts_0)] \\
\xrightarrow{\text{OP-SubstS}} & [gs_{n'} \,|\, (lms_{n'}, vp_{n'}, \underline{\mathbf{send}(\mathbf{v}_c, E_v)\mathbf{;}}\ ts_0)] \\
\xrightarrow{\text{OP-SendExpr2}} & [gs_{a'}, |, (lms_{a'}, vp_{a'}, \underline{E_v}\ \underline{\mathbf{send}(\mathbf{v}_c, \bullet)\mathbf{;}}\ ts_0)] \\
\xrightarrow{\text{Lemma 8.10}}{}^{*} & [gs_{n''} \,|\, (lms_{n''}, vp_{n''}, \underline{RTErr})] \ \textit{ie.}\ \text{the lemma holds} \\
\textit{or} & [gs_{n''} \,|\, (lms_{n''}, vp_{n''}, \underline{\mathbf{v}_v}\ \underline{\mathbf{send}(\mathbf{v}_c, \bullet)\mathbf{;}}\ ts_0)] \\
\xrightarrow{\text{OP-SubstS}} & [gs_{n''} \,|\, (lms_{n''}, vp_{n''}, \underline{\mathbf{send}(\mathbf{v}_c, \mathbf{v}_v)\mathbf{;}}\ ts_0)]
\end{array}
$$

The last configuration is exactly the previous case for which the lemma holds.

Case $B = T\ \widetilde{I}$**;:** By possibly multiple application of OP-VarDeclMulti and OP-VarDecl we reach configuration $[gs_a \,|\, (lms_a, vp_a, ts_0)]$, hence $BFOk(B)$.

Case $B = \mathsf{channel}[T]\ I$ **withends**$[I,I]$**;:** By application of OP-VarDeclChE we reach configuration $[gs_a \,|\, (lms_a, vp_a, ts_0)]$, hence $BFOk(B)$.

Case $B = I$ **aliasfor** $[\widetilde{I}]$**;:** By application of OP-VarDeclAlF we reach configuration $[gs_a \,|\, (lms_a, vp_a, ts_0)]$, hence $BFOk(B)$.

Next we assume that the lemma holds for all substatements Be_0 of B (inductive hypotesis, IH). Then:

Case $B = Be_0\ Be_1\ \ldots\ Be_m$**:** We get the following evolution:

$$
\begin{array}{ll}
[gs_a \,|\, (lms_a, vp_a, \underline{Be_0\ Be_1\ \ldots\ Be_m}\ ts_0)] & \\
\xrightarrow{\text{OP-BlockHead}} & [gs_a \,|\, (lms_a, vp_a, \underline{Be_0}\ \underline{Be_1\ \ldots\ Be_m}\ ts_0)] \\
\xrightarrow{\text{IH}}{}^{*} & [gs_{n'} \,|\, (lms_{n'}, vp_{n'}, \underline{RTErr})]\ \textit{ie.}\ \text{the lemma holds} \\
\textit{or} & [gs_{n'} \,|\, (lms_{n'}, vp_{n'}, \underline{\textbf{return;}}\ \underline{Be_1\ \ldots\ Be_m}\ ts_0)]\ \textit{ie.}\ \text{the lemma holds} \\
\textit{or} & [gs_{n'} \,|\, (lms_{n'}, vp_{n'}, \underline{\textbf{return v;}}\ \underline{Be_1\ \ldots\ Be_m}\ ts_0)]\ \textit{ie.}\ \text{the lemma holds} \\
\textit{or} & \text{diverges}\ \textit{ie.}\ \text{the lemma holds} \\
\textit{or} & [gs_{n'} \,|\, (lms_{n'}, vp_{n'}, \underline{Be_1\ \ldots\ Be_m}\ ts_0)] \\
 & \vdots \\
\xrightarrow{\text{OP-BlockHead}} & [gs_{a'} \,|\, (lms_{a'}, vp_{a'}, \underline{Be_{m-1}}\ \underline{Be_m}\ ts_0)] \\
\xrightarrow{\text{IH}}{}^{*} & [gs_{n''} \,|\, (lms_{n''}, vp_{n''}, \underline{RTErr})]\ \textit{ie.}\ \text{the lemma holds} \\
\textit{or} & [gs_{n''} \,|\, (lms_{n''}, vp_{n''}, \underline{\textbf{return;}}\ \underline{Be_m}\ ts_0)]\ \textit{ie.}\ \text{the lemma holds} \\
\textit{or} & [gs_{n''} \,|\, (lms_{n''}, vp_{n''}, \underline{\textbf{return v;}}\ \underline{Be_m}\ ts_0)]\ \textit{ie.}\ \text{the lemma holds} \\
\textit{or} & \text{diverges}\ \textit{ie.}\ \text{the lemma holds} \\
\textit{or} & [gs_{n''} \,|\, (lms_{n''}, vp_{n''}, \underline{Be_m}\ ts_0)] \\
\xrightarrow{\text{IH}}{}^{*} & [gs_{n'''} \,|\, (lms_{n'''}, vp_{n'''}, \underline{RTErr})]\ \textit{ie.}\ \text{the lemma holds} \\
\textit{or} & [gs_{n'''} \,|\, (lms_{n'''}, vp_{n'''}, \underline{\textbf{return;}}\ ts_0)]\ \textit{ie.}\ \text{the lemma holds} \\
\textit{or} & [gs_{n'''} \,|\, (lms_{n'''}, vp_{n'''}, \underline{\textbf{return v;}}\ ts_0)]\ \textit{ie.}\ \text{the lemma holds} \\
\textit{or} & \text{diverges}\ \textit{ie.}\ \text{the lemma holds} \\
\textit{or} & [gs_{n'''} \,|\, (lms_{n'''}, vp_{n'''}, ts_0)]
\end{array}
$$

Therefore the lemma holds for this case.

Case $B =$ **if (v)** S_1 **else** S_2**:** Depending on **v**, the evolution continues by rule OP-IfTrue to configuration $[gs_{n'} \,|\, (lms_{n'}, vp_{n'}, \underline{S_1}\ ts_0)]$, or by rule OP-IfFalse to configuration $[gs_{n'} \,|\, (lms_{n'}, vp_{n'}, \underline{S_2}\ ts_0)]$. By IH, we assume that the lemma holds for both S_1 and S_2, *ie.* the lemma holds for this case.

Case $B =$ **if** (E) S_1 **else** S_2**:** We get the following evolution:

$$
\begin{array}{ll}
[gs_a \,|\, (lms_a, vp_a, \underline{\textbf{if}\ (E)\ S_1\ \textbf{else}\ S_2}\ ts_0)] & \\
\xrightarrow{\text{OP-IfExpr}} & [gs_a \,|\, (lms_a, vp_a, \underline{E}\ \underline{\textbf{if}\ (\bullet)\ S_1\ \textbf{else}\ S_2}\ ts_0)] \\
\xrightarrow{\text{Lemma 8.10}}{}^{*} & [gs_{n'} \,|\, (lms_{n'}, vp_{n'}, \underline{RTErr})]\ \textit{ie.}\ \text{the lemma holds} \\
\textit{or} & [gs_{n'} \,|\, (lms_{n'}, vp_{n'}, \underline{\mathbf{v}}\ \underline{\textbf{if}\ (\bullet)\ S_1\ \textbf{else}\ S_2}\ ts_0)] \\
\xrightarrow{\text{OP-SubstS}} & [gs_{n''} \,|\, (lms_{n''}, vp_{n''}, \underline{\textbf{if (v)}\ S_1\ \textbf{else}\ S_2}\ ts_0)]
\end{array}
$$

The last configuration is exactly the previous case for which the lemma holds.

Case $B = \{\ B\ \}$: We get the following evolution:

$$
\begin{array}{rl}
 & [gs_a \mid (lms_a, vp_a, \underline{\{\ B\ \}}\ ts_0)] \\
\xrightarrow{\text{OP-Block}} & [gs_a \mid (lms_a, vp_{a'}, \underline{B}\ \underline{\circ_{\mathbf{L}}}\ ts_0)] \\
\xrightarrow{\text{IH}}{}^{*} & [gs_{n'} \mid (lms_{n'}, vp_{n'}, \underline{RTErr})]\ \textit{ie.}\ \text{the lemma holds} \\
\textit{or} & [gs_{n'} \mid (lms_{n'}, vp_{n'}, \underline{\mathbf{return;}}\ ts_{n'}\ \underline{\circ_{\mathbf{L}}}\ ts_0)]\ \textit{ie.}\ \text{the lemma holds} \\
\textit{or} & [gs_{n'} \mid (lms_{n'}, vp_{n'}, \underline{\mathbf{return\ v;}}\ ts_{n'}\ \underline{\circ_{\mathbf{L}}}\ ts_0)]\ \textit{ie.}\ \text{the lemma holds} \\
\textit{or} & \text{diverges}\ \textit{ie.}\ \text{the lemma holds} \\
\textit{or} & [gs_{n'} \mid (lms_{n'}, vp_{n'}, \underline{\circ_{\mathbf{L}}}\ ts_0)] \\
\xrightarrow{\text{OP-BlockEnd}} & [gs_{n'} \mid (lms_{n'}, vp_a, ts_0)]
\end{array}
$$

Therefore the lemma holds for this case.

Case $B = \mathbf{while}\ (E)\ S$: We get the following evolution:

$$
\begin{array}{rl}
 & [gs_a \mid (lms_a, vp_a, \underline{\mathbf{while}\ (E)\ S}\ ts_0)] \\
\xrightarrow{\text{OP-While}} & [gs_a \mid (lms_a, vp_a, \underline{\mathbf{if}\ (E)\ \{S\ \mathbf{while}\ (E)\ S\}\ \mathbf{else\ ;}}\ ts_0)] \\
\xrightarrow{\text{see if case}}{}^{*} & [gs_{n'} \mid (lms_{n'}, vp_{n'}, \underline{RTErr})]\ \textit{ie.}\ \text{the lemma holds} \\
\textit{or} & \text{diverges}\ \textit{ie.}\ \text{the lemma holds} \\
\textit{or} & [gs_{n'} \mid (lms_{n'}, vp_{n'}, \underline{\mathbf{if}\ (\mathbf{v})\ \{S\ \mathbf{while}\ (E)\ S\}\ \mathbf{else\ ;}}\ ts_0)]
\end{array}
$$

Depending on $\mathbf{v}$, the evaluation continues either this way:

$$
\begin{array}{rl}
\xrightarrow{\text{OP-IfFalse}} & [gs_{n'} \mid (lms_{n'}, vp_{n'}, \underline{\mathbf{;}}\ ts_0)] \\
\xrightarrow{\text{OP-Skip}} & [gs_{n'} \mid (lms_{n'}, vp_{n'}, ts_0)]
\end{array}
$$

so the lemma holds for this case; or the evaluation continues as follows:

$$
\begin{array}{rl}
\xrightarrow{\text{OP-IfTrue}} & [gs_{n'} \mid (lms_{n'}, vp_{n'}, \underline{\{S\ \mathbf{while}\ (E)\ S\}}\ ts_0)] \\
\xrightarrow{\text{OP-Block}} & [gs_{n'} \mid (lms_{n'}, vp_{n'}, \underline{S\ \mathbf{while}\ (E)\ S}\ \underline{\circ_{\mathbf{L}}}\ ts_0)] \\
\xrightarrow{\text{OP-BlockHead}} & [gs_{n'} \mid (lms_{n'}, vp_{n'}, \underline{S}\ \underline{\mathbf{while}\ (E)\ S}\ \underline{\circ_{\mathbf{L}}}\ ts_0)] \\
\xrightarrow{\text{IH}}{}^{*} & [gs_{n''} \mid (lms_{n''}, vp_{n''}, \underline{RTErr})]\ \textit{ie.}\ \text{the lemma holds} \\
\textit{or} & [gs_{n''} \mid (lms_{n''}, vp_{n''}, \underline{\mathbf{return;}}\ ts_{n''}\ \underline{\circ_{\mathbf{L}}}\ ts_0)]\ \textit{ie.}\ \text{the lemma holds} \\
\textit{or} & [gs_{n''} \mid (lms_{n''}, vp_{n''}, \underline{\mathbf{return\ v;}}\ ts_{n''}\ \underline{\circ_{\mathbf{L}}}\ ts_0)]\ \textit{ie.}\ \text{the lemma holds} \\
\textit{or} & \text{diverges}\ \textit{ie.}\ \text{the lemma holds} \\
\textit{or} & [gs_{n''} \mid (lms_{n''}, vp_{n''}, \underline{\mathbf{while}\ (E)\ S}\ \underline{\circ_{\mathbf{L}}}\ ts_0)]
\end{array}
$$

Therefore it is possible that evaluation of **while** statement continues forever. This is the statement of the lemma, hence the lemma holds even for this case.

□

Lemma 8.12. *Let E be an expression such that $ExpOk_{i+1}(E)$. Then $ExpOk(E)$ is satisfied.*

Proof. The proof proceeds similarly to the proof of Lemma 8.10. We must only add two cases:

Case $E = I(\widetilde{\mathbf{v}})$: We know that the predicate $BFOk_i(M_B(I))$ is satisfied, hence by Lemmata 8.13 for $BFOk_i$ and 8.8, the predicate $StkRetOk(M_B(I))$ is satisfied (†). We get the following evolution:

$$
\begin{array}{rl}
 & [gs_a \mid (lms_a, vp_a, \underline{I(\widetilde{\mathbf{v}})}\ ts_a] \\
\xrightarrow{\text{OP-DoMethodCallCl}} & [gs_{n'} \mid (lms_{n'}, vp_{n'}, \underline{M_B(I)}\ \underline{\circ_{\mathbf{M}}}\ ts_0)] \\
\xrightarrow{(\dagger)}{}^{*} & [gs_{n''} \mid (lms_{n''}, vp_{n''}, \underline{RTErr}) \parallel \cdots \parallel (lms_{k,n''}, vp_{k,n''}, ts_{k,n''})] \\
\textit{or} & [gs_{n''} \mid (lms_{0,n''}, vp_{0,n''}, \underline{\mathbf{v}}\ ts_0) \parallel \cdots \parallel (lms_{k,n''}, vp_{k,n''}, ts_{k,n''})] \\
\textit{or} & \text{diverges}
\end{array}
$$

Therefore the lemma holds, $ExpOk(E)$.

Case $E = I(E_0, \ldots, E_e)$**:** By induction on the structure of E: Assume that the lemma holds for all $E_0, \ldots, E_e$ (inductive hypotesis, IH). We get the following evolution:

$$
\begin{array}{ll}
[gs_0 \,|\, (lms_0, vp_0, \underline{I(E_0, \ldots, E_e)}\ ts_0)] & \\
\xrightarrow{\text{OP-MethodCallExpr}} & [gs_0 \,|\, (lms_0, vp_0, \underline{E_0}\ \underline{I(\bullet, \ldots, E_e)}\ ts_0)] \\
\xrightarrow{IH}{}^{*} & [gs_{n'} \,|\, (lms_{n'}, vp_{n'}, \underline{\mathbf{v}_0}\ \underline{I(\bullet, \ldots, E_e)}\ ts_0)] \\
\xrightarrow{\text{OP-SubstE}} & [gs_{n'} \,|\, (lms_{n'}, vp_{n'}, \underline{I(\mathbf{v}_0, \ldots, E_e)}\ ts_0)] \\
\vdots & \\
\xrightarrow{\text{OP-MethodCallExpr}} & [gs_{n''} \,|\, (lms_{n''}, vp_{n''}, \underline{E_e}\ \underline{I(\mathbf{v}_0, \ldots, \bullet)}\ ts_0)] \\
\xrightarrow{IH}{}^{*} & [gs_{n'''} \,|\, (lms_{n'''}, vp_{n'''}, \underline{\mathbf{v}_e}\ \underline{I(\mathbf{v}_0, \ldots, \bullet)}\ ts_0)] \\
\xrightarrow{\text{OP-SubstE}} & [gs_{n'''} \,|\, (lms_{n'''}, vp_{n'''}, \underline{I(\mathbf{v}_0, \ldots, \mathbf{v}_e)}\ ts_0)]
\end{array}
$$

The last configuration is one of the base cases for which the lemma holds. Hence $ExpOk(E)$.

□

Lemma 8.13. *Let B be a block-forming statement such that $BFOk_{i+1}(B)$. Then $BFOk(B)$ is satisfied.*

Proof. The proof is nearly identical to the proof of Lemma 8.11. We only exchange all usages of Lemma 8.10 for evaluation of expression E by Lemma 8.12 for $ExpOk_i(E)$. □

Corollary 8.14. *For any at most countably derivable expression E, $ExpOk(E)$ is satisfied. For any at most countably derivable block-forming statement B, $BFOk(B)$ is satisfied.*

Informally, as the programs are always countably derivable, we have proved the following:

- Unless a runtime error occurs, the evaluation of an expression E never modifies term stack elements under the evaluated expression. If the evaluation does not diverge, the expression E always yields an internal value,
- Unless a runtime error occurs, the evaluation of a block-forming statement B never modifies term stack elements under the evaluated statement if it does not get to a **return** statement. In that case, it pops all the elements from the term stack up to the first occurence of the symbol $\circ_{\mathbf{M}}$.

Lemma 8.15. *Let B be a block-forming statement such that $RetOk(B)$. For any configuration $C_0 = [gs_0 \,|\, (lms_0, vp_0, \underline{B}\ ts_0)] : \tau$, the evaluation either diverges or reaches one of the following configurations:*

- $[gs_n \,|\, (lms_{0,n}, vp_{0,n}, \underline{\mathbf{return;}}\ ts_{0,n}\ ts_0) \parallel \cdots \parallel (lms_{k,n}, vp_{k,n}, ts_{k,n})], k \geq 0$, *$ts_{0,n}$ does not contain $\circ_{\mathbf{M}}$, or*
- $[gs_n \,|\, (lms_{0,n}, vp_{0,n}, \underline{\mathbf{return\ v;}}\ ts_{0,n}\ ts_0) \parallel \cdots \parallel (lms_{k,n}, vp_{k,n}, ts_{k,n})], k \geq 0$, *$ts_{0,n}$ does not contain $\circ_{\mathbf{M}}$, or*
- $[gs_n \,|\, (lms_{0,n}, vp_{0,n}, \underline{RTErr}) \parallel \cdots \parallel (lms_{k,n}, vp_{k,n}, ts_{k,n})], k \geq 0$.

Proof. By induction on the structure of B. Let $C_0 = [gs_0 \,|\, (lms_0, vp_0, \underline{B}\ ts_0)] : \tau$ be any configuration. Base case:

Case $B =$ **return v;:** The lemma holds trivially.

Case $B = \textbf{return } E\textbf{;}$**:** We get the following evolution:

$$
\begin{array}{ll}
[gs_a \,|\, (lms_a, vp_a, \underline{\textbf{return } E\textbf{;}}\ ts_0)] & \\
\xrightarrow{\text{OP-ReturnExpr}} & [gs_a, |, (lms_a, vp_a, \underline{E}\ \underline{\textbf{return } \bullet\textbf{;}}\ ts_0)] \\
\xrightarrow{ExpOk(E)}{}^{*} & [gs_{n'} \,|\, (lms_{n'}, vp_{n'}, \underline{RTErr})] \ \textit{ie.}\ \text{the lemma holds} \\
\textit{or} & \text{diverges } \textit{ie.}\ \text{the lemma holds} \\
\textit{or} & [gs_{n'} \,|\, (lms_{n'}, vp_{n'}, \underline{\textbf{v}}\ \underline{\textbf{return } \bullet\textbf{;}}\ ts_0)] \\
\xrightarrow{\text{OP-SubstS}} & [gs_{n'} \,|\, (lms_{n'}, vp_{n'}, \underline{\textbf{return v;}}\ ts_0)]
\end{array}
$$

In the last step we see we get requested configuration, hence the lemma holds.

Case $B = \textbf{return;}$**:** The lemma holds trivially.

Next we assume that the lemma holds for all substatements Be of B such that $RetOk(Be)$ (inductive hypotesis, IH). Then:

Case $B = Be_0\ Be_1 \ldots Be_m$**:** We know that there is at least one j such that $RetOk(Be_j)$. Let b be smallest such j. Moreover, $BFOk(Be_i)$ for $0 \leq i \leq m$. We get the following evolution:

$$
\begin{array}{ll}
[gs_a \,|\, (lms_a, vp_a, \underline{Be_0\ Be_1\ \ldots\ Be_m}\ ts_a)] & \\
\xrightarrow{\text{OP-BlockHead}} & [gs_a \,|\, (lms_a, vp_a, \underline{Be_0}\ \underline{Be_1\ \ldots\ Be_m}\ ts_0)] \\
\xrightarrow{BFOk(Be_0)}{}^{*} & [gs_{n'} \,|\, (lms_{n'}, vp_{n'}, \underline{RTErr})] \ \textit{ie.}\ \text{the lemma holds} \\
\textit{or} & [gs_{n'} \,|\, (lms_{n'}, vp_{n'}, \underline{\textbf{return;}}\ \underline{Be_1\ \ldots\ Be_m}\ ts_0)] \ \textit{ie.}\ \text{the lemma holds} \\
\textit{or} & [gs_{n'} \,|\, (lms_{n'}, vp_{n'}, \underline{\textbf{return v;}}\ \underline{Be_1\ \ldots\ Be_m}\ ts_0)] \ \textit{ie.}\ \text{the lemma holds} \\
\textit{or} & \text{diverges } \textit{ie.}\ \text{the lemma holds} \\
\textit{or} & [gs_{n'} \,|\, (lms_{n'}, vp_{n'}, \underline{Be_1\ \ldots\ Be_m}\ ts_0)] \\
 & \vdots \\
\xrightarrow{\text{OP-BlockHead}} & [gs_{a'} \,|\, (lms_{a'}, vp_{a'}, \underline{Be_b}\ ts_{0,a'}\ ts_0)]
\end{array}
$$

By IH, the lemma holds for the last configuration, therefore it holds for this case.

Case $B = \textbf{if (v)}\ S_1\ \textbf{else}\ S_2$**:** Depending on $\textbf{v}$, the evolution continues by rule OP-IfTrue to configuration $[gs_{n'} \,|\, (lms_{n'}, vp_{n'}, \underline{S_1}\ ts_0)]$, or by rule OP-IfFalse to configuration $[gs_{n'} \,|\, (lms_{n'}, vp_{n'}, \underline{S_2}\ ts_0)]$. From definition of $RetOk$, both $RetOk(S_1)$ and $RetOk(S_2)$ are satisfied. By IH, the lemma holds for this case.

Case $B = \textbf{if } (E)\ S_1\ \textbf{else}\ S_2$**:** We get the following evolution:

$$
\begin{array}{ll}
[gs_a \,|\, (lms_a, vp_a, \underline{\textbf{if } (E)\ S_1\ \textbf{else}\ S_2}\ ts_0)] & \\
\xrightarrow{\text{OP-IfExpr}} & [gs_a \,|\, (lms_a, vp_a, \underline{E}\ \underline{\textbf{if } (\bullet)\ S_1\ \textbf{else}\ S_2}\ ts_0)] \\
\xrightarrow{ExpOk(E)}{}^{*} & [gs_{n'} \,|\, (lms_{n'}, vp_{n'}, \underline{RTErr})] \ \textit{ie.}\ \text{the lemma holds} \\
\textit{or} & \text{diverges } \textit{ie.}\ \text{the lemma holds} \\
\textit{or} & [gs_{n'} \,|\, (lms_{n'}, vp_{n'}, \underline{\textbf{v}}\ \underline{\textbf{if } (\bullet)\ S_1\ \textbf{else}\ S_2}\ ts_0)] \\
\xrightarrow{\text{OP-SubstS}} & [gs_{n'} \,|\, (lms_{n'}, vp_{n'}, \underline{\textbf{if (v)}\ S_1\ \textbf{else}\ S_2}\ ts_0)]
\end{array}
$$

The last configuration is exactly the previous case for which the lemma holds.

Case $B = \textbf{\{}\ B\ \textbf{\}}$**:** We get the following evolution:

$$
[gs_a \,|\, (lms_a, vp_a, \underline{\textbf{\{}\ B\ \textbf{\}}}\ ts_0)] \xrightarrow{\text{OP-Block}} [gs_a \,|\, (lms_a, vp_{a'}, \underline{B}\ \underline{\circ_{\mathbf{L}}}\ ts_0)]
$$

By IH, the lemma holds for the last configuration, therefore it holds for this case.

□

Informally, for any block-forming statement B such that $RetOk(B)$, we have proved the following:

- Unless a runtime error occurs, the evaluation of B always leads to a **return** statement or diverges.

8.3.2 Type preservation lemmata

In the previous subsections, we have proved that for any well-typed configuration $C_0 : \tau$, there exists a configuration C_1 such that $C_0 \longrightarrow C_1$. The next step in proving type soundness is to prove that any such one-step evaluation preserves the type of the configuration. In other words, that the configuration C_1 is well-typed, $C_1 : \tau'$, and that $\tau' = \tau$ in the case of single process-evolution or $\tau' = \tau \times \theta$ in the case of the fork statement. In all cases, the type of the original processes is preserved.

Lemma 8.16 (Type preservation for $\longrightarrow_v$)**.** *Let* $C_0 = [gs_0 \,|\, (lms_0, vp_0, \underline{TE}\ ts_0)]$, $C_1 = \boxplus_{i=1}^{q} p_i \bullet C_{i,1}$ *where* $C_{i,1} = [gs_{i,1} \,|\, (lms_{i,1}, vp_{i,1}, \underline{TE_i}\ ts_{i,1})]$ *be two configurations such that* $C_0 \longrightarrow_v C_1$. *If* $C_0 : \tau$ *then* $C_{i,1} : \tau$ *for all* i.

Proof. From $C_0 : \tau$ we know:

$$\dfrac{\dfrac{\dfrac{(1)}{M; vpContext(vp_0) \vdash_T TE : \tau'}\ \text{T-rule} \qquad \dfrac{(2)}{M; \emptyset \vdash_C (lms_0, vp_0, ts_0) : \tau' \to \tau}}{M; \emptyset \vdash_C (lms_0, vp_0, \underline{TE}\ ts_0) : \mathsf{void} \to \tau}\ \text{TC-ExprClo}}{M; \emptyset \vdash [gs_0 \,|\, (lms_0, vp_0, \underline{TE}\ ts_0)] : \tau}\ \text{T-Config}$$

For each $i = 1, \ldots, n$, we want to prove $C_{i,1} : \tau$:

$$\dfrac{\dfrac{\dfrac{(\text{premise(T-rule)})}{M; vpContext(vp_{i,1}) \vdash_T TE_i : \tau'}\ \text{T-rule} \qquad \dfrac{(3)}{M; \emptyset \vdash_C (lms_{i,1}, vp_{i,1}, ts_{i,1}) : \tau' \to \tau}}{M; \emptyset \vdash_C (lms_{i,1}, vp_{i,1}, \underline{TE_i}\ ts_{i,1}) : \mathsf{void} \to \tau}\ \text{TC-ExprClo}}{M; \emptyset \vdash [gs_{i,1} \,|\, (lms_{i,1}, vp_{i,1}, \underline{TE_i}\ ts_{i,1})] : \tau}\ \text{T-Config}$$

As $\longrightarrow_v$ rules change the top stack element only, $ts_0 = ts_{i,1}$ for all i, therefore we must only prove $TE : \tau' \Rightarrow TE_i : \tau'$. We do this by case examination of relation $\longrightarrow_v$.

OP-AllocC: $TE : \mathsf{channel[T]}$ from T-Alloc. $TE_i = \langle\!\langle R, \mathbf{v} \rangle\!\rangle_{\mathsf{channel[T]}}$, thus $TE_i : \mathsf{channel[T]}$ from T-Value.

OP-AllocQ: $TE : \mathsf{Q_d}$ from T-Alloc. $TE_i = \langle\!\langle R, \mathbf{v} \rangle\!\rangle_{\mathsf{Q_d}}$, thus $TE_i : \mathsf{Q_d}$ from T-Value.

OP-AssignNewValue, OP-AssignQAValue, OP-AssignQValue, OP-AssignValue: $TE : \mathsf{T}$ from T-Assign. $TE_i = \langle\!\langle R, \mathbf{v} \rangle\!\rangle_{\mathsf{T}}$, thus $TE_i : \mathsf{T}$ from T-Value.

OP-DoMeasure: $TE : \mathsf{int}$ from T-Measurement. $TE_i = \langle\!\langle \mathsf{none}, \mathbf{v}_i \rangle\!\rangle_{\mathsf{int}}$, thus $TE_i : \mathsf{int}$ from T-Value.

OP-DoMethodCallQ: As all methods representing quantum operators are regarded as method with return type void, $TE : \mathsf{void}$ from T-MethodCall. $TE_i = \langle\!\langle \mathsf{none}, \bot \rangle\!\rangle_{\mathsf{void}}$, thus $TE_i : \mathsf{void}$ from T-Value.

OP-Var: $TE = x : \mathsf{T}$ from T-Var, hence $vpContext(vp_0)(x) = \mathsf{T}$. $TE_i = \langle\!\langle R, \mathbf{v}_x \rangle\!\rangle_{typeOf(x, vp_{i,1})}$, thus $TE_i : typeOf(x, vp_{i,1})$ from T-Value. We know that $vp_0 = vp_{i,1}$ as rule OP-Var does not modify variable properties. From definition of $vpContext$, $typeOf(x, vp_{i,1}) = \mathsf{T} \Leftrightarrow vpContext(vp_{i,1})(x) = \mathsf{T}$.

□

Lemma 8.17 (Type preservation for $\longrightarrow_e$)**.** *Let* $C_0 = [gs_0 \,|\, (lms_0, vp_0, \underline{TE_0}\ ts_0)]$, $C_1 = [gs_1 \,|\, (lms_1, vp_1, ts_1)]$ *be two configurations such that* $C_0 \longrightarrow_e C_1$. *If* $C_0 : \tau$ *then* $C_1 : \tau$.

Proof. We prove this lemma for each rule of relation $\longrightarrow_e$:

OP-MethodCallExpr $TE_0 = m(\widetilde{\mathbf{v}},E,\widetilde{E})$. From $C_0 : \tau$ we know:

$$\dfrac{\dfrac{(4) \qquad \dfrac{(5)}{M;\emptyset \vdash_C (lms_0, vp_0, ts_0) : \tau' \to \tau}}{M;\emptyset \vdash_C (lms_0, vp_0, \underline{m(\widetilde{\mathbf{v}},E,\widetilde{E})}\ ts_0) : \mathsf{void} \to \tau}\ \text{TC-ExprClo}}{M;\emptyset \vdash [gs_0 \,|\, (lms_0, vp_0, \underline{m(\widetilde{\mathbf{v}},E,\widetilde{E})}\ ts_0)] : \tau}\ \text{T-Config}$$

where (4) is:

$$\dfrac{(6) \qquad \dfrac{(7)}{M; vpContext(vp_0) \vdash_T E : \sigma} \qquad (8)}{M; vpContext(vp_0) \vdash_T m(\widetilde{\mathbf{v}},E,\widetilde{E}) : \tau'}\ \text{T-MethodCall}$$

We want to prove $C_1 : \tau$:

$$\dfrac{\dfrac{\dfrac{(9)}{M; vpContext(vp_0) \vdash_T E : \sigma} \qquad (10)}{M;\emptyset \vdash_C (lms_0, vp_0, \underline{E}\ \underline{m(\widetilde{\mathbf{v}}, \bullet ,\widetilde{E})}\ ts_0) : \mathsf{void} \to \tau}\ \text{TC-ExprClo}}{M;\emptyset \vdash [gs_0 \,|\, (lms_0, vp_0, \underline{E}\ \underline{m(\widetilde{\mathbf{v}}, \bullet ,\widetilde{E})}\ ts_0)] : \tau}\ \text{T-Config}$$

where (10) is (here we denote $vpc = vpContext(vp_0)$):

$$\dfrac{\dfrac{(11) \qquad \dfrac{}{M; vpc, \bullet : \sigma \vdash_T \bullet : \sigma} \qquad (12)}{M; vpc, \bullet : \sigma \vdash_T m(\widetilde{\mathbf{v}}, \bullet ,\widetilde{E}) : \tau'} \qquad \dfrac{(13)}{M;\emptyset \vdash_C (lms_0, vp_0, ts_0) : \tau' \to \tau}}{M;\emptyset \vdash_C (lms_0, vp_0, \underline{m(\widetilde{\mathbf{v}}, \bullet ,\widetilde{E})}\ ts_0) : \sigma \to \tau}\ \text{TC-ExprHole}$$

Realizing that (9) = (7), (11) = (6), (12) = (8), and (13) = (5) finishes the proof.

OP-AssignExpr, OP-IfExpr, OP-MeasureExpr, OP-PromoExpr, OP-RecvExpr, OP-SendExpr1, OP-SendExpr2 The proof is a straightforward alteration of the proof of OP-MethodCallExpr.

OP-ForkExpr $TE_0 = \mathbf{fork}\ m(\widetilde{\mathbf{v}},E,\widetilde{E})\mathbf{;}$. From $C_0 : \tau$ we know:

$$\dfrac{\dfrac{(14) \qquad \dfrac{(15)}{M;\emptyset \vdash_C (lms_0, vp_0, ts_0) : \mathsf{void} \to \tau}}{M;\emptyset \vdash_C (lms_0, vp_0, \underline{\mathbf{fork}\ m(\widetilde{\mathbf{v}},E,\widetilde{E})\mathbf{;}}\ ts_0) : \mathsf{void} \to \tau}\ \text{TC-StatClo}}{M;\emptyset \vdash [gs_0 \,|\, (lms_0, vp_0, \underline{\mathbf{fork}\ m(\widetilde{\mathbf{v}},E,\widetilde{E})\mathbf{;}}\ ts_0)] : \tau}\ \text{T-Config}$$

where (14) is (here we denote $vpc = vpContext(vp_0)$):

$$\dfrac{\dfrac{(16) \qquad \dfrac{(17)}{M; vpc \vdash_T E : \sigma} \qquad (18)}{M; vpc \vdash_T m(\widetilde{\mathbf{v}},E,\widetilde{E})\mathbf{;} : \tau'}\ \text{T-MethodCall} \qquad m \text{ is a classical method}}{M; vpc \vdash_T \mathbf{fork}\ m(\widetilde{\mathbf{v}},E,\widetilde{E})\mathbf{;} : \mathsf{void}}\ \text{T-Fork}$$

We want to prove $C_1 : \tau$:

$$\dfrac{\dfrac{\dfrac{(19)}{M; vpContext(vp_0) \vdash_T E : \sigma} \quad (20)}{M; \emptyset \vdash_C (lms_0, vp_0, \underline{E}\ \underline{\textbf{fork}\ m(\widetilde{\mathbf{v}}, \bullet, \widetilde{E})\textbf{;}}\ ts_0) : \mathsf{void} \to \tau}\ \text{TC-ExprClo}}{M; \emptyset \vdash [gs_0 \,|\, (lms_0, vp_0, \underline{E}\ \underline{\textbf{fork}\ m(\widetilde{\mathbf{v}}, \bullet, \widetilde{E})\textbf{;}}\ ts_0)] : \tau}\ \text{T-Config}$$

where (20) is (here we denote $vpc = vpContext(vp_0)$):

$$\dfrac{\dfrac{\dfrac{(21) \quad \dfrac{}{M; vpc, \bullet : \sigma \vdash_T \bullet : \sigma}\ (22)}{M; vpc, \bullet : \sigma \vdash_T m(\widetilde{\mathbf{v}}, \bullet, \widetilde{E})\textbf{;} : \tau'}}{M; vpc, \bullet : \sigma \vdash_T \textbf{fork}\ m(\widetilde{\mathbf{v}}, \bullet, \widetilde{E})\textbf{;} : \mathsf{void}} \quad \dfrac{(23)}{M; \emptyset \vdash_C (lms_0, vp_0, ts_0) : \mathsf{void} \to \tau}}{M; \emptyset \vdash_C (lms_0, vp_0, \underline{\textbf{fork}\ m(\widetilde{\mathbf{v}}, \bullet, \widetilde{E})\textbf{;}}\ ts_0) : \sigma \to \tau}\ \text{TC-StatHole}$$

Realizing that (19) = (17), (22) = (18), (21) = (16), and (23) = (15) finishes the proof.

OP-ReturnExpr $TE_0 = \textbf{return}\ E\textbf{;}$. We know that $vp_0 = vp_1 = [vp_G \circ_G vp]$, $ts_0 = \ldots \underline{\circ_{\mathbf{M}}}\ ts_r$. Denote by $\rho = typeOf(@retVal, vp_1)$. From $C_0 : \tau$ we know:

$$\dfrac{\dfrac{\dfrac{(24)}{M; vpContext(vp_0) \vdash_T E : \rho} \quad \dfrac{(25)}{M; \emptyset \vdash_C (lms_0, vp_G, ts_r) : \rho \to \tau}}{M; \emptyset \vdash_C (lms_0, vp_0, \underline{\textbf{return}\ E\textbf{;}} \ldots \underline{\circ_{\mathbf{M}}}\ ts_r) : \mathsf{void} \to \tau}\ \text{TC-RetExpr}}{M; \emptyset \vdash [gs_0 \,|\, (lms_0, vp_0, \underline{\textbf{return}\ E\textbf{;}} \ldots \underline{\circ_{\mathbf{M}}}\ ts_r)] : \tau}\ \text{T-Config}$$

We want to prove $C_1 : \tau$:

$$\dfrac{\dfrac{\dfrac{(26)}{M; vpContext(vp_0) \vdash_T E : \rho} \quad (27)}{M; \emptyset \vdash_C (lms_0, vp_0, \underline{E}\ \underline{\textbf{return}\ \bullet\textbf{;}} \ldots \underline{\circ_{\mathbf{M}}}\ ts_r) : \mathsf{void} \to \tau}\ \text{TC-ExprClo}}{M; \emptyset \vdash [gs_0 \,|\, (lms_0, vp_0, \underline{E}\ \underline{\textbf{return}\ \bullet\textbf{;}} \ldots \underline{\circ_{\mathbf{M}}}\ ts_r)] : \tau}\ \text{T-Config}$$

where (27) is:

$$\dfrac{\dfrac{(28)}{M; \emptyset \vdash_C (lms_0, vp_G, ts_r) : \rho \to \tau}}{M; \emptyset \vdash_C (lms_0, vp_0, \underline{\textbf{return}\ \bullet\textbf{;}} \ldots \underline{\circ_{\mathbf{M}}}\ ts_r) : \mathsf{void} \to \tau}\ \text{TC-RetHole}$$

Realizing that (26) = (24), and (28) = (25) finishes the proof.

OP-DoMethodCallCl $TE_0 = m(\widetilde{\mathbf{v}})$. From $C_0 : \tau$ we know:

$$\dfrac{\dfrac{\dfrac{(29) \quad \dfrac{(30)}{M; \emptyset \vdash_C (lms_0, vp_0, ts_0) : \tau' \to \tau}}{M; \emptyset \vdash_C (lms_0, vp_0, \underline{m(\mathbf{v}_0, \ldots, \mathbf{v}_n)}\ ts_0) : \mathsf{void} \to \tau}\ \text{TC-ExprClo}}{M; \emptyset \vdash [gs_0 \,|\, (lms_0, vp_0, \underline{m(\mathbf{v}_0, \ldots, \mathbf{v}_n)}\ ts_0)] : \tau}\ \text{T-Config}$$

where (29) is inferred by rule T-MethodCall:

$$\dfrac{M; vpContext(vp_0) \vdash_T M_T(m) = \sigma_0, \ldots, \sigma_n \to \tau' \quad (31) \qquad \dfrac{(32)}{M; vpContext(vp_0) \vdash_T \mathbf{v}_0 : \sigma_0} \quad \ldots}{M; vpContext(vp_0) \vdash_T m(\mathbf{v}_0, \ldots, \mathbf{v}_n) : \tau'}$$

We want to prove $C_1 : \tau$:

$$\dfrac{\dfrac{\dfrac{(33)}{M; \mathit{vpContext}([vp_0 \circ_G [\Box \circ_L vp'_M]]) \vdash_T M_B(m) : \mathsf{void}}\ \text{T-Block} \quad (34)}{M; \emptyset \vdash_C (lms_0, [vp_0 \circ_G [\Box \circ_L vp'_M]], \underline{M_B(m)}\ \underline{\circ_{\mathbf{M}}}\ ts_0) : \mathsf{void} \to \tau}\ \text{TC-StatClo}}{M; \emptyset \vdash [gs_0 \,|\, (lms_0, [vp_0 \circ_G [\Box \circ_L vp'_M]], \underline{M_B(m)}\ \underline{\circ_{\mathbf{M}}}\ ts_0)] : \tau}\ \text{T-Config}$$

where $M = (M_T, M_H, M_B)$ and (34) is:

$$\dfrac{\dfrac{(35)}{M; \emptyset \vdash_C (lms_0, vp_0, ts_0) : \mathit{typeOf}(@retVal, [vp_0 \circ_G [\Box \circ_L vp'_M]]) \to \tau}}{M; \emptyset \vdash_C (lms_0, [vp_0 \circ_G [\Box \circ_L vp'_M]], \underline{\circ_{\mathbf{M}}}\ ts_0) : \sigma \to \tau}\ \text{TC-RetImpl}$$

Method body is always a block. This justifies usage of rule T-Block. To see that the premises (35) and (30) specify the same proof, we recall the premise (31): the return type of method m is τ'. From the definition of rule OP-DoMethodCallCl we get that $\mathit{typeOf}(@retVal, [vp_0 \circ_G [\Box \circ_L vp'_M]]) = \tau'$. This finishes the proof.

OP-SubstE $TE_0 = \mathbf{v}$, moreover the symbol under the top element is Ec. From $C_0 : \tau$ we know:

$$\dfrac{\dfrac{\dfrac{}{M; \emptyset \vdash_T \langle\!\langle r, \mathbf{v} \rangle\!\rangle_{\mathsf{T}} : \mathsf{T}\ (36)}\ \text{T-Value} \quad (37)}{M; \emptyset \vdash_C (lms_0, vp_0, \underline{\mathbf{v}}\ \underline{Ec}\ ts_0) : \mathsf{void} \to \tau}\ \text{TC-ExprClo}}{M; \emptyset \vdash [gs_0 \,|\, (lms_0, vp_0, \underline{\mathbf{v}}\ \underline{Ec}\ ts_0)] : \tau}\ \text{T-Config}$$

where (37) is:

$$\dfrac{\dfrac{(38)}{M; \mathit{vpContext}(vp_0), \bullet : \mathsf{T} \vdash_T Ec : \tau'} \quad \dfrac{(39)}{M; \emptyset \vdash_C (lms_0, vp_0, ts_0) : \tau' \to \tau}}{M; \emptyset \vdash_C (lms_0, vp_0, \underline{Ec}\ ts_0) : \mathsf{T} \to \tau}\ \text{TC-ExprHole}$$

We want to prove $C_1 : \tau$:

$$\dfrac{\dfrac{\dfrac{(40)}{M; \mathit{vpContext}(vp_0) \vdash_T Ec[\mathbf{v}] : \tau'} \quad \dfrac{(41)}{M; \emptyset \vdash_C (lms_0, vp_0, ts_0) : \tau' \to \tau}}{M; \emptyset \vdash_C (lms_0, vp_0, \underline{Ec[\mathbf{v}]}\ ts_0) : \mathsf{void} \to \tau}\ \text{TC-ExprClo}}{M; \emptyset \vdash [gs_0 \,|\, (lms_0, vp_0, \underline{Ec[\mathbf{v}]}\ ts_0)] : \tau}\ \text{T-Config}$$

From the premise (36), we know $\mathbf{v} : \mathsf{T}$. The substitution of $\mathbf{v}$ in place of $\bullet$ is just an α-conversion that does not change type ($\bullet : \mathsf{T}$). Indeed, (41) = (39), and (40) = (38) what finishes the proof.

OP-SubstS The proof is a straightforward alteration of the proof of OP-SubstE.

□

Lemma 8.18 (Type preservation for OP-DoFork)**.** *Let* $C_0 = [gs_0 \,|\, (lms_0, vp_0, \underline{\mathbf{fork}\ m(\tilde{\mathbf{v}});}\ ts_0)]$, $C_1 = [gs_1 \,|\, (lms_{1,1}, vp_{1,1}, ts_{1,1}) \parallel (lms_{1,2}, vp_{1,2}, ts_{1,2})]$ *be two configurations such that* $C_0 \xrightarrow{\text{OP-DoFork}}_p C_1$. *If* $C_0 : \tau$ *then* $C_1 : \tau \times \theta$.

Proof. From $C_0 : \tau$ we know:

$$\dfrac{\dfrac{\dfrac{\dfrac{(42)}{M;\emptyset \vdash_T m(\widetilde{\mathbf{v}}) : \mathsf{T}}\ \text{T-MethodCall}}{M;\emptyset \vdash_T \mathbf{fork}\ m(\widetilde{\mathbf{v}})\mathbf{;} : \mathsf{void}}\ \text{T-Fork} \qquad \dfrac{(43)}{M;\emptyset \vdash_C (lms_0, vp_0, ts_0) : \mathsf{void} \to \tau}}{M;\emptyset \vdash_C (lms_0, vp_0, \underline{\mathbf{fork}\ m(\widetilde{\mathbf{v}})\mathbf{;}}\ ts_0) : \mathsf{void} \to \tau}\ \text{TC-StatClo}}{M;\emptyset \vdash [gs_0 \,|\, (lms_0, vp_0, \underline{\mathbf{fork}\ m(\widetilde{\mathbf{v}})\mathbf{;}}\ ts_0)] : \tau}\ \text{T-Config}$$

We want to prove $C_1 : \tau \times \theta$:

$$\dfrac{\dfrac{(44)}{M;\emptyset \vdash_C (lms_{1,1}, vp_0, ts_0) : \mathsf{void} \to \tau} \qquad (45)}{M;\emptyset \vdash [gs_0 \,|\, (lms_{1,1}, vp_0, ts_0) \parallel (lms_{1,2}, vp_{1,2}, ts_{1,2})] : \tau \times \theta}\ \text{T-Config}$$

where (45) is:

$$\dfrac{\dfrac{(46)}{M;\emptyset \vdash_T m(\widetilde{\mathbf{v}'}) : \mathsf{T}}\ \text{T-MethodCall} \qquad \dfrac{}{M;\emptyset \vdash_C (lms_{1,2}, \blacksquare, \varepsilon) : \mathsf{T} \to \theta}\ \text{TC-Empty}}{M;\emptyset \vdash_C (lms_{1,2}, \blacksquare, \underline{m(\widetilde{\mathbf{v}'})}) : \mathsf{void} \to \theta}\ \text{TC-ExprClo}$$

By inspecting the definition of rule OP-DoFork we see that transformation from $\widetilde{\mathbf{v}}$ to $\widetilde{\mathbf{v}'}$ preserves typing of individual values. Hence (46) = (42) and $\theta = \mathsf{T}$. Realizing that (44) = (43) finishes the proof.

□

Lemma 8.19 (Type preservation for OP-SendRecv)**.** *Let* $C_0 = [gs_0 \,|\, (lms_{0,1}, vp_{0,1}, \underline{\mathbf{send}(\mathbf{v}_s,\mathbf{v}_v)\mathbf{;}}\ ts_{0,1}) \parallel (lms_{0,2}, vp_{0,2}, \underline{\mathbf{recv}(\mathbf{v}_r)}\ ts_{0,2})]$, $C_1 = [gs_1 \,|\, (lms_{1,1}, vp_{1,1}, ts_{1,1}) \parallel (lms_{1,2}, vp_{1,2}, ts_{1,2})]$ *be two configurations such that* $C_0 \xrightarrow{\text{OP-SendRecv}}_p C_1$*. If* $C_0 : \tau \times \theta$ *then* $C_1 : \tau \times \theta$*.*

Proof. From $C_0 : \tau \times \theta$ by rule T-Config we know:

$$\dfrac{(47) \qquad \dfrac{(48)}{M;\emptyset \vdash_C (lms_{0,1}, vp_{0,1}, ts_{0,1}) : \mathsf{void} \to \tau}}{M;\emptyset \vdash_C (lms_{0,1}, vp_{0,1}, \underline{\mathbf{send}(\mathbf{v}_s, \mathbf{v}_v)\mathbf{;}}\ ts_{0,1}) : \mathsf{void} \to \tau}\ \text{TC-StatClo}$$

where (47) is (here we denote $vpc_{0,1} = vpContext(vp_{0,1})$):

$$\dfrac{\dfrac{}{M; vpc_{0,1} \vdash_T \mathbf{v}_s : \mathsf{channelEnd}[\tau_{\mathsf{v}}]}\ \text{T-Value} \qquad \dfrac{}{M; vpc_{0,1} \vdash_T \mathbf{v}_v : \tau_{\mathsf{v}}}\ \text{T-Value}}{M; vpc_{0,1} \vdash_T \mathbf{send}(\mathbf{v}_s, \mathbf{v}_v)\mathbf{;} : \mathsf{void}}\ \text{T-Send}$$

and (here we denote $vpc_{0,2} = vpContext(vp_{0,2})$):

$$\dfrac{\dfrac{\dfrac{}{M; vpc_{0,2} \vdash_T \mathbf{v}_r : \mathsf{channelEnd}[\theta']}\ \text{T-Value}}{M; vpc_{0,2} \vdash_T \mathbf{recv}(\mathbf{v}_r) : \theta'}\ \text{T-Recv} \qquad \dfrac{(49)}{M;\emptyset \vdash_C (lms_{0,1}, vp_{0,2}, ts_{0,2}) : \theta' \to \theta}}{M;\emptyset \vdash_C (lms_{0,2}, vp_{0,2}, \underline{\mathbf{recv}(\mathbf{v}_r)}\ ts_{0,2}) : \mathsf{void} \to \theta}\ \text{TC-ExprClo}$$

We want to prove $C_1 : \tau \times \theta$:

$$\dfrac{\dfrac{(50)}{M;\emptyset \vdash_C (lms_{1,1}, vp_{0,1}, ts_{0,1}) : \mathsf{void} \to \tau} \qquad (51)}{M;\emptyset \vdash [gs_0 \,|\, (lms_{1,1}, vp_{0,1}, ts_{0,1}) \parallel (lms_{1,2}, vp_{1,2}, \underline{\mathbf{v}_v}\ ts_{1,2})] : \tau \times \theta}\ \text{T-Config}$$

where (51) is (here we denote $vpc_{1,2} = vpContext(vp_{1,2})$):

$$\dfrac{\dfrac{}{M; vpc_{1,2} \vdash_T \mathbf{v}_v : \theta''}\ \text{T-Value} \qquad \dfrac{(52)}{M;\emptyset \vdash_C (lms_{1,2}, vp_{0,2}, ts_{0,2}) : \theta'' \to \theta}}{M;\emptyset \vdash_C (lms_{1,2}, vp_{0,2}, \underline{\mathbf{v}_v}\ ts_{0,2}) : \mathsf{void} \to \theta}\ \text{TC-ExprClo}$$

By inspecting the definition of rule OP-SendRecv we see that $\mathbf{v}_s$ and $\mathbf{v}_r$ are ends of the same channel, hence their types match. Therefore, $\theta' = \tau_{\mathsf{v}}$. Indeed, $\theta'' = \tau_{\mathsf{v}}$. As changes in local memory state does not change type, (52) = (49). Realizing that (50) = (48) finishes the proof.

□

Lemma 8.20 (Type preservation for $\longrightarrow_r$)**.** *Let* $C_0 = [gs_0 \,|\, (lms_0, vp_0, \underline{TE_0}\ ts_0)]$, $C_1 = [gs_1 \,|\, (lms_1, vp_1, ts_1)]$ *be two configurations such that* $C_0 \longrightarrow_r C_1$. *If* $C_0 : \tau$ *then* $C_1 : \tau$.

Proof. We prove this lemma for each rule of relation $\longrightarrow_r$:

OP-BlockHead $TE_0 = \overline{Be} = Be_0\ Be_1 \ldots Be_n$, $n \geq 1$. From $C_0 : \tau$ we know:

$$\dfrac{\dfrac{\dfrac{(53)}{M;\emptyset \vdash_C (lms_0, vp_0, \underline{Be_0}\ \underline{Be_1 \ldots Be_n}\ ts_0) : \mathsf{void} \to \tau}}{M;\emptyset \vdash_C (lms_0, vp_0, \underline{\overline{Be}}\ ts_0) : \mathsf{void} \to \tau}\ \text{TC-BlockHead}}{M;\emptyset \vdash [gs_0 \,|\, (lms_0, vp_0, \underline{\overline{Be}}\ ts_0)] : \tau}\ \text{T-Config}$$

We want to prove $C_1 : \tau$:

$$\dfrac{\dfrac{(54)}{M;\emptyset \vdash_C (lms_0, vp_0, \underline{Be_0}\ \underline{Be_1 \ldots Be_n}\ ts_0) : \mathsf{void} \to \tau}}{M;\emptyset \vdash [gs_0 \,|\, (lms_0, vp_0, \underline{Be_0}\ \underline{Be_1 \ldots Be_n}\ ts_0)] : \tau}\ \text{T-Config}$$

Indeed, (53) = (54).

OP-Bracket $TE_0 = (E)$. From $C_0 : \tau$ we know:

$$\dfrac{\dfrac{\dfrac{\dfrac{(55)}{M; vpContext(vp_0) \vdash_T E : \tau'}}{M; vpContext(vp_0) \vdash_T (E) : \tau'}\ \text{T-Bracket} \qquad \dfrac{(56)}{M;\emptyset \vdash_C (lms_0, vp_0, ts_0) : \tau' \to \tau}}{M;\emptyset \vdash_C (lms_0, vp_0, \underline{(E)}\ ts_0) : \mathsf{void} \to \tau}\ \text{TC-ExprClo}}{M;\emptyset \vdash [gs_0 \,|\, (lms_0, vp_0, \underline{(E)}\ ts_0)] : \tau}\ \text{T-Config}$$

We want to prove $C_1 : \tau$:

$$\dfrac{\dfrac{\dfrac{(57)}{M; vpContext(vp_0) \vdash_T E : \tau'} \qquad \dfrac{(58)}{M;\emptyset \vdash_C (lms_0, vp_0, ts_0) : \tau' \to \tau}}{M;\emptyset \vdash_C (lms_0, vp_0, \underline{E}\ ts_0) : \mathsf{void} \to \tau}\ \text{TC-ExprClo}}{M;\emptyset \vdash [gs_0 \,|\, (lms_0, vp_0, \underline{E}\ ts_0)] : \tau}\ \text{T-Config}$$

Indeed, (57) = (55), and (58) = (56).

OP-VarDeclMulti $TE_0 = \mathsf{T}\ I_0,I_1,\ldots,I_n;$. From $C_0 : \tau$ we know:

$$\dfrac{\dfrac{\dfrac{(59)}{M;\emptyset \vdash_C (lms_0, vp_0, \underline{\mathsf{T}\ I_0;}\ \underline{\mathsf{T}\ I_1,\ldots,I_n;}\ ts_0) : \mathsf{void} \to \tau}}{M;\emptyset \vdash_C (lms_0, vp_0, \underline{\mathsf{T}\ I_0,I_1,\ldots,I_n;}\ ts_0) : \mathsf{void} \to \tau}\ \text{TC-VarDeclMulti}}{M;\emptyset \vdash [gs_0 \,|\, (lms_0, vp_0, \underline{\mathsf{T}\ I_0,I_1,\ldots,I_n;}\ ts_0)] : \tau}\ \text{T-Config}$$

We want to prove $C_1 : \tau$:

$$\dfrac{\dfrac{(60)}{M;\emptyset \vdash_C (lms_0, vp_0, \underline{\mathsf{T}\ I_0;}\ \underline{\mathsf{T}\ I_1, \ldots, I_n;}\ ts_0) : \mathsf{void} \to \tau}}{M;\emptyset \vdash [gs_0 \,|\, (lms_0, vp_0, \underline{\mathsf{T}\ I_0;}\ \underline{\mathsf{T}\ I_1, \ldots, I_n;}\ ts_0)] : \tau}\ \text{T-Config}$$

Indeed, (60) = (59).

OP-While $TE_0 = \textbf{while}\ (E)\ S$. From $C_0 : \tau$ we know:

$$\dfrac{\dfrac{(61) \qquad \dfrac{(62)}{M;\emptyset \vdash_C (lms_0, vp_0, ts_0) : \mathsf{void} \to \tau}}{M;\emptyset \vdash_C (lms_0, vp_0, \underline{\textbf{while}\ (E)\ S}\ ts_0) : \mathsf{void} \to \tau}\ \text{TC-StatClo}}{M;\emptyset \vdash [gs_0 \,|\, (lms_0, vp_0, \underline{\textbf{while}\ (E)\ S}\ ts_0)] : \tau}\ \text{T-Config}$$

where (61) is (here we denote $vpc = vpContext(vp_0)$):

$$\dfrac{\dfrac{(63)}{M; vpc \vdash_T E : \mathsf{bool}} \qquad \dfrac{(64)}{M; vpc \vdash_T S : \mathsf{void}}}{M; vpc \vdash_T \textbf{while}\ (E)\ S : \mathsf{void}}\ \text{T-While}$$

We want to prove $C_1 : \tau$:

$$\dfrac{\dfrac{(65) \qquad \dfrac{(66)}{M;\emptyset \vdash_C (lms_0, vp_0, ts_0) : \tau' \to \tau}}{M;\emptyset \vdash_C (lms_0, vp_0, \underline{\textbf{if}\ (E)\ \{S\ \textbf{while}\ (E)\ S\}\ \textbf{else}\ ;}\ ts_0) : \mathsf{void} \to \tau}\ \text{TC-StatClo}}{M;\emptyset \vdash [gs_0 \,|\, (lms_0, vp_0, \underline{\textbf{if}\ (E)\ \{S\ \textbf{while}\ (E)\ S\}\ \textbf{else}\ ;}\ ts_0)] : \tau}\ \text{T-Config}$$

where (65) is (here we denote $vpc = vpContext(vp_0)$):

$$\dfrac{\dfrac{(67)}{M; vpc \vdash_T E : \mathsf{bool}} \qquad \dfrac{(68)}{M; vpc \vdash_T \{S\ \textbf{while}\ (E)\ S\} : \mathsf{void}} \qquad \dfrac{}{M; vpc \vdash_T \textbf{;} : \mathsf{void}}\ \text{T-Skip}}{M; vpc \vdash_T \textbf{if}\ (E)\ \{S\ \textbf{while}\ (E)\ S\}\ \textbf{else}\ \textbf{;} : \mathsf{void}}\ \text{T-If}$$

where (68) is by T-Block (here we again denote $vpc = vpContext(vp_0)$):

$$\dfrac{\dfrac{(69)}{M; vpc \vdash_T S : \mathsf{void}} \qquad \dfrac{\dfrac{(70)}{M; vpc \vdash_T E : \mathsf{bool}} \quad \dfrac{(71)}{M; vpc \vdash_T S : \mathsf{void}}}{\textbf{while}\ (E)\ S : \mathsf{void}}\ \text{T-While}}{M; vpc \vdash_T S\ \textbf{while}\ (E)\ S : \mathsf{void}}\ \text{T-BlockHead}$$

Indeed, (67) = (70) = (63), (69) = (71) = (64), and (66) = (62).

□

Lemma 8.21 (Type preservation for $\longrightarrow_s$)**.** *Let* $C_0 = [gs_0 \,|\, (lms_0, vp_0, \underline{TE_0}\ ts_0)]$, $C_1 = [gs_1 \,|\, (lms_1, vp_1, ts_1)]$ *be two configurations such that* $C_0 \longrightarrow_s C_1$. *If* $C_0 : \tau$ *then* $C_1 : \tau$.

Proof. We prove this lemma for each rule of relation $\longrightarrow_s$:

OP-Block $TE_0 = \{B\}$. From $C_0 : \tau$ we know (here we denote $vpc = vpContext([vp_G \circ_G vp])$):

$$\dfrac{\dfrac{\dfrac{\dfrac{(72)}{M; vpc \vdash_T B : \mathsf{void}}}{M; vpc \vdash_T \{B\} : \mathsf{void}}\ \text{T-Block} \qquad \dfrac{(73)}{M; \emptyset \vdash_C (lms_0, [vp_G \circ_G vp], ts_0) : \mathsf{void} \to \tau}}{M; \emptyset \vdash_C (lms_0, [vp_G \circ_G vp], \underline{\{B\}}\ ts_0) : \mathsf{void} \to \tau}\ \text{TC-StatClo}}{M; \emptyset \vdash [gs_0 \mid (lms_0, [vp_G \circ_G vp], \underline{\{B\}}\ ts_0)] : \tau}\ \text{T-Config}$$

We want to prove $C_1 : \tau$:

$$\dfrac{\dfrac{\dfrac{(74)}{M; vpContext([vp_G \circ_G [vp \circ_L \square]]) \vdash_T B : \mathsf{void}} \qquad (75)}{M; \emptyset \vdash_C (lms_0, [vp_G \circ_G [vp \circ_L \square]], \underline{B}\ \underline{\circ_{\mathbf{L}}}\ ts_0) : \mathsf{void} \to \tau}\ \text{TC-StatClo}}{M; \emptyset \vdash [gs_0 \mid (lms_0, [vp_G \circ_G [vp \circ_L \square]], \underline{B}\ \underline{\circ_{\mathbf{L}}}\ ts_0)] : \tau}\ \text{T-Config}$$

where (75) is:

$$\dfrac{\dfrac{(76)}{M; \emptyset \vdash_C (lms_0, [vp_G \circ_G vp], ts_0) : \mathsf{void} \to \tau}}{M; \emptyset \vdash_C (lms_0, [vp_G \circ_G [vp \circ_L \square]], \underline{\circ_{\mathbf{L}}}\ ts_0) : \mathsf{void} \to \tau}\ \text{TC-BlockEnd}$$

Indeed, $vpContext([vp_G \circ_G [vp \circ_L \square]]) = vpContext([vp_G \circ_G vp])$, hence the premises (74) and (72) are the same. Realizing that (76) = (73) finishes the proof.

OP-BlockEnd $TE_0 = \circ_{\mathbf{L}}$. From $C_0 : \tau$ we know:

$$\dfrac{\dfrac{\dfrac{(77)}{M; \emptyset \vdash_C (lms_0, [vp_G \circ_G vp], ts_0) : \mathsf{void} \to \tau}}{M; \emptyset \vdash_C (lms_0, [vp_G \circ_G [vp \circ_L vp_L]], \underline{\circ_{\mathbf{L}}}\ ts_0) : \mathsf{void} \to \tau}\ \text{TC-BlockEnd}}{M; \emptyset \vdash [gs_0 \mid (lms_0, [vp_G \circ_G [vp \circ_L vp_L]], \underline{\circ_{\mathbf{L}}}\ ts_0)] : \tau}\ \text{T-Config}$$

We want to prove $C_1 : \tau$:

$$\dfrac{\dfrac{(78)}{M; \emptyset \vdash_C (lms_0, [vp_G \circ_G vp], ts_0) : \mathsf{void} \to \tau}}{M; \emptyset \vdash [gs_0 \mid (lms_0, [vp_G \circ_G vp], ts_0)] : \tau}\ \text{T-Config}$$

Indeed, (78) = (77).

OP-IfFalse $TE_0 = \textbf{if}\ (\langle\!\langle r, \textbf{false} \rangle\!\rangle_{\mathsf{bool}})\ S_1\ \textbf{else}\ S_2$. From $C_0 : \tau$ we know:

$$\dfrac{\dfrac{(79) \qquad \dfrac{(80)}{M; \emptyset \vdash_C (lms_0, vp_0, ts_0) : \mathsf{void} \to \tau}}{M; \emptyset \vdash_C (lms_0, vp_0, \underline{\textbf{if}\ (\langle\!\langle r, \textbf{false} \rangle\!\rangle_{\mathsf{bool}})\ S_1\ \textbf{else}\ S_2}\ ts_0) : \mathsf{void} \to \tau}\ \text{TC-StatClo}}{M; \emptyset \vdash [gs_0 \mid (lms_0, vp_0, \underline{\textbf{if}\ (\langle\!\langle r, \textbf{false} \rangle\!\rangle_{\mathsf{bool}})\ S_1\ \textbf{else}\ S_2}\ ts_0)] : \tau}\ \text{T-Config}$$

where (79) is (here we denote $vpc = vpContext(vp_0)$):

$$\dfrac{\dfrac{(81)}{M; vpc \vdash_T \langle\!\langle r, \textbf{false} \rangle\!\rangle_{\mathsf{bool}} : \mathsf{bool}} \qquad \dfrac{(82)}{M; vpc \vdash_T S_1 : \mathsf{void}} \qquad \dfrac{(83)}{M; vpc \vdash_T S_2 : \mathsf{void}}}{M; vpc \vdash_T \textbf{if}\ (\langle\!\langle r, \textbf{false} \rangle\!\rangle_{\mathsf{bool}})\ S_1\ \textbf{else}\ S_2 : \mathsf{void}}\ \text{T-If}$$

We want to prove $C_1 : \tau$:

$$\dfrac{\dfrac{\dfrac{(84)}{M; vpContext(vp_0) \vdash_T \underline{S_2} : \mathsf{void}} \quad \dfrac{(85)}{M; \emptyset \vdash_C (lms_0, vp_0, ts_0) : \mathsf{void} \to \tau}}{M; \emptyset \vdash_C (lms_0, vp_0, \underline{S_2}\ ts_0) : \mathsf{void} \to \tau}\ \text{TC-StatClo}}{M; \emptyset \vdash [gs_0 \,|\, (lms_0, vp_0, \underline{S_2}\ ts_0)] : \tau}\ \text{T-Config}$$

Realizing that (84) = (83), and (85) = (80) finishes the proof.

OP-IfTrue The proof is a straightforward alteration of the proof of OP-IfFalse.

OP-PromoForget $TE_0 = \mathbf{v;}$. From $C_0 : \tau$ we know:

$$\dfrac{\dfrac{\dfrac{(86)}{M; vpContext(vp_0) \vdash_T \mathbf{v;} : \mathsf{void}} \quad \dfrac{(87)}{M; \emptyset \vdash_C (lms_0, vp_0, ts_0) : \mathsf{void} \to \tau}}{M; \emptyset \vdash_C (lms_0, vp_0, \underline{\mathbf{v;}}\ ts_0) : \mathsf{void} \to \tau}\ \text{TC-StatClo}}{M; \emptyset \vdash [gs_0 \,|\, (lms_0, vp_0, \underline{\mathbf{v;}}\ ts_0)] : \tau}\ \text{T-Config}$$

We want to prove $C_1 : \tau$:

$$\dfrac{\dfrac{(88)}{M; \emptyset \vdash_C (lms_0, vp_0, ts_0) : \mathsf{void} \to \tau}}{M; \emptyset \vdash [gs_0 \,|\, (lms_0, vp_0, ts_0)] : \tau}\ \text{T-Config}$$

Realizing that (88) = (87) finishes the proof.

OP-ReturnVoidImpl $TE_0 = \circ_{\mathbf{M}}$. From $C_0 : \tau$ we know:

$$\dfrac{\dfrac{\dfrac{(89)}{M; \emptyset \vdash_C (lms_0, vp_G, ts_0) : typeOf(@retVal, [vp_G \circ_G vp]) \to \tau}}{M; \emptyset \vdash_C (lms_0, [vp_G \circ_G vp], \underline{\circ_{\mathbf{M}}}\ ts_0) : \mathsf{void} \to \tau}\ \text{TC-RetImpl}}{M; \emptyset \vdash [gs_0 \,|\, (lms_0, [vp_G \circ_G vp], \underline{\circ_{\mathbf{M}}}\ ts_0)] : \tau}\ \text{T-Config}$$

We want to prove $C_1 : \tau$:

$$\dfrac{\dfrac{\dfrac{}{M; vpContext(vp_G) \vdash_T \langle\!\langle \mathsf{none}, \bot \rangle\!\rangle_{\mathsf{void}} : \mathsf{void}} \quad \dfrac{(90)}{M; \emptyset \vdash_C (lms_0, vp_G, ts_0) : \mathsf{void} \to \tau}}{M; \emptyset \vdash_C (lms_0, vp_G, \underline{\langle\!\langle \mathsf{none}, \bot \rangle\!\rangle_{\mathsf{void}}}\ ts_0) : \mathsf{void} \to \tau}\ \text{TC-ExprClo}}{M; \emptyset \vdash [gs_0 \,|\, (lms_0, vp_G, \underline{\langle\!\langle \mathsf{none}, \bot \rangle\!\rangle_{\mathsf{void}}}\ ts_0)] : \tau}\ \text{T-Config}$$

We must prove that $typeOf(@retVal, [vp_G \circ_G vp]) = \mathsf{void}$.

The only way to get $\circ_{\mathbf{M}}$ to the stack is by invoking method m using rule OP-DoMethodCallCl which also sets $typeOf(@retVal, [vp_G \circ_G vp])$ to the return type T of the method m. The method is well-typed.

Let T be not void. From the premises of typing rule T-Method, this means that $RetOk(M_B(m))$ is satisfied. However, by Lemma 8.15 we know that there must be a **return;** or **return v;** statement evaluated during evaluation of method m's body. This would replace $\circ_{\mathbf{M}}$. But this contradicts our assumption that $TE = \circ_{\mathbf{M}}$. Therefore $typeOf(@retVal, [vp_G \circ_G vp]) = \mathsf{void}$.

The proof of this case is now finished as (90) is equal to (89).

OP-Skip $TE_0 = \mathbf{;}$. From $C_0 : \tau$ we know:

$$\dfrac{\dfrac{\dfrac{}{M; vpContext(vp_0) \vdash_T \mathbf{;} : \mathsf{void}}\ \text{T-Skip} \quad \dfrac{(91)}{M; \emptyset \vdash_C (lms_0, vp_0, ts_0) : \mathsf{void} \to \tau}}{M; \emptyset \vdash_C (lms_0, vp_0, \underline{\mathbf{;}}\ ts_0) : \mathsf{void} \to \tau}\ \text{TC-StatClo}}{M; \emptyset \vdash [gs_0 \,|\, (lms_0, vp_0, \underline{\mathbf{;}}\ ts_0)] : \tau}\ \text{T-Config}$$

We want to prove $C_1 : \tau$:

$$\dfrac{\dfrac{(92)}{M;\emptyset \vdash_C (lms_0, vp_0, ts_0) : \mathsf{void} \to \tau}}{M;\emptyset \vdash [gs_0 \,|\, (lms_0, vp_0, ts_0)] : \tau}\ \text{T-Config}$$

Realizing that (92) = (91) finishes the proof.

OP-VarDecl $TE_0 = T\ x$**;**. From $C_0 : \tau$ we know:

$$\dfrac{\dfrac{\dfrac{(93)}{M;\vdash_C (lms_0, [vp_G \circ_G [vp \circ_L vp'_L]], ts_0) : \mathsf{void} \to \tau}}{M;\emptyset \vdash_C (lms_0, [vp_G \circ_G [vp \circ_L vp_L]], \underline{T\ x\mathbf{;}}\ ts_0) : \mathsf{void} \to \tau}\ \text{TC-VarDeclOne}}{M;\emptyset \vdash [gs_0 \,|\, (lms_0, [vp_G \circ_G [vp \circ_L vp_L]], \underline{T\ x\mathbf{;}}\ ts_0)] : \tau}\ \text{T-Config}$$

We want to prove $C_1 : \tau$:

$$\dfrac{\dfrac{(94)}{M;\emptyset \vdash_C (lms_0, [vp_G \circ_G [vp \circ_L vp''_L]], ts_0) : \mathsf{void} \to \tau}}{M;\emptyset \vdash [gs_0 \,|\, (lms_0, [vp_G \circ_G [vp \circ_L vp''_L]], ts_0)] : \tau}\ \text{T-Config}$$

where $vp''_L = vp_L[x \mapsto \mathsf{none}]_{+,var}[x \mapsto T]_{+,type}$.

By definition, $vp'_L = (f'_{var}, f'_{ch}, f'_{qa}, f'_{type})$ and $vp''_L = (f''_{var}, f''_{ch}, f''_{qa}, f''_{type})$. From definitions of OP-VarDecl and TC-VarDeclOne, we get $f'_{type} = f''_{type}$. This is enough to see that (94) is equal to (93) as the other parts of the variable properties tuples (*ie.* $f'_{var}, f'_{ch}, f'_{qa}, f''_{var}, f''_{ch}, f''_{qa}$) are never used in typing.

OP-VarDeclAlF, OP-VarDeclChE The proof is a straightforward alteration of the proof of OP-VarDecl.

□

Lemma 8.22 (Type preservation for $\longrightarrow_{ret}$)**.** *Let* $C_0 = [gs_0 \,|\, (lms_0, [vp_G \circ_G vp], TE_0\ \ldots \circ_{\mathbf{M}} ts_0)]$, $C_1 = [gs_1 \,|\, (lms_1, vp_G, ts_1)]$ *be two configurations such that* $C_0 \longrightarrow_{ret} C_1$. *If* $C_0 : \tau$ *then* $C_1 : \tau$.

Proof. We prove this lemma for each rule of relation $\longrightarrow_{ret}$:

OP-ReturnValue $TE_0 =$ **return v;**. From $C_0 : \tau$ we know (here we denote $vpc = vpContext([vp_G \circ_G vp])$ and $\sigma = typeOf(@retVal, [vp_G \circ_G vp])$):

$$\dfrac{\dfrac{\dfrac{}{M;vpc \vdash_T \mathbf{v} : \sigma}\ \text{T-Value} \quad \dfrac{(95)}{M;\emptyset \vdash_C (lms_0, vp_G, ts_0) : \sigma \to \tau}}{M;\emptyset \vdash_C (lms_0, [vp_G \circ_G vp], \underline{\mathbf{return\ v;}}\ \ldots \circ_{\mathbf{M}} ts_0) : \mathsf{void} \to \tau}\ \text{TC-RetExpr}}{M;\emptyset \vdash [gs_0 \,|\, (lms_0, [vp_G \circ_G vp], \underline{\mathbf{return\ v;}}\ \ldots \circ_{\mathbf{M}} ts_0)] : \tau}\ \text{T-Config}$$

We want to prove $C_1 : \tau$:

$$\dfrac{\dfrac{\dfrac{}{M;vpContext(vp_G) \vdash_T \mathbf{v} : \sigma}\ \text{T-Value} \quad \dfrac{(96)}{M;\emptyset \vdash_C (lms_0, vp_G, ts_0) : \sigma \to \tau}}{M;\emptyset \vdash_C (lms_0, vp_G, \underline{\mathbf{v}}\ ts_0) : \mathsf{void} \to \tau}\ \text{TC-ExprClo}}{M;\emptyset \vdash [gs_0 \,|\, (lms_0, vp_G, \underline{\mathbf{v}}\ ts_0)] : \tau}\ \text{T-Config}$$

The type of return value **v** does not depend on the context, therefore it is always σ. Realizing that (96) = (95) finishes the proof.

OP-ReturnVoid $TE_0 =$ **return;**. From $C_0 : \tau$ we know:

$$\dfrac{\dfrac{\dfrac{(97)}{M;\emptyset \vdash_C (lms_0, vp_G, ts_0) : \mathsf{void} \to \tau}}{M;\emptyset \vdash_C (lms_0, [vp_G \circ_G vp], \underline{\textbf{return;}}\ \ldots \circ_{\mathbf{M}} ts_0) : \mathsf{void} \to \tau}\ \text{TC-RetVoid}}{M;\emptyset \vdash [gs_0 \,|\, (lms_0, [vp_G \circ_G vp], \underline{\textbf{return;}}\ \ldots \circ_{\mathbf{M}} ts_0)] : \tau}\ \text{T-Config}$$

We want to prove $C_1 : \tau$ (here we denote $vpc = vpContext(vp_G)$):

$$\dfrac{\dfrac{\dfrac{}{M; vpc \vdash_T \langle\!\langle \mathsf{none}, \bot \rangle\!\rangle_{\mathsf{void}} : \mathsf{void}}\ \text{T-Value} \quad \dfrac{(98)}{M;\emptyset \vdash_C (lms_0, vp_G, ts_0) : \mathsf{void} \to \tau}}{M;\emptyset \vdash_C (lms_0, vp_G, \langle\!\langle \mathsf{none}, \bot \rangle\!\rangle_{\mathsf{void}}\ ts_0) : \mathsf{void} \to \tau}\ \text{TC-ExprClo}}{M;\emptyset \vdash [gs_0 \,|\, (lms_0, vp_G, \langle\!\langle \mathsf{none}, \bot \rangle\!\rangle_{\mathsf{void}}\ ts_0)] : \tau}\ \text{T-Config}$$

Realizing that (98) = (97) finishes the proof.

□

8.4 Type soundness

Theorem 8.23 (Type soundness for single-process programs)**.** *For any program which does not contain* **fork**, **send** *and* **recv** *constructs, if a configuration $C_0 : \tau$ evolves to a terminal configuration C_n, then either C_n is an errorneous configuration, or $C_n : \tau$.*

Proof. This is the corollary of progress and type preservation lemmata. □

Theorem 8.24 (Type soundness)**.** *For any program which does not contain* **send** *and* **recv** *constructs, if a configuration $C_0 = [gs_0 \,|\, ls_{0,1} \parallel \cdots \parallel ls_{0,m}], C_0 : \tau$ evolves to a terminal configuration C_n, then either C_n is an errorneous configuration, or $C_n : \tau \times \tau'$ for some τ'.*

Proof. This is the corollary of progress and type preservation lemmata. □

9 Conclusion and future work

We have described the LanQ imperative quantum programming language. This language can be used for implementation of both quantum algorithms and quantum protocols. We have formalized its syntax, both concrete and internal, typing, operational semantics, and proved type soundness of the non-communicating part of the language using proofs of standard lemmata in style of Wright and Felleisen [WF94] and [BPP03].

The language can be used for proving correctness of implemented quantum algorithms. If a correct pairing of sending and receiving processes is assured, it is also possible to prove correctness of quantum protocols.

By-reference usage of variables causes sometimes unwanted behaviour: it is possible to declare a program where a process sends a qubit and then it attempts to measure it. However, this is impossible as the qubit is then not owned by the process. Such situations are handled by runtime errors what also helps in debugging programs and protocols written in LanQ.

The simulator of LanQ is also being developed and it is publicly available from address `http://lanq.sourceforge.net/`.

An example of a truly random number generator and its evaluation sequence can be found in Appendix A.

Acknowledgements

I would like to express my thanks to Jan Bouda, Simon Gay, Philippe Jorrand, Rajagopal Nagarajan, Nick Papanikolaou, Igor Peterlík, and Libor Škarvada for their invaluable comments on LanQ and related discussions. I also wish to thank my supervisor, 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. 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.

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

[BPP03] Gavin Bierman, Matthew Parkinson, and Andrew Pitts. MJ: An imperative core calculus for Java and Java with effects. Technical Report 563, Cambridge University Computer Laboratory, April 2003.

[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 Figure 16.

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

Figure 16: Program example: Random number generator

Before the execution, we must specify the method typing context $M = (M_T, M_H, M_B)$. We have a program containing only a method *main*, hence the domain of the functions in M is $\{main\}$. M is specified as:

$$\begin{aligned} M_T(main) &= \mathsf{void} \longrightarrow \mathsf{int} \\ M_H(main) &= \mathsf{int}\ main() \\ M_B(main) &= \{\ \mathsf{qbit}\ q;\ q = \textbf{new}\ \mathsf{qbit}(); \\ &\qquad \textbf{return measure}\ (\langle\!\langle \mathsf{none}, StdBasis \rangle\!\rangle_{\mathsf{MeasurementBasis}}, q);\ \} \end{aligned}$$

The execution of the program is shown in Figure 17. For typographical reasons we have simplified variable properties tuple – we do not show states of f_{ch}, f_{qa} and f_{type} as only the f_{var} element of the tuple is needed in the example.

$$\mathbf{start} = [(((\mathbf{1}), []), []) \mid (([], [], [], []), \blacksquare, \underline{main()})]$$

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

$$[(((1), []), []) \mid (([], [], [], []), [\blacksquare \circ_G [\square \circ_L \Diamond]], \underline{\{\ \mathsf{qbit}\ q;\ q = \mathbf{new}\ \mathsf{qbit}();\ \mathbf{return\ measure}\ (\langle\!\langle \mathsf{none}, StdBasis \rangle\!\rangle_{\mathsf{MeasurementBasis}}, q);\ \}}\ \underline{\circ_{\mathbf{M}}})]$$

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

$$[(((1), []), []) \mid (([], [], [], []), [\blacksquare \circ_G [[\square \circ_L \Diamond] \circ_L \Diamond]],\ \underline{\mathsf{qbit}\ q;\ q = \mathbf{new}\ \mathsf{qbit}();\ \mathbf{return\ measure}\ (\langle\!\langle \mathsf{none}, StdBasis \rangle\!\rangle_{\mathsf{MeasurementBasis}}, q);}\ \underline{\circ_{\mathbf{L}}}\ \underline{\circ_{\mathbf{M}}})]$$

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

$$[(((1), []), []) \mid (([], [], [], []), [\blacksquare \circ_G [[\square \circ_L \Diamond] \circ_L \Diamond]],\ \underline{\mathsf{qbit}\ q;}\ \underline{q = \mathbf{new}\ \mathsf{qbit}();\ \mathbf{return\ measure}\ (\langle\!\langle \mathsf{none}, StdBasis \rangle\!\rangle_{\mathsf{MeasurementBasis}}, q);}\ \underline{\circ_{\mathbf{L}}}\ \underline{\circ_{\mathbf{M}}})]$$

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

$$[(((1), []), []) \mid (([], [], [], []), [\blacksquare \circ_G [[\square \circ_L \Diamond] \circ_L [q \mapsto \mathsf{none}]]], \underline{q = \mathbf{new}\ \mathsf{qbit}();\ \mathbf{return\ measure}\ (\langle\!\langle \mathsf{none}, StdBasis \rangle\!\rangle_{\mathsf{MeasurementBasis}}, q);}\ \underline{\circ_{\mathbf{L}}}\ \underline{\circ_{\mathbf{M}}})]$$

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

$$[(((1), []), []) \mid (([], [], [], []), [\blacksquare \circ_G [[\square \circ_L \Diamond] \circ_L [q \mapsto \mathsf{none}]]], \underline{q = \mathbf{new}\ \mathsf{qbit}();}\ \underline{\mathbf{return\ measure}\ (\langle\!\langle \mathsf{none}, StdBasis \rangle\!\rangle_{\mathsf{MeasurementBasis}}, q);}\ \underline{\circ_{\mathbf{L}}}\ \underline{\circ_{\mathbf{M}}})]$$

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

$$[(((1), []), []) \mid (([], [], [], []), [\blacksquare \circ_G [[\square \circ_L \Diamond] \circ_L [q \mapsto \mathsf{none}]]], \underline{q = \mathbf{new}\ \mathsf{qbit}()\ \bullet\ ;}\ \underline{\mathbf{return\ measure}\ (\langle\!\langle \mathsf{none}, StdBasis \rangle\!\rangle_{\mathsf{MeasurementBasis}}, q);}\ \underline{\circ_{\mathbf{L}}}\ \underline{\circ_{\mathbf{M}}})]$$

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

Figure 17: Program example: Random number generator execution (to be continued)

$\downarrow_e$ OP-AssignExpr

$$[(((1),[]),[]) \mid (([],[],[],[]),[\blacksquare \circ_G [[\square \circ_L \lozenge] \circ_L [q \mapsto \mathsf{none}]]], \underline{\textbf{new}\ \mathsf{qbit}()}\ \underline{q = \bullet}\ \underline{\bullet\ ;}\ \underline{\textbf{return measure}}\ (\langle\!\langle \mathsf{none}, \mathit{StdBasis} \rangle\!\rangle_{\mathsf{MeasurementBasis}}, q);\ \underline{\circ_{\mathbf{L}}}\ \underline{\circ_{\mathbf{M}}})]$$

$\downarrow_v$ OP-AllocQ

$$[((\tfrac{1}{2}\begin{pmatrix}1 & 0\\ 0 & 1\end{pmatrix}, [2]), []) \mid (([], [(\mathit{Quantum}, [1]) \mapsto (\mathit{GQuantum}, [1])], [], []), [\blacksquare \circ_G [[\square \circ_L \lozenge] \circ_L [q \mapsto \mathsf{none}]]],$$
$$\underline{\langle\!\langle (\mathit{Quantum}, [1]), (\mathit{GQuantum}, [1]) \rangle\!\rangle_{\mathsf{Q_2}}}\ \underline{q = \bullet}\ \underline{\bullet\ ;}\ \underline{\textbf{return measure}}\ (\langle\!\langle \mathsf{none}, \mathit{StdBasis} \rangle\!\rangle_{\mathsf{MeasurementBasis}}, q);\ \underline{\circ_{\mathbf{L}}}\ \underline{\circ_{\mathbf{M}}})]$$

$\downarrow_e$ OP-SubstE

$$[((\tfrac{1}{2}\begin{pmatrix}1 & 0\\ 0 & 1\end{pmatrix}, [2]), []) \mid (([], [(\mathit{Quantum}, [1]) \mapsto (\mathit{GQuantum}, [1])], [], []), [\blacksquare \circ_G [[\square \circ_L \lozenge] \circ_L [q \mapsto \mathsf{none}]]],$$
$$\underline{q = \langle\!\langle (\mathit{Quantum}, [1]), (\mathit{GQuantum}, [1]) \rangle\!\rangle_{\mathsf{Q_2}}}\ \underline{\bullet\ ;}\ \underline{\textbf{return measure}}\ (\langle\!\langle \mathsf{none}, \mathit{StdBasis} \rangle\!\rangle_{\mathsf{MeasurementBasis}}, q);\ \underline{\circ_{\mathbf{L}}}\ \underline{\circ_{\mathbf{M}}})]$$

$\downarrow_v$ OP-AssignQValue

$$[((\tfrac{1}{2}\begin{pmatrix}1 & 0\\ 0 & 1\end{pmatrix}, [2]), []) \mid (([], [(\mathit{Quantum}, [1]) \mapsto (\mathit{GQuantum}, [1])], [], []), [\blacksquare \circ_G [[\square \circ_L \lozenge] \circ_L [q \mapsto (\mathit{Quantum}, [1])]]],$$
$$\underline{\langle\!\langle (\mathit{Quantum}, [1]), (\mathit{GQuantum}, [1]) \rangle\!\rangle_{\mathsf{Q_2}}}\ \underline{\bullet\ ;}\ \underline{\textbf{return measure}}\ (\langle\!\langle \mathsf{none}, \mathit{StdBasis} \rangle\!\rangle_{\mathsf{MeasurementBasis}}, q);\ \underline{\circ_{\mathbf{L}}}\ \underline{\circ_{\mathbf{M}}})]$$

$\downarrow_e$ OP-SubstS

$$[((\tfrac{1}{2}\begin{pmatrix}1 & 0\\ 0 & 1\end{pmatrix}, [2]), []) \mid (([], [(\mathit{Quantum}, [1]) \mapsto (\mathit{GQuantum}, [1])], [], []), [\blacksquare \circ_G [[\square \circ_L \lozenge] \circ_L [q \mapsto (\mathit{Quantum}, [1])]]],$$
$$\underline{\langle\!\langle (\mathit{Quantum}, [1]), (\mathit{GQuantum}, [1]) \rangle\!\rangle_{\mathsf{Q_2}}\ ;}\ \underline{\textbf{return measure}}\ (\langle\!\langle \mathsf{none}, \mathit{StdBasis} \rangle\!\rangle_{\mathsf{MeasurementBasis}}, q);\ \underline{\circ_{\mathbf{L}}}\ \underline{\circ_{\mathbf{M}}})]$$

$\downarrow_s$ OP-PromoForget

Figure 17: Program example: Random number generator execution (continued) (to be continued)

$\downarrow_s$ OP-PromoForget

$$[((\tfrac{1}{2}\begin{pmatrix}1 & 0\\ 0 & 1\end{pmatrix}, [2]), []) \mid (([], [(Quantum, [1]) \mapsto (GQuantum, [1])], [], []), [\blacksquare \circ_G [[\square \circ_L \lozenge] \circ_L [q \mapsto (Quantum, [1])]]], \underline{\textbf{return measure}}\ (\langle\!\langle \mathsf{none}, StdBasis \rangle\!\rangle_{\mathsf{MeasurementBasis}}, q);\ \underline{\circ_{\mathbf{L}}}\ \underline{\circ_{\mathbf{M}}})]$$

$\downarrow_e$ OP-ReturnExpr

$$[((\tfrac{1}{2}\begin{pmatrix}1 & 0\\ 0 & 1\end{pmatrix}, [2]), []) \mid (([], [(Quantum, [1]) \mapsto (GQuantum, [1])], [], []), [\blacksquare \circ_G [[\square \circ_L \lozenge] \circ_L [q \mapsto (Quantum, [1])]]], \underline{\textbf{measure}}(\langle\!\langle \mathsf{none}, StdBasis \rangle\!\rangle_{\mathsf{MeasurementBasis}}, q)\ \underline{\textbf{return}}\ \bullet;\ \underline{\circ_{\mathbf{L}}}\ \underline{\circ_{\mathbf{M}}})]$$

$\downarrow_e$ OP-MeasureExpr

$$[((\tfrac{1}{2}\begin{pmatrix}1 & 0\\ 0 & 1\end{pmatrix}, [2]), []) \mid (([], [(Quantum, [1]) \mapsto (GQuantum, [1])], [], []), [\blacksquare \circ_G [[\square \circ_L \lozenge] \circ_L [q \mapsto (Quantum, [1])]]], \underline{q}\ \underline{\textbf{measure}}(\langle\!\langle \mathsf{none}, StdBasis \rangle\!\rangle_{\mathsf{MeasurementBasis}}, \bullet)\ \underline{\textbf{return}}\ \bullet;\ \underline{\circ_{\mathbf{L}}}\ \underline{\circ_{\mathbf{M}}})]$$

$\downarrow_v$ OP-Var

$$[((\tfrac{1}{2}\begin{pmatrix}1 & 0\\ 0 & 1\end{pmatrix}, [2]), []) \mid (([], [(Quantum, [1]) \mapsto (GQuantum, [1])], [], []), [\blacksquare \circ_G [[\square \circ_L \lozenge] \circ_L [q \mapsto (Quantum, [1])]]], \underline{\langle\!\langle (Quantum, [1]), (GQuantum, [1]) \rangle\!\rangle_{\mathsf{Q_2}}}\ \underline{\textbf{measure}}(\langle\!\langle \mathsf{none}, StdBasis \rangle\!\rangle_{\mathsf{MeasurementBasis}}, \bullet)\ \underline{\textbf{return}}\ \bullet;\ \underline{\circ_{\mathbf{L}}}\ \underline{\circ_{\mathbf{M}}})]$$

$\downarrow_e$ OP-SubstE

$$[((\tfrac{1}{2}\begin{pmatrix}1 & 0\\ 0 & 1\end{pmatrix}, [2]), []) \mid (([], [(Quantum, [1]) \mapsto (GQuantum, [1])], [], []), [\blacksquare \circ_G [[\square \circ_L \lozenge] \circ_L [q \mapsto (Quantum, [1])]]], \underline{\textbf{measure}}(\langle\!\langle \mathsf{none}, StdBasis \rangle\!\rangle_{\mathsf{MeasurementBasis}}, \underline{\langle\!\langle (Quantum, [1]), (GQuantum, [1]) \rangle\!\rangle_{\mathsf{Q_2}}})\ \underline{\textbf{return}}\ \bullet;\ \underline{\circ_{\mathbf{L}}}\ \underline{\circ_{\mathbf{M}}})]$$

$\downarrow_v$ OP-DoMeasure

Figure 17: Program example: Random number generator execution (continued) (to be continued)

$$\downarrow_v \text{ OP-DoMeasure}$$

$$0.5 \bullet [((\begin{pmatrix}1 & 0\\ 0 & 0\end{pmatrix}, [2]), []) \mid (([], [(Quantum, [1]) \mapsto (GQuantum, [1])], [], []), [\blacksquare \circ_G [[\square \circ_L \lozenge] \circ_L [q \mapsto (Quantum, [1])]]], \underline{\langle\!\langle \mathsf{none}, 0 \rangle\!\rangle_{\mathsf{int}}}\ \underline{\mathbf{return}\ \bullet;}\ \underline{\circ_{\mathbf{L}}}\ \underline{\circ_{\mathbf{M}}})]$$

$$\boxplus\ 0.5 \bullet [((\begin{pmatrix}0 & 0\\ 0 & 1\end{pmatrix}, [2]), []) \mid (([], [(Quantum, [1]) \mapsto (GQuantum, [1])], [], []), [\blacksquare \circ_G [[\square \circ_L \lozenge] \circ_L [q \mapsto (Quantum, [1])]]], \underline{\langle\!\langle \mathsf{none}, 1 \rangle\!\rangle_{\mathsf{int}}}\ \underline{\mathbf{return}\ \bullet;}\ \underline{\circ_{\mathbf{L}}}\ \underline{\circ_{\mathbf{M}}})]$$

$$\overset{0.5}{\swarrow} \quad \text{NP-ProbEvol} \quad \overset{0.5}{\searrow}$$

Evolution of measurement branch 0 | Evolution of measurement branch 1

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.

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

$$[((\begin{pmatrix}1 & 0\\ 0 & 0\end{pmatrix}, [2]), []) \mid (([], [(Quantum, [1]) \mapsto (GQuantum, [1])], [], []), [\blacksquare \circ_G [[\square \circ_L \lozenge] \circ_L [q \mapsto (Quantum, [1])]]], \underline{\langle\!\langle \mathsf{none}, 0 \rangle\!\rangle_{\mathsf{int}}}\ \underline{\mathbf{return}\ \bullet;}\ \underline{\circ_{\mathbf{L}}}\ \underline{\circ_{\mathbf{M}}})]$$

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

$$[((\begin{pmatrix}1 & 0\\ 0 & 0\end{pmatrix}, [2]), []) \mid (([], [(Quantum, [1]) \mapsto (GQuantum, [1])], [], []), [\blacksquare \circ_G [[\square \circ_L \lozenge] \circ_L [q \mapsto (Quantum, [1])]]], \underline{\mathbf{return}\ \langle\!\langle \mathsf{none}, 0 \rangle\!\rangle_{\mathsf{int}};}\ \underline{\circ_{\mathbf{L}}}\ \underline{\circ_{\mathbf{M}}})]$$

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

$$[((\begin{pmatrix}1 & 0\\ 0 & 0\end{pmatrix}, [2]), []) \mid (([], [(Quantum, [1]) \mapsto (GQuantum, [1])], [], []), \blacksquare, \underline{\langle\!\langle \mathsf{none}, 0 \rangle\!\rangle_{\mathsf{int}}})]$$

Figure 17: Program example: Random number generator execution (continued)