Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 210 (2008) 85–105

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Quantum Data and Control Made Easier

Michael Lampis[1,2] Kyriakos G. Ginis[3]
Michalis A. Papakyriakou[4] Nikolaos S. Papaspyrou[5]

School of Electrical and Computer Engineering
National Technical University of Athens
Athens, Greece.

Abstract

In this paper we define nQML, a functional quantum programming language that follows the “quantum data and control” paradigm. In comparison to Altenkirch and Grattage’s QML, the control constructs of nQML are simpler and can implement quantum algorithms more directly and naturally. We avoid the unnecessary complexities of a linear type system by using types that carry the address of qubits in the quantum state. We provide a denotational semantics over density matrices and unitary transformations, inspired by Selinger’s semantics for QPL. Our semantics leads naturally to an interpreter for nQML, written in Haskell. We also explore the extension of nQML with polymorphic higher-order functions.

Keywords: Functional quantum programming language, type system, denotational semantics.

1 Introduction

In the years following the discovery of Shor’s factoring algorithm [11] and Grover’s algorithm for database search [5] the field of quantum computations has attracted much scientific interest. Unlike classical algorithms, quantum algorithms are almost invariably studied at a low level, involving quantum circuits and their properties. The fact that reasoning about quantum circuits is no easier than reasoning about their classical counterparts has given rise to quantum programming languages, that is, languages that allow programmers to implement quantum algorithms and make use of the added power of the quantum computational model, while respecting its special restrictions. In this paper we present such a language named nQML.

[1] Research supported in part by the European Social Fund (75%) and the Greek Ministry of Education (25%) through grant “Pythagoras” of the Operational Programme on Education and Initial Vocational Training.

[2] Email: mlampis@cs.ntua.gr

[3] Email: kyrginis@softlab.ntua.gr

[4] Email: mpapakyr@softlab.ntua.gr

[5] Email: nickie@softlab.ntua.gr

doi:10.1016/j.entcs.2008.04.020

Our main focus in the design of nQML is to give programmers sufficient expressive power to implement quantum algorithms easily, while preventing them from breaking the rules of quantum computation. nQML is a high-level functional language based on the concept of "quantum data and control". It includes constructs which allow any unitary transformation to be expressed as a program in nQML quite naturally, more or less using the same notation that is used by the designers of quantum algorithms. It also permits quantum measurements to be carried out at any point during the execution of a program.

The relative ease of use comes at the cost of putting aside a number of important practical issues, such as the existence of imperfect quantum hardware, the need for quantum error correction and the fact that every quantum program will eventually have to be implemented as a quantum circuit using only a finite set of quantum gates and, therefore, some of the unitary transformations that nQML allows will have to be approximated. Similar problems were a source of concern for the founders of the classical programming model many decades ago. Fortunately they have been resolved and their solutions have been abstracted in such a way that people who use modern high-level programming languages do not need to know anything about them. We believe that the same can and must be done for the quantum programming languages of the future and adopt the approach that such issues should be tackled not by the designer and users of a quantum programming language, but by the architect of a quantum computer, the designer of its operating system and, to a lesser extent, the designer of the compiler.

nQML admits a simple type system and denotational semantics. By simple, we mean that both use structures and techniques that are typical in the study of classical programming languages of similar size and complexity. They should therefore be easily accessible to readers with a basic knowledge of programming language semantics and an elementary understanding of the quantum computation model. The main novelty of nQML's type system is that the type of a quantum expression conveys information which reveals the exact qubits of the quantum state in which the expression's value resides. Qubit aliasing is allowed in such a way that the "no cloning" and "no dropping" principles are not violated. Programmers have the look-and-feel of a classical programming language, without linearity restrictions.

The denotational semantics of nQML is based on the use of density matrices to describe quantum states. The meaning of a well-typed nQML program is a function from density matrices to density matrices and describes the program's effect on an arbitrary quantum input state. Well-typed programs which conduct no measurements[6] are also assigned a meaning in the form of a unitary matrix which describes the transformation they perform on the quantum state. The execution of a nQML program can be seen as a sequence of steps which affect the quantum state either by allocating new qubits, by applying unitary transformations to existing qubits or by measuring existing qubits. Our semantics leads to a straightforward implementation for nQML in the form of an interpreter written in Haskell.[7] The interpreter,

[6] In the sequel, such programs will be called "pure" quantum programs, for short.

[7] The source code of the interpreter is available from ftp://ftp.softlab.ntua.gr/pub/users/nickie/

quite obviously, simulates quantum computations in a classical computer and takes exponential time.

The rest of the paper is structured as follows. Section 2 discusses related work. In Section 3 we describe the syntax and semantics of nQML and section 4 contains a number of examples. In section 5 we discuss how to extend nQML with a polymorphic type system supporting higher-order functions. Section 6 concludes with our final remarks. The appendix contains the complete formal definition of nQML.

2 Related work

Starting with Knill's conventions for quantum pseudocode [6], several quantum programming languages have been proposed and an excellent survey of the emerging field can be found in [2]. Among the most notable are Ömer's QCL, an imperative language with quantum primitives and automatic quantum scratch space management [7], and Sanders and Zuliani's qGCL, an extension of Dijkstra's guarded command language [8]. Moreover, van Tonder has proposed a λ-calculus for higher-order quantum programs without measurements [12]. It is not clear however how this calculus corresponds to lower-level descriptions of quantum computations, such as quantum circuits.

Selinger's QPL is a language following the paradigm "quantum data, classical control" [9]. It is functional in nature, although from a programmer's point of view it looks more imperative than functional. QPL allows the programmer to access both classical and quantum memory and includes high-level features such as loops and recursion. Program control in QPL is strictly classical and quantum branching can only be implemented indirectly with appropriate unitary transformations. The denotational semantics of QPL is given in the form of superoperators on density matrices. A higher-order extension of QPL in the form of a quantum lambda calculus has also been proposed by Selinger and Valiron [10].

On the other hand, Altenkirch and Grattage's QML is a functional language that follows the paradigm "quantum data and control" [1,3,4]. QML comes with a linear type system prohibiting implicit weakening, which would lead to implicit measurements and quantum collapse. Variables in QML correspond to wires in the produced quantum circuit and thus have to be shared implicitly when they are used in several places in a program so as not to break the "no cloning" rule. The sharing of wires is also monitored by the linear type system. The semantics of QML assigns to every well-typed program a quantum circuit. QML's **if**$^\circ$ operator implements the notion of quantum control and is the only available means of performing unitary transformation. The two branches of an **if**$^\circ$ must be "orthogonal" quantum expressions, in order to preserve the reversibility of pure quantum computations.

The nature of our nQML is inspired from QML, the main addition being the quantum transformation construct $|e\rangle \rightarrow x, x'. c$ which will be described in section 3. Its type system, although not linear, is an adaptation of Altenkirch and

`papers/nqml06.code.tar.gz`.

Grattage's type system. The semantics of nQML, however, is very much in the spirit of Selinger's denotational semantics for QPL.

3 The language nQML

The complete syntax of nQML is given in the following grammar. It is assumed that x is a variable identifier and λ is a complex constant. The grammar defines two syntactic classes. Quantum expressions are denoted by e; they represent quantum programs and their syntax is similar to that of QML. Classical expressions are denoted by c; they are only needed in the quantum transformation construct $|e\rangle \rightarrow x, x'. \, c$ and they can represent two types of information: a structure of classical bits or a complex number.

$$
\begin{array}{rcl}
e & ::= & x \mid \{\, (\lambda)\, \mathbf{qfalse} + (\lambda')\, \mathbf{qtrue} \,\} \mid \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \\
 & \mid & (e_1, e_2) \mid \mathbf{let}\ (x_1, x_2) = e_1\ \mathbf{in}\ e_2 \\
 & \mid & \mathbf{if}\ e\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 \mid \mathbf{ifm}\ e\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 \mid |e\rangle \rightarrow x, x'. \, c \\
c & ::= & x \mid \mathbf{false} \mid \mathbf{true} \mid \lambda \mid \mathbf{let}\ x = c_1\ \mathbf{in}\ c_2 \\
 & \mid & (c_1, c_2) \mid \mathbf{let}\ (x_1, x_2) = c_1\ \mathbf{in}\ c_2 \mid \mathbf{if}\ c\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2 \\
 & \mid & \mathbf{int}\, c \mid c_1 + c_2 \mid c_1 - c_2 \mid c_1 * c_2 \mid c_1 / c_2 \mid c_1^{c_2} \mid c_1 = c_2 \mid c_1 < c_2
\end{array}
$$

Variables in nQML are viewed as references to quantum information that is stored in a global quantum state. There are two types of quantum information: qubits and products. A new qubit is allocated in the quantum state when the superposition operator $\{\, (\lambda)\, \mathbf{qfalse} + (\lambda')\, \mathbf{qtrue} \,\}$ is used, in the same way that new objects are allocated on the heap when a data constructor is used in a functional programming language. Products are introduced and eliminated with the constructs (e_1, e_2) and **let** $(x_1, x_2) = e_1$ **in** e_2. nQML also features three control constructs:

- **ifm** e **then** e_1 **else** e_2: It conducts a measurement on e, which must be of type qubit. Depending on the result, it executes one of its branches. It is similar to a classical random branching, based on a toss of a biased coin with probabilities depending on the state of the qubit being measured.
- **if** e **then** e_1 **else** e_2: It allows the programmer to perform quantum branching. If e, which must be of type qubit, is in a classical state, then the effect is what we would expect from **ifm**. But if e is in a quantum superposition, the program proceeds in a quantum superposition of both branches, most likely creating entanglement among the qubits of the quantum state.
- $|e\rangle \rightarrow x, x'. \, c$: A generic means of expressing any unitary transformation, which has to be relied upon when a transformation can not be easily broken down to a series of controlled operations, expressible with **if**. Its advantage is that, rather than forcing programmers to precompute and provide the whole unitary matrix of the transformation, whose size is exponential in the number of qubits that it affects, it allows them to express that matrix as a complex function of the input and output state of the transformed qubits. This leads to a succinct and clear expression of many useful quantum algorithms, such as the quantum Fourier transform that is described in Section 4.

In quantum pseudocode notation, all unitary transformations can be expressed in the form:

$$|i\rangle \to \sum_{j=0}^{2^n-1} f(i,j)\,|j\rangle$$

where $f(i,j)$ is a function of the input state i of the quantum register and its output state j. The construct $|e\rangle \to x, x'.\, c$ allows the programmers to use precisely this natural notation: the classical variables x and x' denote the register's input and output state and the classical expression c denotes the function's body.

From this notation, if the function f is known, the unitary matrix can be easily constructed by taking $S_{j,i} = f(i,j)$. Of course, not all functions f result in unitary matrices and the type system of nQML cannot decide whether the resulting transformation is indeed unitary. The type system of Altenkirch and Grattage's QML is able to do that, at the expense of making the size of the program exponential and complicating the typing with orthogonality constraints.

3.1 The type system of nQML

There are two kinds of types: quantum types (τ) and classical types (ϕ). For each quantum expression, the type system of nQML keeps track of the exact qubits of the state in which the value of this expression is stored. This information is stored in the types. It is used to make sure that the same qubit cannot be used twice in a transformation, thus allowing qubit aliasing without breaking the "no cloning" rule.

$$\tau \ ::= \ \mathbf{qbit}[n] \ | \ \tau_1 \otimes \tau_2$$
$$\phi \ ::= \ \mathbf{bit} \ | \ \phi_1 \times \phi_2 \ | \ \mathbf{complex}$$

For example, an expression has type $\mathbf{qbit}[5]$ if its value is stored in the 5th qubit of the state.

For each quantum type τ, we define $\mathcal{C}(\tau)$ to be the corresponding classical type; no quantum types correspond to **complex**. We denote by $|\mathcal{C}(\tau)|$ the size, in classical bits, of the classical type corresponding to τ and by $\boldsymbol{qbits}(\tau)$ the set of the state's qubits that are used by expressions of type τ. For example, $\boldsymbol{qbits}(\mathbf{qbit}[4] \otimes \mathbf{qbit}[2]) = \{2, 4\}$. A quantum type τ is called *pure* if its representation uses distinct qubits. Notice that, in general, $|\boldsymbol{qbits}(\tau)| \leq |\mathcal{C}(\tau)|$, the two being equal if and only if the type τ is pure. A quantum type environment Γ is a mapping of variables to quantum types and, similarly, a classical type environment Δ is a mapping of variables to classical types. $\Gamma|_k$ denotes the environment Γ restricted in such a way that it does not contain variables whose types use the state's k-th qubit.

The typing relation for nQML is denoted by $\Gamma; n \vdash^\alpha e : \tau; m$. More precisely, as in the type system of Altenkirch and Grattage's QML, there are two typing relations: one for pure quantum expressions (i.e. without measurements), denoted by $\Gamma; n \vdash^\circ e : \tau; m$, and one for arbitrary quantum expressions, denoted by $\Gamma; n \vdash e : \tau; m$. We refer to both by allowing the superscript $^\alpha$ to be either $^\circ$ or empty. As the types of nQML convey information regarding the position of qubits in the quantum state, the typing relation is forced to process and propagate such information. In $\Gamma; n \vdash^\alpha$

$e : \tau; m$, the natural number n appearing on the left side of the relation stands for the number of qubits of the original quantum state, before e starts evaluating. Obviously, for all pairs $(x : \tau_x) \in \Gamma$ it must be $\boldsymbol{qbits}(\tau_x) \subseteq \{0, \dots n-1\}$. The natural number m appearing on the right side of the relation stands for the number of new qubits, that are allocated during the evaluation of e. The final quantum state after e has been evaluated has $n+m$ qubits and, obviously again, it must be $\boldsymbol{qbits}(\tau) \subseteq \{0, \dots n+m-1\}$.

The typing rules for nQML follow Altenkirch and Grattage's QML, with the exception that the type system is not linear and qubit information must be processed. For example, the typing rule for quantum superposition plans for the allocation of one new qubit and uses its position in the returned type.

$$\frac{|\lambda|^2 + |\lambda'|^2 = 1}{\Gamma; n \vdash^{\circ} \{\, (\lambda)\,\mathbf{qfalse} + (\lambda')\,\mathbf{qtrue}\,\} : \mathbf{qbit}[n]; 1} \; \textit{(SUP)}$$

Rules with more than one quantum expression must carefully combine the newly allocated qubits, e.g.

$$\frac{\Gamma; n \vdash^{\alpha} e_1 : \tau_1; m_1 \qquad \Gamma; n+m_1 \vdash^{\alpha} e_2 : \tau_2; m_2}{\Gamma; n \vdash^{\alpha} (e_1, e_2) : \tau_1 \otimes \tau_2; m_1 + m_2} \; \textit{(PROD)}$$

The most complex of nQML's typing rules are those for the control constructs. We explain two of them below. In a quantum branching expression **if** e **then** e_1 **else** e_2 the control qubit must not be used in the two branches. This restriction is necessary to simplify the semantics of **if** and eliminate the need for orthogonal branches. Unitary transformations which cannot easily be described as quantum controlled operations have their own dedicated construct in nQML. Notice also that the number of newly allocated qubits takes the maximum of the two branches.

$$\frac{\begin{array}{c}\Gamma; n \vdash^{\alpha} e : \mathbf{qbit}[k]; m \\ \Gamma|_k; n+m \vdash^{\circ} e_1 : \tau; m_1 \qquad \Gamma|_k; n+m \vdash^{\circ} e_2 : \tau; m_2\end{array}}{\Gamma; n \vdash^{\alpha} \mathbf{if}\ e\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 : \tau; m + \max(m_1, m_2)} \; \textit{(IF)}$$

The typing rule for nQML's new construct $|e\rangle \to x, x'.\, c$ is also straightforward. A unitary transformation is performed on the quantum bits where the value of expression e is stored. The type τ of this expression must be pure, to obey the "no cloning" rule. In the classical expression c which determines the contents of the transformation, the two variables x and x' are bound to the classical value of the expression. The type of both is $\mathcal{C}(\tau)$.

$$\frac{\Gamma; n \vdash^{\alpha} e : \tau; m \qquad \boldsymbol{pure}(\tau) \qquad x : \mathcal{C}(\tau), x' : \mathcal{C}(\tau) \vdash c : \mathbf{complex}}{\Gamma; n \vdash^{\alpha} |e\rangle \to x, x'.\, c : \tau; m} \; \textit{(TRANS)}$$

The typing $\Delta \vdash c : \phi$ of classical expressions presents no difficulties.

3.2 The denotational semantics of nQML

Our denotational semantics for nQML uses density matrices for representing the quantum state. The semantic domain $\mathbf{S}(n) \subset \mathbb{C}^{2^n \times 2^n}$ contains density matrices. The meaning of an arbitrary well-typed expression e with a type derivation $\Gamma; n \vdash e : \tau; m$ is a function of type $\mathbf{S}(n) \to \mathbf{S}(n+m)$; it maps an input quantum state of n

qubits to an output quantum state of $n+m$ qubits. Pure quantum expressions that perform no measurements can be assigned unitary transformations as meanings. We denote by $\mathbf{T}(n) \subset \mathbb{C}^{2^n \times 2^n}$ the domain of unitary transformation matrices. If e is a well-typed pure quantum expression with a type derivation $\Gamma; n \vdash^\circ e : \tau; m$, then its meaning is a unitary transformation matrix of type $\mathbf{T}(n+m)$. The semantics of embedding pure quantum expressions in impure quantum expressions is given below. The tensor product of $A : \mathbf{S}(n)$ with the matrix Δ_m appropriately expands the state with m new qubits which are initialized with zeroes.

$$\begin{array}{lrcl} \textit{EMB}: & [\![\Gamma; n \vdash e : \tau; m]\!](A) & = & T\,(A \otimes \Delta_m)\,T^* \\ & \textbf{where} \quad T & = & [\![\Gamma; n \vdash^\circ e : \tau; m]\!] \end{array}$$

The use of a variable has no effect on the state, as variables are just references. However, superpositions extend the state by allocating a new qubit and appropriately initializing it.

$$\begin{array}{ll} \textit{VAR}: & [\![\Gamma; n \vdash^\circ x : \tau; 0]\!] = \mathbb{I}_n \\ \textit{SUP}: & [\![\Gamma; n \vdash^\circ \{\,(\lambda)\,\mathbf{qfalse} + (\lambda')\,\mathbf{qtrue}\,\} : \mathbf{qbit}[n]; 1]\!] = \\ & \quad \mathbb{I}_n \otimes \begin{pmatrix} \lambda & \lambda' \\ \lambda' & -\lambda \end{pmatrix} \end{array}$$

The semantics of the **let** construct, product introduction and elimination is straightforward and very similar. In each of them, evaluation begins with the evaluation of e_1 and continues with the evaluation of e_2 on the new state. The impure cases are very similar. [8]

$$\begin{array}{lrcl} \textit{LET}^\circ: & [\![\Gamma; n \vdash^\circ \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \tau; m_1 + m_2]\!] & = & T_2\,(T_1 \otimes \mathbb{I}_{m_2}) \\ & \textbf{where} \quad T_1 & = & [\![\Gamma; n \vdash^\circ e_1 : \tau_1; m_1]\!] \\ & T_2 & = & [\![\Gamma, x : \tau_1; n + m_1 \vdash^\circ e_2 : \tau; m_2]\!] \end{array}$$

The case of **if** is slightly more complicated. Evaluation begins with the condition. The matrices that correspond to the two branches are calculated and their (inexistent) effect on the control bit is removed by using the auxiliary function ***except***. Then, the two expressions are executed conditionally, with e as the control qubit. The impure case is again very similar.

$$\begin{array}{lrcl} \textit{IF}^\circ: & \multicolumn{3}{l}{[\![\Gamma; n \vdash^\circ \mathbf{if}\ e\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 : \tau; m + \max(m_1, m_2)]\!] =} \\ & \multicolumn{3}{l}{\quad T_c\,(T \otimes \mathbb{I}_{\max(m_1,m_2)})} \\ & \textbf{where} \quad T & = & [\![\Gamma; n \vdash^\circ e : \mathbf{qbit}[k]; m]\!] \\ & T_1 & = & [\![\Gamma|_k; n + m \vdash^\circ e_1 : \tau; m_1]\!] \\ & T_2 & = & [\![\Gamma|_k; n + m \vdash^\circ e_2 : \tau; m_2]\!] \\ & T_1' & = & \boldsymbol{except}(k, T_1) \otimes \mathbb{I}_{\max(m_1,m_2)-m_1} \\ & T_2' & = & \boldsymbol{except}(k, T_2) \otimes \mathbb{I}_{\max(m_1,m_2)-m_2} \\ & T_c & = & \boldsymbol{cond}(k, T_1', T_2') \end{array}$$

[8] It can easily be proved that the semantics of pure and impure quantum expressions is consistent with the embedding rule. For example, the meaning is the same if EMB is applied separately to two pure expressions and then PROD is applied to the result, or if EMB is applied once to the result of PROD.

Surprisingly, the measuring conditional **ifm** is more straightforward. The condition is evaluated and then the corresponding qubit is measured. The auxiliary function ***measure*** returns the two density matrices that correspond to collapsing a qubit to a classical state. Then the two branches are combined. Each branch is evaluated on the corresponding result state of the measurement and their sum is the total result.

$$
\begin{array}{lrcl}
\textit{IFM}: & [\![\Gamma; n \vdash \textbf{ifm}\ e\ \textbf{then}\ e_1\ \textbf{else}\ e_2 : \tau; m + \max(m_1, m_2)]\!](A) & = & \\
& \multicolumn{3}{c}{B_1 \otimes \Delta_{\max(m_1,m_2)-m_1} + B_2 \otimes \Delta_{\max(m_1,m_2)-m_2}} \\
& \textbf{where}\quad B & = & [\![\Gamma; n \vdash e : \textbf{qbit}[k]; m]\!](A) \\
& (B_t, B_f) & = & \boldsymbol{measure}(k, B) \\
& B_1 & = & [\![\Gamma; n+m \vdash e_1 : \tau; m_1]\!](B_t \otimes \Delta_{m_1}) \\
& B_2 & = & [\![\Gamma; n+m \vdash e_2 : \tau; m_2]\!](B_f \otimes \Delta_{m_2})
\end{array}
$$

Finally, in the semantics of $|e\rangle \to x, x'.\, c$ the described unitary transformation C is computed. As C only applies to the qubits used by e, it must be properly expanded to apply to the complete state.

$$
\begin{array}{lrcl}
\textit{TRANS}^\circ: & [\![\Gamma; n \vdash^\circ |e\rangle \to x, x'.\, c : \tau; m]\!] & = & T_c T \\
& \textbf{where}\quad T_c & = & \boldsymbol{expand}(n, \boldsymbol{qbits}(\tau), C) \\
& T & = & [\![\Gamma; n \vdash^\circ e : \tau; m]\!] \\
& C_{j,i} & = & [\![x : \mathcal{C}(\tau), x' : \mathcal{C}(\tau) \vdash c : \textbf{complex}]\!](\rho) \\
& \multicolumn{3}{l}{\textbf{where}\quad \rho \ = \ \rho_0\{x \mapsto \boldsymbol{val}_\tau(i)\}\{x' \mapsto \boldsymbol{val}_\tau(j)\}} \\
& \multicolumn{3}{l}{\quad\text{for all } 0 \le i, j < 2^k, \text{ where } k = |\boldsymbol{qbits}(\tau)|}
\end{array}
$$

Again, the semantics of classical expressions is standard and presents no difficulty.

3.3 Some metatheory

We start with the notion of well-formed types and environments, with respect to the current state. A type is only well-formed if all the qubit positions that it refers to exist in the state.

Definition 3.1 A type τ is *well-formed* in a state of n qubits, written as $n \vdash \tau$, if for all $k \in \boldsymbol{qbits}(\tau)$ we have $k < n$.

Definition 3.2 An environment Γ is *well-formed* in a state of n qubits, written as $n \vdash \Gamma$, if for all $(x : \tau) \in \Gamma$ we have $n \vdash \tau$.

It is then easy to prove the following theorem, which states that the types produced by the typing relation are well-formed.

Theorem 3.3 *If* $\Gamma; n \vdash^\alpha e : \tau; m$ *and* $n \vdash \Gamma$ *then* $n + m \vdash \tau$.

Proof (Sketch) By a straightforward induction on the typing derivation. □

To prove that the denotational semantics is well-defined is a little trickier. It is only true if all constructs $|e\rangle \to x, x'.\, c$ define unitary transformations.

Definition 3.4 A transformation construct $|e\rangle \to x, x'.\, c$ with a typing $\Gamma; n \vdash^\circ |e\rangle \to x, x'.\, c : \tau; m$ is *well-defined* if the matrix C in equation *TRANS*$^\circ$ is unitary.

Lemma 3.5 *If* $\Gamma|_k; n \vdash^\circ e : \tau; m$ *then the matrix* $[\![\Gamma|_k; n \vdash^\circ e : \tau; m]\!]$ *is of the form* $A \otimes \mathbb{I}_1 \otimes B$ *where* $A : \mathbb{C}^{2^k \times 2^k}$*, i.e. it is a transformation that leaves unchanged the k-th qubit of the state.*

Proof (Sketch) By induction on the typing derivation. Most cases are tedious and the interesting ones are *IF* and *TRANS*. □

Theorem 3.6 *If* $\Gamma; n \vdash^\alpha e : \tau; m$ *and all transformation constructs in e are well-defined with their typings, then* $[\![\Gamma; n \vdash^\alpha e : \tau; m]\!]$ *is well-defined and:*

(i) *if* $\alpha = \circ$*, then the matrix* $[\![\Gamma; n \vdash^\circ e : \tau; m]\!]$ *is unitary.*

(ii) *if* $\alpha =$ *empty, then the function* $[\![\Gamma; n \vdash e : \tau; m]\!]$ *maps density matrices to density matrices.*

Proof (Sketch) By induction on the typing derivation. The proof uses the closure properties of density and unitary matrices. The case of *TRANS* is handled by the assumption. The only case worth mentioning is *IF*, which uses the matrices ***except***(k, T_1) and ***except***(k, T_2), where k is the qubit position of the condition. It is necessary at that point to prove that T_1 and T_2 are of the form $A \otimes \mathbb{I}_1 \otimes B$, where $A : \mathbb{C}^{2^k \times 2^k}$. This can be obtained by the previous lemma. □

4 Examples

We now present a few example nQML programs of varying complexity. We start with two useful operators in quantum programming: ***not*** and ***had***, standing respectively for quantum negation and the Hadamard transformation. Both can be applied to any expression q of type $\mathbf{qbit}[n]$.

$$\begin{aligned} \textit{\textbf{not}}(q) &\equiv |q\rangle \to x, x'.\, \textbf{if } x' = x \textbf{ then } 0 \textbf{ else } 1 \\ \textit{\textbf{had}}(q) &\equiv |q\rangle \to x, x'.\, \textbf{if } x \textbf{ then } (\textbf{if } x' \textbf{ then } -\tfrac{1}{\sqrt{2}} \textbf{ else } \tfrac{1}{\sqrt{2}}) \textbf{ else } \tfrac{1}{\sqrt{2}} \end{aligned}$$

These simple transformations may seem a bit awkward at first but now that we have defined them we can easily use them in conjunction with the quantum conditional construct to define more complex transformations. For example, controlled quantum negation of p by q can be defined as:

$$\textit{\textbf{cnot}}(q, p) \equiv \textbf{if } q \textbf{ then } \textit{\textbf{not}}(p) \textbf{ else } p$$

where ***not***(p) is defined as above.

This leads us to our first nQML program: an implementation of Deutsch's algorithm. In this algorithm we are presented with a black box classical one-bit boolean function and we want to decide whether it is balanced, in which case we return 1, or constant, in which case we return 0. We assume that the unknown function is somehow included in our program and we write $f(q)$ for the application of that function to a quantum parameter q. By using the definition of ***had*** and ***not*** given above, we arrive to the following program.

$$\begin{aligned} \textit{\textbf{deutsch}}(f) \equiv\ & \textbf{let } (i, j) = (\{\, (\tfrac{1}{\sqrt{2}})\, \textbf{qfalse} + (\tfrac{1}{\sqrt{2}})\, \textbf{qtrue} \,\}, \\ & \qquad\qquad\quad \{\, (\tfrac{1}{\sqrt{2}})\, \textbf{qfalse} + (-\tfrac{1}{\sqrt{2}})\, \textbf{qtrue} \,\}) \textbf{ in} \end{aligned}$$

```
    let $r = $ if $f(i)$ then $\mathit{not}(j)$ else $j$ in
    $\mathit{had}(i)$
```

The program's result is stored in variable i. This variable is used as the first operand of our branching operator, after f is applied to it. When f is a constant function and therefore $f(i)$ has a classical value, i will be unaffected by the execution of the branching and its result after the Hadamard transform will be 0. When, however, f is balanced, i.e. it is the identity or the negation function, even though its application will have no direct effect on i, the use of i as a control bit for j's negation means that the two variables interact non-classically.

Let us now see a few more examples that demonstrate the power of $|e\rangle \to x, x'. c$. Addition of a constant to a n-bit quantum register modulo 2^n, which is typically denoted by $|r\rangle \to |r + c\rangle$ in quantum pseudocode, can be implemented as:

$$\mathit{add}(r, c) \;\equiv\; |r\rangle \to x, x'.\,\textbf{if int}\, x' = \textbf{int}\, x + c \textbf{ then } 1 \textbf{ else } 0$$

Any other permutation of base states can easily be implemented in a similar manner. The implementation of the quantum Fourier transform for n qubits contained in register r is:

$$\mathit{fourier}(r, n) \;\equiv\; |r\rangle \to x, x'.\, 1/2^n * e^{2*\pi*i*x*x'/2^n}$$

which is derived in a straightforward way from the transform's definition. In Selinger's QPL, one can do the same by applying the unitary matrix S corresponding to the quantum Fourier transform to the quantum register r, using the construct $r *\!= S$. However, unless some sophisticated language is used in combination with QPL to represent unitary transformations, the programmer has to use a precalculated S and, as its size is $2^n \times 2^n$, the size of the program increases exponentially. The same is true in the case of Altenkirch and Grattage's QML, where the transform can be implemented by a tree of height n containing nested **if**$^\circ$ branches; the size of the program is again exponential in n.

As a last example, let us see an implementation of Grover's fast database search. Assuming that c denotes the value we are searching for, we first need to implement the query and diffusion operators.

```
    $\mathit{query}(q)$          $\equiv$  $|q\rangle \to x, x'.$ if $x = x'$
                                        then (if int $x = c$ then $-1$ else $1$)
                                        else $0$
    $\mathit{diffusion}(q, n)$   $\equiv$  $|q\rangle \to x, x'.$ if $x = x'$ then $-1 + 2/2^n$ else $2/2^n$
```

Let us consider the most simple application of Grover's algorithm: searching in a space of size 4 ($n = 2$ qubits). Even though $O(\sqrt{n})$ applications of the two operators are generally needed to obtain high probability, in this special case one application is enough to produce the correct result with certainty:

```
    $\mathit{grover}$  $\equiv$  let $q_1 = \{ (\frac{1}{\sqrt{2}})$ qfalse $+ (\frac{1}{\sqrt{2}})$ qtrue $\}$ in
                let $q_2 = \{ (\frac{1}{\sqrt{2}})$ qfalse $+ (\frac{1}{\sqrt{2}})$ qtrue $\}$ in
                let $qs = (q_1, q_2)$ in
                $\mathit{diffusion}(\mathit{query}(qs), 2)$
```

Assuming that the element we were looking for was $c = 2$, the Haskell interpreter that implements our semantics produces the following state (density matrix) of two qubits:

$$\begin{pmatrix} 0.0 :+ 0.0 & 0.0 :+ 0.0 & 0.0 :+ 0.0 & 0.0 :+ 0.0 \\ 0.0 :+ 0.0 & 0.0 :+ 0.0 & 0.0 :+ 0.0 & 0.0 :+ 0.0 \\ 0.0 :+ 0.0 & 0.0 :+ 0.0 & 0.9999999999999997 :+ 0.0 & 0.0 :+ 0.0 \\ 0.0 :+ 0.0 & 0.0 :+ 0.0 & 0.0 :+ 0.0 & 0.0 :+ 0.0 \end{pmatrix}$$

where $\alpha :+ \beta$ is Haskell's notation for the complex number $\alpha + \beta i$. From it, we can easily verify that the correct answer was found: the register qs is in the classical state $|10\rangle$, allowing for numerical errors.

5 Towards polymorphic higher-order functions

In the examples of the previous section we have used parametric expressions in nQML, such as $\boldsymbol{not}(q)$ or $\boldsymbol{cnot}(q, p)$. Such parametric expressions were used as *macros*: when used in another expression, $\boldsymbol{not}(e)$ is syntactically expanded by substituting the expression e for q in the body of $\boldsymbol{not}$. Macro expansion is an easy way to enjoy some of the advantages of having functions in nQML, without resorting to a more complex type system. Macros can even simulate higher-order functions, such as $\boldsymbol{deutsch}(f)$ where f is a function from qubit to qubit, or polymorphic functions, such as $\boldsymbol{add}(r, c)$ where r is a quantum register of unknown type. They cannot, however, simulate recursive functions or currying.

In this section, we sketch a polymorphic type system for nQML to support higher-order functions. Although we do not deal with the denotational semantics of the extended language here, we believe that it is possible to extend nQML with polymorphic higher-order recursive functions, as also suggested by Selinger and Valiron [10]. The type system that is briefly presented here lays the ground for such an extension.

We adopt a very conservative extension to the syntax of nQML. Types are not visible by programmers; a type inference algorithm is used to typecheck the programs of the extended language (see section 5.4). Functions take arguments of three kinds: quantum expressions, classical expressions and function names. For simplicity, we separate the three kinds of arguments in the syntax. We denote by f a function name. We also denote by $\vec{e}$ a list of quantum expressions, by $\vec{c}$ a list of classical expressions, by $\vec{f}$ a list of function names, etc. A program p is a list of function definitions, followed by a quantum expression to be computed. A function definition d determines the function's name, the function's arguments' names and the function's body.

$$\begin{aligned} e &::= \ \ldots \ \mid \ f(\vec{e}; \vec{c}; \vec{f}) \\ d &::= \ f(\vec{x}; \vec{y}; \vec{f}) \equiv e \\ p &::= \ \vec{d}; e \end{aligned}$$

5.1 Types

To justify our design of nQML's extended type system, we begin by discussing the types of the macros of section 4. A naïve type system would give function ***not*** the type $\mathbf{qbit}[3] \to \mathbf{qbit}[3]$. Such a type would make sense, but the function would only be useful if its argument resided in the 3rd qubit of the state. For functions to be useful, function types must be polymorphic w.r.t. the exact qubits to which they refer. A more useful polymorphic type for ***not*** would be:

$$\textit{\textbf{not}} \quad : \quad \forall k.\ \mathbf{qbit}[k] \to \mathbf{qbit}[k]$$

where the variable k ranges over qubit positions. We take this idea even further and we disallow types of the form $\mathbf{qbit}[n]$ in function arguments, for any constant n. All function arguments must be polymorphic w.r.t. the qubits in which they reside.

However, qubit positions are not the only source of polymorphism in our extension. Consider the function ***add***, whose first argument is a quantum register of unknown type. It is reasonable to give ***add*** the polymorphic type:

$$\textit{\textbf{add}} \quad : \quad \forall t.\ (t; \mathbf{complex}) \to t$$

where the variable t ranges over quantum types. Similarly, a function type may be polymorphic in a variable u ranging over classical types.

Consider now a function that returns its result in a newly allocated qubit.[9] In order to know the exact qubit that will be used for the result, we need to know the number of qubits in the caller's state. As this may vary, we need to make the function's type polymorphic w.r.t. the number of qubits of the state. Fortunately we have disallowed constant qubit types in function arguments, so we can do this implicitly by taking the type $\mathbf{qbit}[m]$ in a function's result, where m is a constant, to refer to the m-th available qubit above the ones used by caller's state. For the logistics of the typing rule for function application, it will be useful to know the total number of qubits allocated by the function's body and whether the body of a function performs a pure or impure computation; this information should also be present in the function's type. As an example, consider a function of type:

$$f \quad : \quad \forall k_1, k_2.\ \mathbf{qbit}[k_1] \otimes \mathbf{qbit}[k_2] \to^{\circ} \mathbf{qbit}[2]; 4$$

If the caller's state uses n qubits (obviously k_1 and k_2 are two of these qubits, or even the same qubit) then the result of f will reside in the $(n+2)$-th qubit of the state and the state after the call will contain $n+4$ qubits. The function's type also reveals that applying this function results in a pure expression.

We thus extend the syntax of types as follows, where b is a new syntactic class for qubit positions:

$$\begin{aligned} b &::= k \mid n \\ \tau &::= t \mid \mathbf{qbit}[b] \mid \tau_1 \otimes \tau_2 \\ \phi &::= u \mid \mathbf{bit} \mid \phi_1 \times \phi_2 \mid \mathbf{complex} \end{aligned}$$

[9] ***deutsch*** is such a function, but we will not give it a type yet, as it is second-order.

As a last reference to our examples in section 4, consider the function ***cnot***. The type $\forall k_1, k_2.\ (\mathbf{qbit}[k_1], \mathbf{qbit}[k_2]) \rightarrow \mathbf{qbit}[k_2]; 0$ is not enough because, for the function's body to typecheck, the typing rule for **if** requires that $k_1 \neq k_2$. This information must be added to the function's type and, to do this, we need a syntax for *constraints*. One kind of constraint is $\mathbf{disjoint}(\tau_1, \tau_2)$, which provides us with the information that $\boldsymbol{qbits}(\tau_1) \cap \boldsymbol{qbits}(\tau_2) = \emptyset$.

$$\begin{aligned} \boldsymbol{cnot} \quad : \quad & \forall k_1, k_2.\ \mathbf{disjoint}(\mathbf{qbit}[k_1], \mathbf{qbit}[k_2]) \Rightarrow \\ & \quad (\mathbf{qbit}[k_1], \mathbf{qbit}[k_2]) \rightarrow \mathbf{qbit}[k_2]; 0 \end{aligned}$$

This is precisely what is needed for the function's body to typecheck. Other kinds of constraints state that a quantum type τ is pure and that a quantum type τ and a classical type ϕ satisfy $\mathcal{C}(\tau) = \phi$.

$$\kappa \ ::= \ \mathbf{pure}(\tau) \ \mid \ \mathbf{disjoint}(\tau, \tau) \ \mid \ \mathbf{classic}(\tau, \phi)$$

With all these in mind, the syntax of function types θ in our extended type system is the following:

$$\theta \ ::= \ \forall(\vec{k}; \vec{t}; \vec{u}).\ \vec{\kappa} \Rightarrow (\vec{\tau}; \vec{\phi}; \vec{\theta}) \rightarrow^{\alpha} \tau; n$$

and the type of ***deutsch*** is:

$$\boldsymbol{deutsch} \quad : \quad (\forall k.\ .\mathbf{qbit}[k] \rightarrow \mathbf{qbit}[k]) \rightarrow \mathbf{qbit}[0]; 2$$

5.2 Typing

Some new environments are required for the typing of the extended language. In addition to the environments Γ and Δ that provide the types of quantum and classical variables respectively, two new environments are necessary: $\mathcal{E}$ for polymorphic (type) variables and Φ for function names. The environment $\mathcal{E}$ associates polymorphic variables with their *kind*. There are three kinds (#, $*$, \$), where $k : \#$ denotes that k is a qubit position variable, $t : *$ denotes that t is a quantum type variable and $u : \$$ denotes that u is a classical type variable. Moreover, Φ associates function names with function types $(f : \theta)$.

The typing relation for the extended language is similar to that of section 3: $\mathcal{E}; C; \Gamma; \Xi; \Delta; \Phi; n \vdash^{\alpha} e : \tau; m$. In addition to the environments Γ, Δ, $\mathcal{E}$ and Φ, we have added two more elements. C is a set of constraints and Ξ is a set of qubit positions. Their purpose will become apparent in what follows. The typing relation for classical expressions is also extended: $\mathcal{E}; C; \Delta \vdash c : \phi$. We add two new typing relations for definitions and programs: $\Phi \vdash d$ and $\Phi \vdash p$.

One of the existing typing rules that needs to change is the rule for **if**. Consider the case of a function f that is polymorphic in the qubit position k. Suppose that f takes an argument x of type $\mathbf{qbit}[k]$ and that the body of f contains the expression **if** x **then** e_1 **else** e_2. In the presence of a polymorphic qubit position b, it is hard to construct the environment $\Gamma|_b$ by excluding from Γ the variables whose types mention b. It is easier to keep a set Ξ of qubit positions that must not be used in the typing of quantum expressions. The new rule for **if** extends Ξ with b.

$$\frac{\begin{array}{c}\mathcal{E}; C; \Gamma; \Xi; \Delta; \Phi; n \vdash^{\alpha} e : \mathbf{qbit}[b]; m \\ \mathcal{E}; C; \Gamma; \Xi, b; \Delta; \Phi; n+m \vdash^{\circ} e_1 : \tau; m_1 \\ \mathcal{E}; C; \Gamma; \Xi, b; \Delta; \Phi; n+m \vdash^{\circ} e_2 : \tau; m_2\end{array}}{\mathcal{E}; C; \Gamma; \Xi; \Delta; \Phi; n \vdash^{\alpha} \textbf{if } e \textbf{ then } e_1 \textbf{ else } e_2 : \tau; m + \max(m_1, m_2)} \text{ (IF)}$$

The problem is then shifted to the typing of variables. Before deciding that a variable x has type τ, we must make sure that τ does not mention any of the qubit positions contained in Ξ. This may not be possible to verify without access to the function's constraints, and thus we invent the judgement $C \models \boldsymbol{qbits}(\tau) \cap \Xi = \emptyset$.

$$\frac{(x : \tau) \in \Gamma \qquad C \models \boldsymbol{qbits}(\tau) \cap \Xi = \emptyset}{\mathcal{E}; C; \Gamma; \Xi; \Delta; \Phi; n \vdash^{\circ} x : \tau; 0} \text{ (VAR)}$$

In a similar fashion, the typing rule for $|e\rangle \to x, x'. c$ needs to change, to support transforming expressions of polymorphic types.

$$\frac{\begin{array}{c}\mathcal{E}; C; \Gamma; \Xi; \Delta; \Phi; n \vdash^{\alpha} e : \tau; m \qquad C \models \boldsymbol{pure}(\tau) \qquad C \models \mathcal{C}(\tau) = \phi \\ \mathcal{E}; C; \Delta, x : \phi, x' : \phi \vdash c : \mathbf{complex}\end{array}}{\mathcal{E}; C; \Gamma; \Xi; \Delta; \Phi; n \vdash^{\alpha} |e\rangle \to x, x'. c : \tau; m} \text{ (TRANS)}$$

The definition of appropriate inference rules for our new "verification" judgements (of the form $C \models \mathit{prop}$) is far from trivial, but we shall not deal with it in this paper. In the rest of this section, we shift our attention to the typing rules for function application and function definition.

Let f be a polymorphic function of type $\forall(\vec{k}; \vec{t}; \vec{u}).\ \vec{\kappa} \Rightarrow (\vec{\tau}; \vec{\phi}; \vec{\theta}) \to^{\alpha} \tau; m$. When typechecking a call to f, the polymorphic variables $\vec{k}$, $\vec{t}$ and $\vec{u}$ must be substituted with actual qubit positions, quantum and classical types. This is performed by a *substitution* σ, whose formal definition we omit here. We denote by $\sigma\{\tau\}$ the effect of σ on the quantum type τ, by $\sigma\{\phi\}$ its effect on the classical type ϕ, etc. The actual arguments to f must typecheck with the σ-substituted types of the formal arguments. Furthermore, the σ-substituted function constraints must be verified. The type of the result is τ, appropriately "shifted" and σ-substituted. We denote by $\tau \uparrow^{m}$ the type that results from adding m to all constant qubit positions in τ.

$$\frac{\begin{array}{c}(f : \forall(\vec{k}; \vec{t}; \vec{u}).\ \vec{\kappa} \Rightarrow (\vec{\tau}; \vec{\phi}; \vec{\theta}) \to^{\alpha} \tau; m_2) \in \Phi \\ \mathcal{E}; C; \Gamma; \Xi; \Delta; \Phi; n \vdash^{\alpha} \vec{e} : \sigma\{\vec{\tau}\}; m_1 \qquad \mathcal{E}; C; \Delta \vdash \vec{c} : \sigma\{\vec{\phi}\} \\ (\vec{f} : \sigma\{\vec{\theta}\}) \in \Phi \qquad\qquad C \models \sigma\{\vec{\kappa}\}\end{array}}{\mathcal{E}; C; \Gamma; \Xi; \Delta; \Phi; n \vdash^{\alpha} f(\vec{e}; \vec{c}; \vec{f}) : \sigma\{\tau \uparrow^{n+m_1}\}; m_1 + m_2} \text{ (APP)}$$

Typechecking the definition of a polymorphic function f is easier. The function's body must typecheck and return the appropriate type, with the appropriate initial environments. Notice two things when typechecking a function's body: (i) the only external environment that is used is Φ, and (ii) $n = 0$ is used, so that the numbering of the newly created qubits starts from 0.

$$\frac{\begin{array}{c}(f : \forall(\vec{k}; \vec{t}; \vec{u}).\ \vec{\kappa} \Rightarrow (\vec{\tau}; \vec{\phi}; \vec{\theta}) \to^{\alpha} \tau; m) \in \Phi \\ \vec{k} : \#, \vec{t} : *, \vec{u} : \$; \vec{\kappa}; \vec{x} : \vec{\tau}; \emptyset; \vec{y} : \vec{\phi}; \Phi, \vec{f} : \vec{\theta}; 0 \vdash^{\alpha} e : \tau; m\end{array}}{\Phi \vdash f(\vec{x}; \vec{y}; \vec{f}) := e} \text{ (DEF)}$$

5.3 Some metatheory

In this section, we restrict ourselves to the well-formedness of types.

Definition 5.1 A type τ is *well-formed* in a state of n qubits and a type environment $\mathcal{E}$, written as $\mathcal{E}; n \vdash \tau$, if for all $n' \in \boldsymbol{qbits}(\tau)$ we have $n' < n$ and all type variables used by τ are defined in $\mathcal{E}$ with the appropriate kind.

Definition 5.2 A classical type ϕ is *well-formed* in a type environment $\mathcal{E}$, written as $\mathcal{E} \vdash \phi$, if all type variables used by ϕ are defined in $\mathcal{E}$ with the appropriate kind.

Definition 5.3 A constraint κ is *well-formed* in a type environment $\mathcal{E}$, written as $\mathcal{E} \vdash \kappa$, if all type variables used by κ are defined in $\mathcal{E}$ with the appropriate kind.

Definition 5.4 A function type $\theta \equiv \forall(\vec{k}; \vec{t}; \vec{u}).\ \vec{\kappa} \Rightarrow (\vec{\tau}; \vec{\phi}; \vec{\theta}) \rightarrow^{\alpha} \tau; m$ is *well-formed* in a type environment $\mathcal{E}$, written as $\mathcal{E} \vdash \theta$, if in the type environment $\mathcal{E}' \equiv \mathcal{E}, \vec{k} : \#, \vec{t} : *, \vec{u} : \$$ we have $\mathcal{E}' \vdash \vec{\kappa}$, $\mathcal{E}'; 0 \vdash \vec{\tau}$, $\mathcal{E}' \vdash \vec{\phi}$, $\mathcal{E}' \vdash \vec{\theta}$ and $\mathcal{E}'; m \vdash \tau$.

Definition 5.5 An environment Γ is *well-formed* in a state of n qubits and a type environment $\mathcal{E}$, written as $\mathcal{E}; n \vdash \Gamma$, if for all $(x : \tau) \in \Gamma$ we have $\mathcal{E}; n \vdash \tau$.

Definition 5.6 A classical environment Δ is *well-formed* in a type environment $\mathcal{E}$, written as $\mathcal{E} \vdash \Delta$, if for all $(y : \phi) \in \Delta$ we have $\mathcal{E} \vdash \phi$.

Definition 5.7 A function type environment Φ is *well-formed* in a type environment $\mathcal{E}$, written as $\mathcal{E} \vdash \Phi$, if for all $(f : \theta) \in \Phi$ we have $\mathcal{E} \vdash \theta$.

Definition 5.8 A set of constraints C is *well-formed* in a type environment $\mathcal{E}$, written as $\mathcal{E} \vdash C$, if for all $\kappa \in C$ we have $\mathcal{E} \vdash \kappa$.

The following theorem states that the types produced by the typing relation are well-formed.

Theorem 5.9 *If* $\mathcal{E}; C; \Gamma; \Xi; \Delta; \Phi; n \vdash^{\alpha} e : \tau; m$, $\mathcal{E} \vdash C$, $\mathcal{E}; n \vdash \Gamma$, $\mathcal{E} \vdash \Delta$ *and* $\mathcal{E} \vdash \Phi$, *then* $\mathcal{E}; n + m \vdash \tau$.

Proof (Sketch) By induction on the typing derivation. It uses several "weakening" lemmata, e.g. if $\mathcal{E}; n \vdash \tau$ then $\mathcal{E}; n' \vdash \tau$ for all $n' \geq n$. The case of *APP* requires a substitution lemma. □

5.4 Type inference

Although the syntax of nQML extended with functions is rather simple, its type system is very complicated. Fortunately, a type inference algorithm can be used to automatically calculate the types of functions. The algorithm is based on type unification: it generates a set of *unification constraints* whose solution provides a program's missing types. In most aspects, this algorithm is simpler than Hindley-Milner style type inference algorithms, because in nQML polymorphism does not extend to function types and currying is not allowed. However, it faces the problem of calculating one set of constraints $\vec{\kappa}$ for each polymorphic function in the program. Such constraints are gathered from the typing rules when typechecking the bodies of functions. They are updated and the process is repeated, until a fixed point is reached. Although the results from using this type inference algorithm in practice are adequate, a thorough theoretical analysis is still missing.

6 Conclusion

Quantum programming is today more or less at the same point in its history as classical programming was in the 1940s. The hardware is non existent or faulty. The semantics of quantum programming languages is understood either at a very low level of abstraction, using quantum gates and circuits, or at a very high level of abstraction, using tensor products in categories of Hilbert spaces. One thing that is different, though, is our experience of more than half a century in the theory and practice of classical programming languages. It is this experience that must be put into work if, sometime in the future, quantum programming languages are going to be what classical programming languages are today. Quantum programming must exploit the advantages of the quantum computational model, putting aside its peculiarities and insignificant details, so that programmers can add two "quantum integers" and obtain another "quantum integer" without, for example, having to think about the reversibility of this computation.

It can be argued that our work takes the "quantum data and control" paradigm a very small step further towards simplicity. We have defined nQML, a new functional quantum programming language, inspired by Altenkirch and Grattage's QML and following the "quantum data and control" paradigm. The type system of nQML keeps track of the use of qubits in expressions and avoids the complexities of linear type systems. This type system scales well to include polymorphic higher-order functions and admits a type inference algorithm. The semantics of nQML is inspired by Selinger's semantics for QPL. It is a simple denotational semantics with density matrices and unitary transformations as the semantic domains, which leads naturally to a simple implementation, in the form of an interpreter written in Haskell. Furthermore, the $|e\rangle \to x, x'.\, c$ construct allows quantum algorithms to be implemented in a more direct and natural way.

References

[1] Altenkirch, T. and J. Grattage, *A functional quantum programming language*, in: *Proceedings of the 20th Annual IEEE Symposium on Logic in Computer Science*, 2005, pp. 249–258.

[2] Gay, S. J., *Quantum programming languages: Survey and bibliography*, Mathematical Structures in Computer Science **16** (2006), pp. 581–600.

[3] Grattage, J. and T. Altenkirch, *A compiler for a functional quantum programming language* (2005), manuscript.

[4] Grattage, J. and T. Altenkirch, *QML: Quantum data and control* (2005), manuscript.

[5] Grover, L. K., *A fast quantum mechanical algorithm for database search*, in: *Proceedings of the 28th Annual ACM Symposium on the Theory of Computing*, Philadelphia, PA, 1996, pp. 212–219.

[6] Knill, E., *Conventions for quantum pseudocode*, Technical Report LAUR-96-2724, Los Alamos National Laboratory (1996).

[7] Ömer, B., "Structured Quantum Programming," Ph.D. thesis, Institute of Information Systems, Technical University of Vienna (2003).

[8] Sanders, J. W. and P. Zuliani, *Quantum programming*, in: *Proceedings of the 5th International Conference on Mathematics of Program Construction*, Lecture Notes in Computer Science **1837** (2000), pp. 80–99.

[9] Selinger, P., *Towards a quantum programming language*, Mathematical Structures in Computer Science **14** (2004), pp. 527–586.

[10] Selinger, P. and B. Valiron, *A lambda calculus for quantum computation with classical control*, Mathematical Structures in Computer Science **16** (2006), pp. 527–552.

[11] Shor, P. W., *Polynomial time algorithms for prime factorization and discrete logarithms on a quantum computer*, SIAM Journal on Computing **26** (1997), pp. 1484–1509.

[12] van Tonder, A., *A lambda calculus for quantum computation*, SIAM Journal on Computing **33** (2004), pp. 1109–1135.

A Formal definition of nQML

A.1 *Syntax*

$$\begin{array}{lcl} e & ::= & x \mid \{(\lambda)\,\mathbf{qfalse} + (\lambda')\,\mathbf{qtrue}\} \mid \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \mid (e_1, e_2) \mid \mathbf{let}\ (x_1, x_2) = e_1\ \mathbf{in}\ e_2 \\ & \mid & \mathbf{if}\ e\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 \mid \mathbf{ifm}\ e\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 \mid |e\rangle \to x, x'.\, c \end{array}$$

$$\begin{array}{lcl} c & ::= & x \mid \lambda \mid \mathbf{false} \mid \mathbf{true} \mid \mathbf{let}\ x = c_1\ \mathbf{in}\ c_2 \mid (c_1, c_2) \mid \mathbf{let}\ (x_1, x_2) = c_1\ \mathbf{in}\ c_2 \\ & \mid & \mathbf{int}\,c \mid c_1 + c_2 \mid c_1 - c_2 \mid c_1 * c_2 \mid c_1/c_2 \mid c_1^{c_2} \mid c_1 = c_2 \mid c_1 < c_2 \mid \mathbf{if}\ c\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2 \end{array}$$

A.2 *Typing*

Types: quantum and classical

$$\tau \ ::= \ \mathbf{qbit}[n] \mid \tau_1 \otimes \tau_2$$
$$\phi \ ::= \ \mathbf{bit} \mid \phi_1 \times \phi_2 \mid \mathbf{complex}$$

From quantum to classical types

$$\begin{array}{lcl} \mathcal{C}(\mathbf{qbit}[n]) & = & \mathbf{bit} \\ \mathcal{C}(\tau_1 \otimes \tau_2) & = & \mathcal{C}(\tau_1) \times \mathcal{C}(\tau_2) \end{array}$$

Size of classical types

$$\begin{array}{lcl} |\mathcal{C}(\mathbf{qbit}[n])| & = & 1 \\ |\mathcal{C}(\tau_1 \otimes \tau_2)| & = & |\mathcal{C}(\tau_1)| + |\mathcal{C}(\tau_2)| \\ |\mathbf{complex}| & = & \text{undefined} \end{array}$$

Qubits used by a quantum type

$$\begin{array}{lcl} \mathit{qbits}(\tau) & : & \mathcal{P}(\mathbb{N}) \\ \mathit{qbits}(\mathbf{qbit}[n]) & = & \{n\} \\ \mathit{qbits}(\tau_1 \otimes \tau_2) & = & \mathit{qbits}(\tau_1) \cup \mathit{qbits}(\tau_2) \end{array}$$

Pure quantum types

$$\frac{}{\mathit{pure}(\mathbf{qbit}[n])} \qquad \frac{\mathit{pure}(\tau_1) \quad \mathit{pure}(\tau_2) \quad \mathit{qbits}(\tau_1) \cap \mathit{qbits}(\tau_2) = \emptyset}{\mathit{pure}(\tau_1 \otimes \tau_2)}$$

Type environments: quantum and classical

$$\begin{array}{lcl} \Gamma & : & \text{a finite set of pairs of the form } (x : \tau) \\ \Delta & : & \text{a finite set of pairs of the form } (x : \phi) \\ \Gamma|_k & = & \{(x : \tau) \in \Gamma \mid k \notin \mathit{qbits}(\tau)\} \end{array}$$

Typing relation for quantum expressions $\boxed{\Gamma; n \vdash^{\alpha} e : \tau; m}$

where ${}^{\alpha}$ is empty or ${}^{\circ}$

$$\frac{\Gamma; n \vdash^\circ e : \tau; m}{\Gamma; n \vdash e : \tau; m}\ \textit{(EMB)} \qquad \frac{(x : \tau) \in \Gamma}{\Gamma; n \vdash^\circ x : \tau; 0}\ \textit{(VAR)}$$

$$\frac{|\lambda|^2 + |\lambda'|^2 = 1}{\Gamma; n \vdash^\circ \{ (\lambda)\, \mathbf{qfalse} + (\lambda')\, \mathbf{qtrue} \} : \mathbf{qbit}[n]; 1}\ \textit{(SUP)}$$

$$\frac{\Gamma; n \vdash^\alpha e_1 : \tau_1; m_1 \qquad \Gamma, x : \tau_1; n + m_1 \vdash^\alpha e_2 : \tau; m_2}{\Gamma; n \vdash^\alpha \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \tau; m_1 + m_2}\ \textit{(LET)}$$

$$\frac{\Gamma; n \vdash^\alpha e_1 : \tau_1; m_1 \qquad \Gamma; n + m_1 \vdash^\alpha e_2 : \tau_2; m_2}{\Gamma; n \vdash^\alpha (e_1, e_2) : \tau_1 \otimes \tau_2; m_1 + m_2}\ \textit{(PROD)}$$

$$\frac{\Gamma; n \vdash^\alpha e_1 : \tau_1 \otimes \tau_2; m_1 \qquad \Gamma, x_1 : \tau_1, x_2 : \tau_2; n + m_1 \vdash^\alpha e_2 : \tau; m_2}{\Gamma; n \vdash^\alpha \mathbf{let}\ (x_1, x_2) = e_1\ \mathbf{in}\ e_2 : \tau; m_1 + m_2}\ \textit{(LETPROD)}$$

$$\frac{\begin{array}{c}\Gamma; n \vdash^\alpha e : \mathbf{qbit}[k]; m \\ \Gamma|_k; n + m \vdash^\circ e_1 : \tau; m_1 \qquad \Gamma|_k; n + m \vdash^\circ e_2 : \tau; m_2\end{array}}{\Gamma; n \vdash^\alpha \mathbf{if}\ e\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 : \tau; m + \max(m_1, m_2)}\ \textit{(IF)}$$

$$\frac{\Gamma; n \vdash e : \mathbf{qbit}[k]; m \qquad \Gamma; n + m \vdash e_1 : \tau; m_1 \qquad \Gamma; n + m \vdash e_2 : \tau; m_2}{\Gamma; n \vdash \mathbf{ifm}\ e\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 : \tau; m + \max(m_1, m_2)}\ \textit{(IFM)}$$

$$\frac{\Gamma; n \vdash^\alpha e : \tau; m \qquad \textit{pure}(\tau) \qquad x : \mathcal{C}(\tau), x' : \mathcal{C}(\tau) \vdash c : \mathbf{complex}}{\Gamma; n \vdash^\alpha |e\rangle \to x, x'.\, c : \tau; m}\ \textit{(TRANS)}$$

Typing relation for classical expressions $\boxed{\Delta \vdash c : \phi}$

$$\frac{(x : \phi) \in \Delta}{\Delta \vdash x : \phi}\ \textit{(var)} \qquad \frac{}{\Delta \vdash \mathbf{false} : \mathbf{bit}}\ \textit{(false)} \qquad \frac{}{\Delta \vdash \mathbf{true} : \mathbf{bit}}\ \textit{(true)}$$

$$\frac{}{\Delta \vdash \lambda : \mathbf{complex}}\ \textit{(const)} \qquad \frac{\Delta \vdash c_1 : \phi_1 \qquad \Delta, x : \phi_1 \vdash c_2 : \phi}{\Delta \vdash \mathbf{let}\ x = c_1\ \mathbf{in}\ c_2 : \phi}\ \textit{(let)}$$

$$\frac{\Delta \vdash c_1 : \phi_1 \qquad \Delta \vdash c_2 : \phi_2}{\Delta \vdash (c_1, c_2) : \phi_1 \times \phi_2}\ \textit{(prod)} \qquad \frac{\Delta \vdash c_1 : \phi_1 \times \phi_2 \qquad \Delta, x_1 : \phi_1, x_2 : \phi_2 \vdash c_2 : \phi}{\Delta \vdash \mathbf{let}\ (x_1, x_2) = c_1\ \mathbf{in}\ c_2 : \phi}\ \textit{(letprod)}$$

$$\frac{\Delta \vdash c : \mathcal{C}(\tau)}{\Delta \vdash \mathbf{int}\, c : \mathbf{complex}}\ \textit{(int)} \qquad \frac{\Delta \vdash c : \mathbf{bit} \qquad \Delta \vdash c_1 : \phi \qquad \Delta \vdash c_2 : \phi}{\Delta \vdash \mathbf{if}\ c\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2 : \phi}\ \textit{(if)}$$

$$\frac{\Delta \vdash c_1 : \mathbf{complex} \qquad \Delta \vdash c_2 : \mathbf{complex} \qquad op \in \{+, -, *, /, \hat{}\ \}}{\Delta \vdash c_1\, op\, c_2 : \mathbf{complex}}\ \textit{(arith)}$$

$$\frac{\Delta \vdash c_1 : \phi \qquad \Delta \vdash c_2 : \phi}{\Delta \vdash c_1 = c_2 : \mathbf{bit}}\ \textit{(eq)} \qquad \frac{\Delta \vdash c_1 : \mathbf{complex} \qquad \Delta \vdash c_2 : \mathbf{complex}}{\Delta \vdash c_1 < c_2 : \mathbf{bit}}\ \textit{(lt)}$$

A.3 Semantics

Semantic domains

$$\begin{array}{lcl}
\mathbf{S}(n) & = & \left\{ A \in \mathbb{C}^{2^n \times 2^n} \mid A \text{ is a density matrix} \right\} \\
\mathbf{T}(n) & = & \left\{ T \in \mathbb{C}^{2^n \times 2^n} \mid T \text{ is unitary} \right\} \\
[\![\Delta]\!] & = & \Pi x{:}\mathbf{Var}.\ [\![\Delta(x)]\!] \\
[\![\mathbf{bit}]\!] & = & \mathbb{B} \\
[\![\phi_1 \times \phi_2]\!] & = & [\![\phi_1]\!] \times [\![\phi_2]\!] \\
[\![\mathbf{complex}]\!] & = & \mathbb{C}
\end{array}$$

Semantics of pure quantum expressions $\boxed{[\![\Gamma; n \vdash^\circ e : \tau; m]\!] : \mathbf{T}(n + m)}$

VAR: $[\![\Gamma; n \vdash^\circ x : \tau; 0]\!] = \mathbb{I}_n$

SUP: $[\![\Gamma; n \vdash^\circ \{ (\lambda)\, \mathbf{qfalse} + (\lambda')\, \mathbf{qtrue} \} : \mathbf{qbit}[n]; 1]\!] = \mathbb{I}_n \otimes \begin{pmatrix} \lambda & \lambda' \\ \lambda' & -\lambda \end{pmatrix}$

LET°: $[\![\Gamma; n \vdash^\circ \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \tau; m_1 + m_2]\!] = T_2\,(T_1 \otimes \mathbb{I}_{m_2})$
where $T_1 = [\![\Gamma; n \vdash^\circ e_1 : \tau_1; m_1]\!]$
$T_2 = [\![\Gamma, x : \tau_1; n + m_1 \vdash^\circ e_2 : \tau; m_2]\!]$

PROD°: $[\![\Gamma; n \vdash^\circ (e_1, e_2) : \tau_1 \otimes \tau_2; m_1 + m_2]\!] = T_2\,(T_1 \otimes \mathbb{I}_{m_2})$
where $T_1 = [\![\Gamma; n \vdash^\circ e_1 : \tau_1; m_1]\!]$
$T_2 = [\![\Gamma; n + m_1 \vdash^\circ e_2 : \tau_2; m_2]\!]$

LETPROD°: $[\![\Gamma; n \vdash^\circ \mathbf{let}\ (x_1, x_2) = e_1\ \mathbf{in}\ e_2 : \tau; m_1 + m_2]\!] = T_2\,(T_1 \otimes \mathbb{I}_{m_2})$
where $T_1 = [\![\Gamma; n \vdash^\circ e_1 : \tau_1 \otimes \tau_2; m_1]\!]$
$T_2 = [\![\Gamma, x_1 : \tau_1, x_2 : \tau_2; n + m_1 \vdash^\circ e_2 : \tau; m_2]\!]$

IF°: $[\![\Gamma; n \vdash^\circ \mathbf{if}\ e\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 : \tau; m + \max(m_1, m_2)]\!] =$
$T_c\,(T \otimes \mathbb{I}_{\max(m_1,m_2)})$
where $T = [\![\Gamma; n \vdash^\circ e : \mathbf{qbit}[k]; m]\!]$
$T_1 = [\![\Gamma|_k; n + m \vdash^\circ e_1 : \tau; m_1]\!]$
$T_2 = [\![\Gamma|_k; n + m \vdash^\circ e_2 : \tau; m_2]\!]$
$T_1' = \mathit{except}(k, T_1) \otimes \mathbb{I}_{\max(m_1,m_2)-m_1}$
$T_2' = \mathit{except}(k, T_2) \otimes \mathbb{I}_{\max(m_1,m_2)-m_2}$
$T_c = \mathit{cond}(k, T_1', T_2')$

TRANS°: $[\![\Gamma; n \vdash^\circ |e\rangle \to x, x'.\, c : \tau; m]\!] = T_c\, T$
where $T_c = \mathit{expand}(n, \mathit{qbits}(\tau), C)$
$T = [\![\Gamma; n \vdash^\circ e : \tau; m]\!]$
$C_{j,i} = [\![x : \mathcal{C}(\tau), x' : \mathcal{C}(\tau) \vdash c : \mathbf{complex}]\!](\rho)$
where $\rho = \rho_0\{x \mapsto \mathit{val}_\tau(i)\}\{x' \mapsto \mathit{val}_\tau(j)\}$
for all $0 \le i, j < 2^k$, where $k = |\mathit{qbits}(\tau)|$

Semantics of impure quantum expressions $\boxed{[\![\Gamma; n \vdash e : \tau; m]\!] : \mathbf{S}(n) \to \mathbf{S}(n+m)}$

EMB: $[\![\Gamma; n \vdash e : \tau; m]\!](A) = T\,(A \otimes \Delta_m)\,T^*$
where $T = [\![\Gamma; n \vdash^\circ e : \tau; m]\!]$

LET: $[\![\Gamma; n \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \tau; m_1 + m_2]\!](A) = B_2$
where $B_1 = [\![\Gamma; n \vdash e_1 : \tau_1; m_1]\!](A)$
$B_2 = [\![\Gamma, x : \tau_1; n + m_1 \vdash e_2 : \tau; m_2]\!](B_1)$

PROD: $[\![\Gamma; n \vdash (e_1, e_2) : \tau_1 \otimes \tau_2; m_1 + m_2]\!](A) = B_2$
where $B_1 = [\![\Gamma; n \vdash e_1 : \tau_1; m_1]\!](A)$
$B_2 = [\![\Gamma; n + m_1 \vdash e_2 : \tau_2; m_2]\!](B_1)$

LETPROD: $[\![\Gamma; n \vdash \mathbf{let}\ (x_1, x_2) = e_1\ \mathbf{in}\ e_2 : \tau; m_1 + m_2]\!](A) = B_2$
where $B_1 = [\![\Gamma; n \vdash e_1 : \tau_1 \otimes \tau_2; m_1]\!](A)$
$B_2 = [\![\Gamma, x_1 : \tau_1, x_2 : \tau_2; n + m_1 \vdash e_2 : \tau; m_2]\!](B_1)$

IF: $[\![\Gamma; n \vdash \mathbf{if}\ e\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 : \tau; m + \max(m_1, m_2)]\!](A) =$
$T_c\,(B \otimes \Delta_{\max(m_1,m_2)})\,T_c^*$
where $B = [\![\Gamma; n \vdash e : \mathbf{qbit}[k]; m]\!](A)$
$T_1 = [\![\Gamma|_k; n + m \vdash^\circ e_1 : \tau; m_1]\!]$
$T_2 = [\![\Gamma|_k; n + m \vdash^\circ e_2 : \tau; m_2]\!]$
$T_1' = \mathit{except}(k, T_1) \otimes \mathbb{I}_{\max(m_1,m_2)-m_1}$
$T_2' = \mathit{except}(k, T_2) \otimes \mathbb{I}_{\max(m_1,m_2)-m_2}$
$T_c = \mathit{cond}(k, T_1', T_2')$

IFM: $[\![\Gamma; n \vdash \mathbf{ifm}\ e\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 : \tau; m + \max(m_1, m_2)]\!](A) =$
$B_1 \otimes \Delta_{\max(m_1,m_2)-m_1} + B_2 \otimes \Delta_{\max(m_1,m_2)-m_2}$
where $B = [\![\Gamma; n \vdash e : \mathbf{qbit}[k]; m]\!](A)$
$(B_t, B_f) = \mathit{measure}(k, B)$
$B_1 = [\![\Gamma; n + m \vdash e_1 : \tau; m_1]\!](B_t \otimes \Delta_{m_1})$
$B_2 = [\![\Gamma; n + m \vdash e_2 : \tau; m_2]\!](B_f \otimes \Delta_{m_2})$

TRANS: $[\![\Gamma; n \vdash |e\rangle \to x, x'.\, c : \tau; m]\!](A) = T_c\, B\, T_c^*$
where $T_c = \mathit{expand}(n, \mathit{qbits}(\tau), C)$
$B = [\![\Gamma; n \vdash e : \tau; m]\!](A)$
$C_{j,i} = [\![x : \mathcal{C}(\tau), x' : \mathcal{C}(\tau) \vdash c : \mathbf{complex}]\!](\rho)$
where $\rho = \rho_0\{x \mapsto \mathit{val}_\tau(i)\}\{x' \mapsto \mathit{val}_\tau(j)\}$
for all $0 \le i, j < 2^k$, where $k = |\mathit{qbits}(\tau)|$

Semantics of classical expressions $\boxed{[\![\Delta \vdash c : \phi]\!] : [\![\Delta]\!] \to [\![\phi]\!]}$

var: $[\![\Delta \vdash x : \phi]\!](\rho) = \rho(x)$
false: $[\![\Delta \vdash \lambda : \mathbf{false}]\!](\rho) =$ false

true: $[\![\Delta \vdash \lambda : \mathbf{true}]\!](\rho) \quad = \quad \text{true}$

const: $[\![\Delta \vdash \lambda : \mathbf{complex}]\!](\rho) \quad = \quad \lambda$

let: $[\![\Delta \vdash \mathbf{let}\ x = c_1\ \mathbf{in}\ c_2 : \phi]\!](\rho) \quad = \quad [\![\Delta, x : \phi_1 \vdash c_2 : \phi]\!](\rho')$
$\quad \mathbf{where} \quad \rho' \quad = \quad \rho\{x \mapsto [\![\Delta \vdash c_1 : \phi_1]\!](\rho)\}$

prod: $[\![\Delta \vdash (c_1, c_2) : \phi_1 \times \phi_2]\!](\rho) \quad = \quad ([\![\Delta \vdash c_1 : \phi_1]\!](\rho), [\![\Delta \vdash c_2 : \phi_2]\!](\rho))$

letprod: $[\![\Delta \vdash \mathbf{let}\ (x_1, x_2) = c_1\ \mathbf{in}\ c_2 : \phi]\!](\rho) \quad =$
$\quad [\![\Delta, x_1 : \phi_1, x_2 : \phi_2 \vdash c_2 : \phi]\!](\rho')$
$\quad \mathbf{where} \quad (v_1, v_2) \quad = \quad [\![\Delta \vdash c_1 : \phi_1 \times \phi_2]\!](\rho)$
$\quad \rho' \quad = \quad \rho\{x \mapsto v_1\}\{y \mapsto v_2\}$

int: $[\![\Delta \vdash \mathbf{int}\, c : \mathbf{complex}]\!](\rho) \quad = \quad \mathit{code}_\tau([\![\Delta \vdash c : \mathcal{C}(\tau)]\!](\rho))$

arith: $[\![\Delta \vdash c_1\ \mathit{op}\ c_2 : \mathbf{complex}]\!](\rho) \quad =$
$\quad [\![\Delta \vdash c_1 : \mathbf{complex}]\!](\rho)\ \mathit{op}\ [\![\Delta \vdash c_2 : \mathbf{complex}]\!](\rho)$

eq: $[\![\Delta \vdash c_1 = c_2 : \mathbf{bit}]\!](\rho) \quad =$
$$\begin{cases} \text{true}, \text{ if } [\![\Delta \vdash c_1 : \phi]\!](\rho) = [\![\Delta \vdash c_2 : \phi]\!](\rho) \\ \text{false}, \text{ if } [\![\Delta \vdash c_1 : \phi]\!](\rho) \neq [\![\Delta \vdash c_2 : \phi]\!](\rho) \end{cases}$$

lt: $[\![\Delta \vdash c_1 < c_2 : \mathbf{bit}]\!](\rho) \quad =$
$$\begin{cases} \text{true}, \text{ if } [\![\Delta \vdash c_1 : \mathbf{complex}]\!](\rho) < [\![\Delta \vdash c_2 : \mathbf{complex}]\!](\rho) \\ \text{false}, \text{ if } [\![\Delta \vdash c_1 : \mathbf{complex}]\!](\rho) \geq [\![\Delta \vdash c_2 : \mathbf{complex}]\!](\rho) \end{cases}$$

if: $[\![\Delta \vdash \mathbf{if}\ c\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2 : \phi]\!](\rho) \quad =$
$$\begin{cases} [\![\Delta \vdash c_1 : \phi]\!](\rho), \text{ if } [\![\Delta \vdash c : \mathbf{bit}]\!](\rho) = \text{true} \\ [\![\Delta \vdash c_2 : \phi]\!](\rho), \text{ if } [\![\Delta \vdash c : \mathbf{bit}]\!](\rho) = \text{false} \end{cases}$$

Auxiliary functions

$\mathbb{I}_n$: the identity matrix of size $2^n \times 2^n$
Δ_n : a matrix of size $2^n \times 2^n$ with all zeroes and a 1 in the top-left corner

except : $\mathbb{N} \times \mathbf{S}(n+1) \to \mathbf{S}(n)$

$$\mathit{except}(0, \left(\begin{array}{c|c} A & \mathbb{O} \\ \hline \mathbb{O} & A \end{array}\right)) \quad = \quad A$$

$$\mathit{except}(k+1, \left(\begin{array}{c|c} A & B \\ \hline C & D \end{array}\right)) \quad = \quad \left(\begin{array}{c|c} \mathit{except}(k, A) & \mathit{except}(k, B) \\ \hline \mathit{except}(k, C) & \mathit{except}(k, D) \end{array}\right)$$

cond : $\mathbb{N} \times \mathbf{S}(n) \times \mathbf{S}(n) \to \mathbf{S}(n+1)$

$$\mathit{cond}(0, T, F) \quad = \quad \left(\begin{array}{c|c} F & \mathbb{O} \\ \hline \mathbb{O} & T \end{array}\right)$$

$$\mathit{cond}(k+1, \left(\begin{array}{c|c} T_A & T_B \\ \hline T_C & T_D \end{array}\right), \left(\begin{array}{c|c} F_A & F_B \\ \hline F_C & F_D \end{array}\right)) \quad = \quad \left(\begin{array}{c|c} \mathit{cond}(k, T_A, F_A) & \mathit{cond}(k, T_B, F_B) \\ \hline \mathit{cond}(k, T_C, F_C) & \mathit{cond}(k, T_D, F_D) \end{array}\right)$$

measure : $\mathbb{N} \times \mathbf{S}(n+1) \to \mathbf{S}(n+1) \times \mathbf{S}(n+1)$

$$\mathit{measure}(0, \left(\begin{array}{c|c} A & B \\ \hline C & D \end{array}\right)) \quad = \quad (\left(\begin{array}{c|c} \mathbb{O} & \mathbb{O} \\ \hline \mathbb{O} & D \end{array}\right), \left(\begin{array}{c|c} A & \mathbb{O} \\ \hline \mathbb{O} & \mathbb{O} \end{array}\right))$$

$$\mathit{measure}(k+1, \left(\begin{array}{c|c} A & B \\ \hline C & D \end{array}\right)) \quad = \quad (\left(\begin{array}{c|c} T_A & T_B \\ \hline T_C & T_D \end{array}\right), \left(\begin{array}{c|c} F_A & F_B \\ \hline F_C & F_D \end{array}\right))$$

$$\begin{aligned} \mathbf{where} \quad (T_A, F_A) &= \mathit{measure}(k, A) \\ (T_B, F_B) &= \mathit{measure}(k, B) \\ (T_C, F_C) &= \mathit{measure}(k, C) \\ (T_D, F_D) &= \mathit{measure}(k, D) \end{aligned}$$

expand : $\Pi n{:}\mathbb{N}.\ \Pi S{:}\mathcal{P}(\mathbb{N}).\ \mathbf{T}(|S|) \to \mathbf{T}(n)$

$\mathit{expand}(n, S, T) \quad = \quad \mathit{expa}_0(n, S, T)$

$\quad \mathbf{where} \quad \mathit{expa}_n(n, S, T) \quad = \quad T$

$$\mathit{expa}_k(n, S, \left(\begin{array}{c|c} A & B \\ \hline C & D \end{array}\right)) \quad = \quad \left(\begin{array}{c|c} \mathit{expa}_{k+1}(n, S, A) & \mathit{expa}_{k+1}(n, S, C) \\ \hline \mathit{expa}_{k+1}(n, S, B) & \mathit{expa}_{k+1}(n, S, D) \end{array}\right)$$

$\quad\quad$ if $k < n$ and $k \in S$

$\quad \mathit{expa}_k(n, S, T) \quad = \quad \mathbb{I}_1 \otimes \mathit{expa}_{k+1}(n, S, T)$

$\quad\quad$ if $k < n$ and $k \notin S$

code$_\tau$: $[\![\mathcal{C}(\tau)]\!] \to \mathbb{N}$

$$\mathit{code}_{\mathbf{qbit}[k]}(b) \quad = \quad \begin{cases} 1, \text{ if } b = \text{true} \\ 0, \text{ if } b = \text{false} \end{cases}$$

$$\begin{array}{lcl} \mathit{code}_{\tau_1 \otimes \tau_2}(v_1, v_2) & = & 2^k\, \mathit{code}_{\tau_1}(v_1) + \mathit{code}_{\tau_2}(v_2) \\ \quad \textbf{where} \quad k & = & |\mathcal{C}(\tau_2)| \end{array}$$

$$\begin{array}{lcl} \mathit{val}_{\tau} & : & \mathbb{N} \to [\![\mathcal{C}(\tau)]\!] \\ \mathit{val}_{\mathbf{qbit}[k]}(n) & = & \begin{cases} \text{true} & \text{, if } n = 1 \\ \text{false} & \text{, if } n = 0 \end{cases} \\ \mathit{val}_{\tau_1 \otimes \tau_2}(n) & = & (\mathit{val}_{\tau_1}(n/2^k), \mathit{val}_{\tau_2}(n \bmod 2^k)) \\ \quad \textbf{where} \quad k & = & |\mathcal{C}(\tau_2)| \end{array}$$