Quantum Programming Made Easy

Luca Paolini
Dipartimento di Informatica
Università di Torino
Italy
luca.paolini@unito.it

Luca Roversi
Dipartimento di Informatica
Università di Torino
Italy
luca.roversi@unito.it

Margherita Zorzi
Dipartimento di Informatica
Università di Verona
Italy
margherita.zorzi@univr.it

We present IQu, namely a quantum programming language that extends Reynold's Idealized Algol, the paradigmatic core of Algol-like languages. IQu combines imperative programming with high-order features, mediated by a simple type theory. IQu mildly merges its quantum features with the classical programming style that we can experiment through Idealized Algol, the aim being to ease a transition towards the quantum programming world. The proposed extension is done along two main directions. First, IQu makes the access to quantum co-processors by means of quantum stores. Second, IQu includes some support for the direct manipulation of quantum circuits, in accordance with recent trends in the development of quantum programming languages. Finally, we show that IQu is quite effective in expressing well-known quantum algorithms.

1 Introduction

Linearity is an essential ingredient for quantum computing, since quantum data have to undergo restrictions such as non-cloning and non-erasing properties. This is evident from the care that quantum programming language design puts on the management of quantum bits, especially in presence of higher-order features.

Selinger's QPL [29] is a milestone for quantum programming theory. It follows the mainstream approach "*quantum data & classical control* " based on the architecture QRAM [10]. In QPL a classical program generates "directives" for an ideal quantum device. QPL is the first statically typed programming language that enforces the well-formedness of a program at compile time, by avoiding run-time checks. The non-duplication of quantum data is enforced by the syntax of the language. The lack of higher-order functions is the main limitation of QPL.

The introduction of QPL opened the way to the design of several quantum programming languages. Some of them came be found in [7, 6, 17, 36, 30, 31, 11]. Pagani et al. [17] and Zorzi [36] mainly focus on the computational models behind the language. Other papers, like [30], focus on pioneering prototypes of effective quantum programming languages. The languages in [30, 17, 36, 12] deal with higher-order functions and follow the direction suggested by Selinger in [29]: Linear Logic exponential modalities are used to devise typing systems which accomodate quantum data management in a classical programming setting.

A very pragmatic proposal is `Quipper` [33]. Its distinguishing feature is its management of quantum circuits as classical data. It allows to duplicate, erase, operate and meta-operate circuits. The prototypical example of meta-operations are identified in [10]: reversing quantum circuits, conditioning of quantum operations and converting classical algorithms in reversible ones. Some formalizations of the core of `Quipper`, called `ProtoQuipper` and `ProtoQuipper-M`, have been defined in [27, 28] and are base based on Linear-Logic typing systems.

Moving the focus from the quantum-data perspective to the classical-control one, we find recent quantum programming languages as qPCF [22] and `QWire` [23]. `QWire` has been conceived as an ex-

T. Ehrhard, M. Fernández, V. de Paiva, L. Tortora de Falco (Eds.):
Linearity-TLLA 2018
EPTCS 292, 2019, pp. 133–147, doi:10.4204/EPTCS.292.8

tension of a classical language that provides an elegant and manageable language for defining quantum circuits; Haskell and Coq have been considered as possible hosts (in fact, QWire is a "quantum plugin" for a host classical language). It provides a suitable support for quantum circuits (extended with measurement gates) through a boxing interface that rests on a Linear-Logic based typing system. If some normalization assumption holds, then the interface keeps the typing rules for elements that live in the quantum world apart from the typing system of the hosting language. In contrast, qPCF [22] is a stand-alone quantum programming language. It extends PCF with a type for quantum circuits and its operational semantics is supplied by means of a QRAM-compliant device with suitably relaxed decoherence assumptions. qPCF has two main distinctive type features: it uses dependent types and, it avoids types for quantum states, by relegating their relevance to quantum co-processor calls. More precisely, qPCF prevents the access to intermediate quantum states by tightening up the interaction process with the black-box quantum device: whole bunches of quantum directives are supplied to the device that ends their evaluation with a measurement of the whole state. For the sake of completeness, we recall that dependent type extensions of QWire are considered in [23, 26].

In this paper, we introduce IQu (read "*Haiku*" as the japanese poetic form) a new quantum programming language that extends Idealized Algol by conservatively inheriting its positive qualities. Reynolds's Idealized Algol is a paradigmatic language that elegantly combines the fundamental features of procedural languages, i.e. local stores and assignments, with a fully-fledged higher-order procedure mechanism which, in its turn, conservatively includes PCF. Idealized Algol's expressiveness and simple type theory have attracted the attention of many researchers [15].

IQu extends Idealized Algol by introducing two new types. The first one types quantum circuits, the second one types quantum variables. Considered that quantum circuits are classical data, their type allows to operate on them without any special care, as they were a kind of numerals (roughly, they are special strings). Instead, manipulating quantum states requires much care, for which we adapt Idealized Algol's original de-reference mechanism to access the content of classical variables. In Idealized Algol, a classical variable is a name for a classical value stored in a register. We cannot duplicate that register, but we can access its content via suitable methods and, if interested to that, we can duplicate that content. Following this approach, we introduce the type of "quantum variables" that prevent to read quantum states, but allow to measure them. The unique irreversible update of quantum stare allowed by IQu js the measurement, while all other transformations are required to be unitary ones.

Essentially, IQu can be seen as a higher-order extension of QPL [29] where we prevent the quantum state duplication not by means of a Linear Logic-based typing system, but putting them inside suitable variables, whose content cannot be duplicated. However, the duplication can be applied to the name of the variables to safely refer to their content in different spots of a program. IQu manipulates the quantum variables as much as possible as imperative variables.

According to the recent trend in the development of quantum programming languages, we introduce some facilities to duplicate and operate on quantum circuits. This approach stems from how quantum algorithms are usually described and built, from the far-sighted informal ideas proposed in [10] and, constitutes one of the distinctive features of Quipper, QWire and qPCF. However, IQu is different from ProtoQuipper-like approaches and QWire, since the linear management of quantum state does not rest on types based on Linear Logic.

In IQu we do not use dependent types for describing quantum circuits. This is to keep it as simple as possible. Therefore, formally, IQu does not extend qPCF. However, such an extension is possible, thus we state that IQu extends qPCF with classical and quantum stores.

Concluding, IQu is an original stand-alone higher-order type-safe quantum programming language.

The philosophy behind its design is to keep programming simple and rooted in the traditional classical programming approach in accordance with the Idealized Algol design. The proposed extension should help the transition from the classical programming to the quantum one. Moreover the language smoothly adapts to the common descriptions of quantum algorithms coherently with what Knill advocates in [10].
Outline. Section 2 introduces the row syntax of IQu and its typing system with some basic properties. Section 3 provides the details about its operational evaluation and its type safety. Section 4 concretely uses IQu to implement some well-known algorithms. The last section is about conclusions and future work.

2 IQu: Idealized QUantum language

IQu is a prototypical and minimal typed language that combines quantum commands and states with higher-order functional features by using registers. It is an extension Idealized Algol (see [15, 16]), namely a PCF that includes assignments and side-effects.

The grammar of IQu *ground types* is $\beta ::= \mathsf{Nat} \mid \mathsf{cVar} \mid \mathsf{qVar} \mid \mathsf{cmd} \mid \textsc{circ}$.

- Nat is the type of numerical expressions which evaluate to natural numbers;
- cVar is the type of imperative variables that store natural numbers;
- qVar is the type of quantum registers that store quantum states, in accordance with the QRAM model (so states are assumed free of decoherence issues);
- circ is the type of quantum-circuit expressions, i.e. expressions evaluating to strings that describe unitary transformations;
- cmd is the type of commands, i.e. the type of operations on variables (producing side-effects).

The *types* of IQu are the language that $\sigma, \tau, \theta ::= \beta \mid \theta \to \theta$ generates.
The *terms* of IQu belong to the row syntax:

$$\begin{array}{rcl} \mathtt{M},\mathtt{N},\mathtt{P},\mathtt{Q} & ::= & \mathtt{x} \mid \underline{\mathtt{n}} \mid \mathtt{pred} \mid \mathtt{succ} \mid \mathtt{if} \mid \lambda\mathtt{x}.\mathtt{M} \mid \mathtt{MN} \mid \mathtt{Y}_\sigma \\ & \mid & \mathtt{skip} \mid \mathtt{M};\mathtt{N} \mid \mathtt{while}\,\mathtt{P}\,\mathtt{do}\,\mathtt{Q} \mid \mathtt{M} := \mathtt{N} \mid \mathtt{read}\,\mathtt{M} \mid \mathtt{cnew}^{\mathtt{N}}\,\mathtt{x}\,\mathtt{in}\,\mathtt{M} \\ & \mid & \mathtt{U}^{\underline{\mathtt{k}}} \mid \ :: \ \mid \ \| \ \mid \mathtt{reverse} \mid \mathtt{csize} \mid \mathtt{rsize} \mid \mathtt{M} \lhd \mathtt{N} \mid \mathtt{meas}^{\mathtt{N}}\,\mathtt{M} \mid \mathtt{qnew}^{\mathtt{N}}\,\mathtt{x}\,\mathtt{in}\,\mathtt{M}\ . \end{array}$$

- The first line includes in IQu, a boolean-free call-by-name PCF, namely the sub-language of IQu which contains variables, numerals, predecessor, successor, conditional, abstraction, application and recursion;
- The second line includes in IQu, the imperative part of Idealized Algol [16, 25] which contains the commands do-nothing, composition, iteration, assignment, store reader (read) and store binder (cnew);
- The third line includes in IQu, the syntax of quantum circuits, basic operations on both quantum circuits and quantum registers.
  Quantum Circuits. We expect that evaluating a circuit expression yields (evaluated) circuits which are strings generated by the grammar $\mathtt{C} ::= \mathtt{U}^{\underline{\mathtt{k}}} \mid \mathtt{C}::\mathtt{C} \mid \mathtt{C} \parallel \mathtt{C}$ and which include, quantum gates ($\mathtt{U}^{\underline{\mathtt{k}}}$ denotes a gate in the set $\mathscr{U}(k)$ of gates operating on k wires), sequential and parallel compositions. A non null arity, i.e. the number of its inputs and of its output, labels every gate. For every gate symbol, the quantum co-processor [14] implements a unitary transformation to interpret it. W.l.o.g., we assume that a universal base of quantum transformations is available. The design of circuit's syntax is aimed to be basic and to predispose the use of dependent types in IQu.

$$\frac{}{B\cup\{\mathtt{x}:\sigma\}\vdash \mathtt{x}:\sigma}\ (tv) \qquad \frac{}{B\vdash \underline{\mathtt{n}}:\mathsf{Nat}}\ (tn) \qquad \frac{}{B\vdash \mathtt{succ}:\mathsf{Nat}\to\mathsf{Nat}}\ (ts) \qquad \frac{}{B\vdash \mathtt{pred}:\mathsf{Nat}\to\mathsf{Nat}}\ (tp)$$

$$\frac{B\cup\{\mathtt{x}:\sigma\}\vdash \mathtt{N}:\tau}{B\vdash \lambda\mathtt{x}^{\sigma}.\mathtt{N}:\sigma\to\tau}\ (tab) \qquad \frac{B\vdash \mathtt{P}:\sigma\to\tau \quad B\vdash \mathtt{Q}:\sigma}{B\vdash \mathtt{PQ}:\tau}\ (tap) \qquad \frac{}{B\vdash \mathtt{Y}_{\sigma}:(\sigma\to\sigma)\to\sigma}\ (tY)$$

$$\frac{}{B\vdash \mathtt{if}:\mathsf{Nat}\to\beta\to\beta\to\beta}\ (ti) \qquad \frac{}{B\vdash \mathtt{skip}:\mathsf{cmd}}\ (tk) \qquad \frac{B\vdash \mathtt{P}:\mathsf{cmd} \quad B\vdash \mathtt{Q}:\beta}{B\vdash \mathtt{P};\mathtt{Q}:\beta}\ (tc)$$

$$\frac{B\vdash \mathtt{P}:\mathsf{Nat} \quad B\vdash \mathtt{Q}:\mathsf{cmd}}{B\vdash \mathtt{while}\,\mathtt{P}\,\mathtt{do}\,\mathtt{Q}:\mathsf{cmd}}\ (tw) \qquad \frac{B\vdash \mathtt{M}:\mathsf{cVar} \quad B\vdash \mathtt{N}:\mathsf{Nat}}{B\vdash \mathtt{M}:=\mathtt{N}:\mathsf{cmd}}\ (tA) \qquad \frac{B\vdash \mathtt{M}:\mathsf{cVar}}{B\vdash \mathsf{read}\,\mathtt{M}:\mathsf{Nat}}\ (tR)$$

$$\frac{B\vdash \mathtt{N}:\mathsf{Nat} \quad B\cup\{\mathtt{x}:\mathsf{cVar}\}\vdash \mathtt{M}:\beta}{B\vdash \mathsf{cnew}^{\mathtt{N}}\,\mathtt{x}\,\mathtt{in}\,\mathtt{M}:\beta}\ (tcnw) \qquad \frac{\mathtt{U}\in\mathscr{U}(k)}{B\vdash \mathtt{U}^{\underline{\mathtt{k}}}:\mathsf{circ}}\ (tc1)$$

$$\frac{}{B\vdash ::\ \mathsf{circ}\to\mathsf{circ}\to\mathsf{circ}}\ (tc2) \qquad \frac{}{B\vdash \|:\mathsf{circ}\to\mathsf{circ}\to\mathsf{circ}}\ (tc3) \qquad \frac{}{B\vdash \mathtt{reverse}:\mathsf{circ}\to\mathsf{circ}}\ (tmc)$$

$$\frac{B\vdash \mathtt{M}:\mathsf{circ}}{B\vdash c\mathtt{size}(\mathtt{M}):\mathsf{Nat}}\ (tsc) \qquad \frac{B\vdash \mathtt{M}:\mathsf{qVar}}{B\vdash r\mathtt{size}(\mathtt{M}):\mathsf{Nat}}\ (tsr) \qquad \frac{B\vdash \mathtt{M}:\mathsf{qVar} \quad B\vdash \mathtt{N}:\mathsf{circ}}{B\vdash \mathtt{M}\lhd\mathtt{N}:\mathsf{cmd}}\ (tC)$$

$$\frac{B\vdash \mathtt{M}:\mathsf{qVar} \quad B\vdash \mathtt{N}:\mathsf{Nat}}{B\vdash \mathsf{meas}^{\mathtt{N}}\,\mathtt{M}:\mathsf{Nat}}\ (tM) \qquad \frac{B\vdash \mathtt{N}:\mathsf{Nat} \quad B\cup\{\mathtt{x}:\mathsf{qVar}\}\vdash \mathtt{M}:\beta}{B\vdash \mathsf{qnew}^{\mathtt{N}}\,\mathtt{x}\,\mathtt{in}\,\mathtt{M}:\beta}\ (tqnw)$$

Table 1: Typing Rules.

Quantum Circuit Operations. We limit our interest to prototypical quantum circuits operations. `reverse` is the standard quantum meta-operation for reversing (see [10, 23]). c`size` returns the arity of a circuit.

Quantum Operations. r`size` returns the arity of a quantum register. $\mathtt{x}\lhd\mathtt{N}$ evaluates the application of the circuit `N` to the quantum state stored in `x`, then it stores the resulting state in `x`. Roughly, $\mathsf{meas}^{\mathtt{N}}\,\mathtt{x}$ measures `N` qubits of a quantum state which `x` stores (and, update such state, in accordance with the quantum measurement rules).

2.1 Typing system

A *base* is a finite list $\mathtt{x}_1:\sigma_1,\ldots,\mathtt{x}_n:\sigma_n$ that we manage as a set such that $\mathtt{x}_i\neq\mathtt{x}_j$ for every $i\neq j$. If $B=\mathtt{x}_1:\sigma_1,\ldots,\mathtt{x}_n:\sigma_n$, then $\mathrm{dom}(B)=\{\mathtt{x}_1,\ldots,\mathtt{x}_n\}$ and $\mathrm{ran}(B)=\{\sigma_1,\ldots,\sigma_n\}$. The extension of B with $\mathtt{x}:\sigma$ is denoted $B\cup\{\mathtt{x}:\sigma\}$ where, w.l.o.g., we assume such that $\mathtt{x}\notin\mathrm{dom}(B)$.

Definition 1. *A term is* well-typed *whenever it is the conclusion of a finite derivation built with the rules in Table 1.*

The rules (tv), (tn), (ts), (tp), (tab), (tap), (tm) and (ti) come, quite directly, from PCF. We just remark that (ti) extends the conditional in order to operate not only on numerals, but also on other ground types (circuits, commands, classical and quantum variables).

The rules (tk), (tc), (tw), (tA), (tR) and $(tcnw)$ are the typical imperative extensions that Idealized Algol contains. The rules (tk) and (tw) types the standard imperative commands exactly as in Idealized Algol. The rule (tc) serves to consistently *concatenate* commands and to `attach` commands to expressions typed with other ground types. It is worth to remark that, this second use allows to include side-effects in the evaluation of expressions. The rule (tA) gives a type to the assignment of a classical

variable. The rule (tR) gives a type to the result of *reading* a classical variable. As in Idealized Algol, the rule $(tcnw)$ allows to declare local variables. If $\mathtt{x} : \mathsf{cVar}$, then $\mathsf{cnew}^{\mathtt{N}}\ \mathtt{x}\ \mathtt{in}\ \mathtt{M}$ makes a new instance of a classical register available. It is a binder that binds the variable name $\mathtt{x}$ and whose scope is $\mathtt{M}$. The initial value of $\mathtt{x}$ depends on $\mathtt{N}$.

The rules $(tc1)$, $(tc2)$, $(tc3)$ give a type to basic gates and to compositions of quantum circuits. The rule (tmc) gives a type to a meta-operations on quantum circuits. The rules (tsc) and (tsv) give a type to the operations that return arity values. The rule (tC) gives a type to the application of a quantum circuit to a quantum state. The rule (tM) gives a type to the measurement of a quantum state stored in a quantum variable. In accordance with $(tcnw)$, the rule $(tqnw)$ allows to declare local variables. If $\mathtt{x} : \mathsf{qVar}$, then $\mathsf{qnew}^{\mathtt{N}}\ \mathtt{x}\ \mathtt{in}\ \mathtt{M}$ gives a new instance of quantum store. The variable name $\mathtt{x}$ is a binder whose scope is $\mathtt{M}$. Moreover, w.l.o.g., our assumptions are that the initial value of the quantum variable always is the classical state zero and that $\mathtt{N}$ provides the numbers of qubits of the associated register.

The type system enjoys some basic properties. First, if $B \vdash \mathtt{M} : \tau$, then $\mathrm{FV}(\mathtt{M}) \subseteq dom(B)$. Second, if $B \vdash \mathtt{M} : \tau$, then $B' \vdash \mathtt{M} : \tau$ where B' is the restriction of B to $FV(\mathtt{M})$; Third, if $B \vdash \mathtt{M} : \tau$ and $dom(B) \cap dom(B') = \emptyset$, then $B \cup B' \vdash \mathtt{M} : \tau$, which is the weakening of the base. Generation lemmas hold too, however, our focus will mainly be on the dynamics of the typing system than on its logical properties.

Proposition 1 (Substitution)**.** *Let* $B, \mathtt{x} : \sigma \vdash \mathtt{M} : \tau$.
If $B' \vdash \mathtt{N} : \sigma$ *and* $dom(B) \cap dom(B') = \emptyset$*, then* $B \cup B' \vdash \mathtt{M}[\mathtt{N}/\mathtt{x}] : \tau$.

Proof. Standard, by reasoning inductively on the given typing derivation. □

Standard Lemmas of typed subject expansions can be straightforwardly proved too.

Example 1. *Let* $\mathtt{N} : \mathsf{Nat}$ *be a term whose unique free variable is* $\mathtt{r} : \mathsf{qVar}$*. Also, let* $\mathtt{Not}$ *denote the not-gate (a.k.a. Pauli-X gate) and let* $\mathtt{Id}$ *denote the identity gate, both with arity* 1*. Let* $\mathtt{M}$ *denote the term* $(\mathtt{r} \lhd ((\mathtt{Not} \parallel \mathtt{Id}) \parallel \mathtt{Not}))\mathtt{;N}$*. Then,* $\mathtt{M}$ *has type* Nat *by using* (tc) *(just after an application of* (tC)*). Moreover, by means of* $(tqnew)$*, we can conclude* $\vdash \mathsf{qnew}^{3}\ \mathtt{r}\ \mathtt{in}\ \mathtt{M} : \mathsf{Nat}$*. Anticipating the semantics, this means that the variable* $\mathtt{r}$ *in the sub-term* $\mathtt{N}$ *is associated to a quantum register of* 3 *qubits initialized to* $|101\rangle$. □

3 Evaluation Semantics

The evaluation of IQu focuses on programs, i.e. closed terms whose type is ground. Like Idealized Algol, the operational semantics of IQu uses an auxiliary function to record values associated to variables whose type is cVar or qVar. In addition, variables of type qVar must record the number of available qubits.

Definition 2 (Stores)**.** *A store* $\overline{\mathtt{s}}$ *is the disjoint union of two partial functions, both defined on a finite domain. The first one has* cVar *as domain and numerals as co-domain. The second function has* qVar *as domain and pairs (quantum states, numerals) as co-domain, where the second component counts the qubits of the first component. If* $\mathtt{x} : \mathsf{cVar}$*, then* $\overline{\mathtt{s}}(\mathtt{x})$ *denotes the numeral stored in* $\mathtt{x}$*. If* $\mathtt{x} : \mathsf{qVar}$*, then* $\overline{\mathtt{s}}(\mathtt{x})$ *denotes the state stored in* $\mathtt{x}$ *and* $\overline{\mathtt{s}}_\sharp(\mathtt{x})$ *the number of qubits of the state* $\overline{\mathtt{s}}(\mathtt{x})$*. The domain of definition of* $\overline{\mathtt{s}}$ *is denoted* $dom(\overline{\mathtt{s}})$*, i.e. it is a set of variable names.*

We denote $\overline{\mathtt{s}}|_{\mathtt{x}}$ *the store that behaves like* $\overline{\mathtt{s}}$ *everywhere, except on* $\mathtt{x}$ *where it is undefined. Let* $\overline{\mathtt{s}}$ *be a store and let* $\mathtt{x} : \mathsf{cVar}$*; then* $\overline{\mathtt{s}}[\mathtt{x} \leftarrow \underline{\mathtt{k}}]$ *denotes a store that behaves like* $\overline{\mathtt{s}}$ *everywhere except than on* $\mathtt{x}$ *to which it associates* $\underline{\mathtt{k}}$*. Let* $\overline{\mathtt{s}}$ *be a store and let* $\mathtt{x} : \mathsf{qVar}$*; then* $\overline{\mathtt{s}}[\mathtt{x} \overset{\underline{\mathtt{k}}}{\Leftarrow} |\phi\rangle]$ *denotes a store that behaves like* $\overline{\mathtt{s}}$ *everywhere except than on* $\mathtt{x}$ *on which we have that* $\overline{\mathtt{s}}(\mathtt{x}) = |\phi\rangle$ *and* $\overline{\mathtt{s}}_\sharp(\mathtt{x}) = \underline{\mathtt{k}}$. □

We conventionally assume that $\mathtt{C}$ ranges over the strings that describe evaluated circuit expressions, i.e. parallel and series composition of names for gates (cf. Theorem 2). Moreover, $\mathtt{V}$ ranges over numerals, strings that describe circuits, names of registers and the command $\mathtt{skip}$.

$$\frac{}{\overline{\mathtt{s}},\underline{\mathtt{n}} \Downarrow_1 \overline{\mathtt{s}},\underline{\mathtt{n}}}\ (en) \qquad \frac{\overline{\mathtt{s}},\mathtt{M} \Downarrow_\alpha \overline{\mathtt{s}}',\underline{\mathtt{n}}}{\overline{\mathtt{s}},\mathbf{s}(\mathtt{M}) \Downarrow_\alpha \overline{\mathtt{s}}',\underline{\mathtt{n}+1}}\ (es) \qquad \frac{\overline{\mathtt{s}},\mathtt{M} \Downarrow_\alpha \overline{\mathtt{s}}',\underline{\mathtt{n}+1}}{\overline{\mathtt{s}},\mathbf{p}(\mathtt{M}) \Downarrow_\alpha \overline{\mathtt{s}}',\underline{\mathtt{n}}}\ (ep)$$

$$\frac{\overline{\mathtt{s}},\mathtt{M}[\mathtt{N}/\mathtt{x}]\mathtt{P}_1 \cdots \mathtt{P}_m \Downarrow_\alpha \overline{\mathtt{s}}',\mathtt{V}}{\overline{\mathtt{s}},(\lambda \mathtt{x}.\mathtt{M})\mathtt{N}\mathtt{P}_1 \cdots \mathtt{P}_m \Downarrow_\alpha \overline{\mathtt{s}}',\mathtt{V}}\ (e\beta) \qquad \frac{\overline{\mathtt{s}},\mathtt{M}(\mathtt{YM})\mathtt{P}_1 \cdots \mathtt{P}_m \Downarrow_\alpha \overline{\mathtt{s}}',\mathtt{V}}{\overline{\mathtt{s}},\mathtt{YM}\mathtt{P}_1 \cdots \mathtt{P}_m \Downarrow_\alpha \overline{\mathtt{s}}',\mathtt{V}}\ (e\mathtt{Y})$$

$$\frac{\overline{\mathtt{s}},\mathtt{M} \Downarrow_\alpha \overline{\mathtt{s}}',\underline{0} \quad \overline{\mathtt{s}}',\mathtt{L} \Downarrow_{\alpha'} \overline{\mathtt{s}}'',\mathtt{V}}{\overline{\mathtt{s}},\mathtt{if}\ \mathtt{M}\ \mathtt{L}\ \mathtt{R} \Downarrow_{\alpha\cdot\alpha'} \overline{\mathtt{s}}'',\mathtt{V}}\ (e\mathrm{if}_l) \qquad \frac{\overline{\mathtt{s}},\mathtt{M} \Downarrow_\alpha \overline{\mathtt{s}}',\underline{\mathtt{n}+1} \quad \overline{\mathtt{s}}',\mathtt{R} \Downarrow_{\alpha'} \overline{\mathtt{s}}'',\mathtt{V}}{\overline{\mathtt{s}},\mathtt{if}\ \mathtt{M}\ \mathtt{L}\ \mathtt{R} \Downarrow_{\alpha\cdot\alpha'} \overline{\mathtt{s}}'',\mathtt{V}}\ (e\mathrm{if}_r)$$

$$\frac{}{\overline{\mathtt{s}},\mathtt{skip} \Downarrow_1 \overline{\mathtt{s}},\mathtt{skip}}\ (esk) \qquad \frac{\overline{\mathtt{s}},\mathtt{M} \Downarrow_\alpha \overline{\mathtt{s}}',\mathtt{skip} \quad \overline{\mathtt{s}}',\mathtt{N} \Downarrow_{\alpha'} \overline{\mathtt{s}}'',\mathtt{V}}{\overline{\mathtt{s}},\mathtt{M};\mathtt{N} \Downarrow_{\alpha\cdot\alpha'} \overline{\mathtt{s}}'',\mathtt{V}}\ (e;) \qquad \frac{}{\overline{\mathtt{s}},\mathtt{x} \Downarrow_1 \overline{\mathtt{s}},\mathtt{x}}\ (eVar)$$

$$\frac{\overline{\mathtt{s}},\mathtt{M} \Downarrow_\alpha \overline{\mathtt{s}}',\underline{\mathtt{n}+1} \quad \overline{\mathtt{s}}',\mathtt{N} \Downarrow_{\alpha'} \overline{\mathtt{s}}'',\mathtt{skip} \quad \overline{\mathtt{s}}'',\mathtt{while}\,\mathtt{M}\,\mathtt{do}\,\mathtt{N} \Downarrow_{\alpha''} \overline{\mathtt{s}}''',\mathtt{skip}}{\overline{\mathtt{s}},\mathtt{while}\,\mathtt{M}\,\mathtt{do}\,\mathtt{N} \Downarrow_{\alpha\cdot\alpha'\cdot\alpha''} \overline{\mathtt{s}}''',\mathtt{skip}}\ (ew1)$$

$$\frac{\overline{\mathtt{s}},\mathtt{M} \Downarrow_\alpha \overline{\mathtt{s}}',0}{\overline{\mathtt{s}},\mathtt{while}\,\mathtt{M}\,\mathtt{do}\,\mathtt{N} \Downarrow_\alpha \overline{\mathtt{s}}',\mathtt{skip}}\ (ew0) \qquad \frac{\overline{\mathtt{s}},\mathtt{N} \Downarrow_\alpha \overline{\mathtt{s}}',\underline{\mathtt{n}} \quad \overline{\mathtt{s}}',\mathtt{M} \Downarrow_{\alpha'} \overline{\mathtt{s}}'',\mathtt{x}}{\overline{\mathtt{s}},\mathtt{M} := \mathtt{N} \Downarrow_{\alpha\cdot\alpha'} \overline{\mathtt{s}}''[\mathtt{x} \leftarrow \underline{\mathtt{n}}],\mathtt{skip}}\ (ecA)$$

$$\frac{\overline{\mathtt{s}},\mathtt{M} \Downarrow_\alpha \overline{\mathtt{s}}',\mathtt{x}}{\overline{\mathtt{s}},\mathtt{read}\,\mathtt{M} \Downarrow_\alpha \overline{\mathtt{s}}',\overline{\mathtt{s}}'(\mathtt{x})}\ ecR \qquad \frac{\overline{\mathtt{s}},\mathtt{N} \Downarrow_\alpha \overline{\mathtt{s}}',\underline{\mathtt{n}} \quad \overline{\mathtt{s}}'[\mathtt{x} \leftarrow \underline{\mathtt{n}}],\mathtt{M} \Downarrow_{\alpha'} \overline{\mathtt{s}}'',\mathtt{V}}{\overline{\mathtt{s}},\mathtt{cnew}^{\mathtt{N}}\ \mathtt{x}\ \mathtt{in}\ \mathtt{M} \Downarrow_{\alpha\cdot\alpha'} \overline{\mathtt{s}}''\restriction_{\mathtt{x}},\mathtt{V}}\ ecN$$

$$\frac{\overline{\mathtt{s}},\mathtt{M}_0 \Downarrow_\alpha \overline{\mathtt{s}}',\mathtt{C}_0 \quad \overline{\mathtt{s}}',\mathtt{M}_1 \Downarrow_{\alpha'} \overline{\mathtt{s}}'',\mathtt{C}_1 \qquad \mathsf{wr}(\mathtt{C}_0),\ \mathsf{wr}(\mathtt{C}_1) \text{ are defined, and } \mathsf{wr}(\mathtt{C}_0) = \mathsf{wr}(\mathtt{C}_1)}{\overline{\mathtt{s}},\mathtt{M}_0 :: \mathtt{M}_1 \Downarrow_{\alpha\cdot\alpha'} \overline{\mathtt{s}}'',\mathtt{C}_0 :: \mathtt{C}_1}\ (e\mathrm{u}_2)$$

$$\frac{}{\overline{\mathtt{s}},\mathtt{U}^{\underline{\mathtt{k}}} \Downarrow_1 \overline{\mathtt{s}},\mathtt{U}^{\underline{\mathtt{k}}}}\ (e\mathrm{u}_1) \quad \frac{\overline{\mathtt{s}},\mathtt{M}_0 \Downarrow_\alpha \overline{\mathtt{s}}',\mathtt{C}_0 \quad \overline{\mathtt{s}}',\mathtt{M}_1 \Downarrow_{\alpha'} \overline{\mathtt{s}}'',\mathtt{C}_1}{\overline{\mathtt{s}},\mathtt{M}_0 \parallel \mathtt{M}_1 \Downarrow_{\alpha\cdot\alpha'} \overline{\mathtt{s}}'',\mathtt{C}_0 \parallel \mathtt{C}_1}\ (e\mathrm{u}_3) \quad \frac{\overline{\mathtt{s}},\mathtt{M} \Downarrow_\alpha \overline{\mathtt{s}}',\mathtt{U} \quad (\ddagger \mathtt{U}) = \mathtt{U}'}{\overline{\mathtt{s}},\mathtt{reverse}\,\mathtt{M} \Downarrow_\alpha \overline{\mathtt{s}}',\mathtt{U}'}\ (e\mathrm{r}_1)$$

$$\frac{\overline{\mathtt{s}},\mathtt{M} \Downarrow_\alpha \overline{\mathtt{s}}',\mathtt{C}_0 :: \mathtt{C}_1 \quad \overline{\mathtt{s}}',\mathtt{reverse}\,\mathtt{C}_0 \Downarrow_{\alpha'} \overline{\mathtt{s}}'',\mathtt{C}_0' \quad \overline{\mathtt{s}}'',\mathtt{reverse}\,\mathtt{C}_1 \Downarrow_{\alpha''} \overline{\mathtt{s}}''',\mathtt{C}_1'}{\overline{\mathtt{s}},\mathtt{reverse}\,\mathtt{M} \Downarrow_{\alpha\cdot\alpha'\cdot\alpha''} \overline{\mathtt{s}}''',\mathtt{C}_1' :: \mathtt{C}_0'}\ (e\mathrm{r}_2)$$

$$\frac{\overline{\mathtt{s}},\mathtt{M} \Downarrow_\alpha \overline{\mathtt{s}}',\mathtt{C}_0 \parallel \mathtt{C}_1 \quad \overline{\mathtt{s}}',\mathtt{reverse}\,\mathtt{C}_0 \Downarrow_{\alpha'} \overline{\mathtt{s}}'',\mathtt{C}_0' \quad \overline{\mathtt{s}}'',\mathtt{reverse}\,\mathtt{C}_1 \Downarrow_{\alpha''} \overline{\mathtt{s}}''',\mathtt{C}_1'}{\overline{\mathtt{s}},\mathtt{reverse}\,\mathtt{M} \Downarrow_{\alpha\cdot\alpha'\cdot\alpha''} \overline{\mathtt{s}}''',\mathtt{C}_0' \parallel \mathtt{C}_1'}\ (e\mathrm{r}_3)$$

$$\frac{\overline{\mathtt{s}},\mathtt{M} \Downarrow_\alpha \overline{\mathtt{s}}',\mathtt{C}}{\overline{\mathtt{s}},c\mathtt{size}\,\mathtt{M} \Downarrow_\alpha \overline{\mathtt{s}},\mathsf{wr}(\mathtt{C})}\ (e\mathrm{csz}) \qquad \frac{}{\overline{\mathtt{s}},r\mathtt{size}\,\mathtt{x} \Downarrow_1 \overline{\mathtt{s}},\overline{\mathtt{s}}_\sharp(\mathtt{x})}\ (e\mathrm{rsz})$$

$$\frac{\overline{\mathtt{s}},\mathtt{N} \Downarrow_\alpha \overline{\mathtt{s}}',\mathtt{C} \quad \overline{\mathtt{s}}',\mathtt{M} \Downarrow_{\alpha'} \overline{\mathtt{s}}'',\mathtt{x} \quad \mathsf{wr}(\mathtt{C}) = \overline{\mathtt{s}}_\sharp(\mathtt{x}) = \underline{\mathtt{n}}}{\overline{\mathtt{s}},\mathtt{M} \lhd \mathtt{N} \Downarrow_{\alpha\cdot\alpha'} \overline{\mathtt{s}}''[\mathtt{x} \xleftarrow{\overline{\mathtt{s}}''_\sharp(\mathtt{x})} \mathrm{cEval}^{\underline{\mathtt{n}}}(\mathtt{C})(\overline{\mathtt{s}}'(\mathtt{x}))],\mathtt{skip}}\ (eqA_0) \quad \frac{\overline{\mathtt{s}},\mathtt{N} \Downarrow_\alpha \overline{\mathtt{s}}',\mathtt{C} \quad \overline{\mathtt{s}}',\mathtt{M} \Downarrow_{\alpha'} \overline{\mathtt{s}}'',\mathtt{x} \quad \mathsf{wr}(\mathtt{C}) \neq \overline{\mathtt{s}}_\sharp(\mathtt{x})}{\overline{\mathtt{s}},\mathtt{M} \lhd \mathtt{N} \Downarrow_{\alpha\cdot\alpha'} \overline{\mathtt{s}}'',\mathtt{skip}}\ (eqA_1)$$

$$\frac{\begin{array}{l}\overline{\mathtt{s}},\mathtt{N} \Downarrow_\alpha \overline{\mathtt{s}}',\underline{\mathtt{k}} \qquad \overline{\mathtt{s}}_\sharp(\mathtt{x}) = \underline{\mathtt{n}} \\ \overline{\mathtt{s}}',\mathtt{M} \Downarrow_{\alpha'} \overline{\mathtt{s}}'',\mathtt{x} \quad (m,|\phi\rangle,\alpha'') \in \mathrm{pMeas}^{\underline{\mathtt{n}}}(\overline{\mathtt{s}}''(\mathtt{r}),k)\end{array}}{\overline{\mathtt{s}},\mathsf{meas}^{\mathtt{N}}\,\mathtt{M} \Downarrow_{\alpha\cdot\alpha'\cdot\alpha''} \overline{\mathtt{s}}''[\mathtt{x} \leftarrow |\phi\rangle],\underline{m}}\ (eqM) \quad \frac{\overline{\mathtt{s}},\mathtt{N} \Downarrow_\alpha \overline{\mathtt{s}}',\underline{\mathtt{n}} \quad \overline{\mathtt{s}}' \cup \{\mathtt{x} \xleftarrow{\underline{\mathtt{n}}} 0\},\mathtt{M} \Downarrow_{\alpha'} \overline{\mathtt{s}}'',\mathtt{V}}{\overline{\mathtt{s}},\mathsf{qnew}^{\mathtt{N}}\ \mathtt{x}\ \mathtt{in}\ \mathtt{M} \Downarrow_{\alpha\cdot\alpha'} \overline{\mathtt{s}}''\restriction_{\mathtt{x}},\mathtt{V}}\ (eqN)$$

Table 2: Operational Semantics.

Definition 3 (Operational Evaluation)**.** *Let* $\mathtt{x}_1 : \mathsf{cVar}\ldots,\mathtt{x}_n : \mathsf{cVar},\mathtt{z}_1 : \mathsf{qVar}\ldots,\mathtt{z}_m : \mathsf{qVar} \vdash \mathtt{M} : \beta$ *where* $n,m \geq 0$ *and* $\beta \in \{\mathsf{Nat},\mathsf{circ},\mathsf{cmd},\mathsf{cVar},\mathsf{qVar}\}$*. If* $\overline{\mathtt{s}}$ *is a store such that* $\{\mathtt{x}_1,\ldots,\mathtt{x}_n\} \subseteq \mathsf{cVar} \cap dom(\overline{\mathtt{s}})$ *and* $\{\mathtt{z}_1,\ldots,\mathtt{x}_n\} \subseteq \mathsf{qVar} \cap dom(\overline{\mathtt{s}})$*, then the evaluation semantics of* IQu *proves formal statements* $\overline{\mathtt{s}},\mathtt{M} \Downarrow_\alpha \overline{\mathtt{s}}',\mathtt{V}$ *that we obtain as conclusion of a (finite) derivation* $\mathscr{D}$ *built with the rules in Table 2. As expected,* $\alpha \in (0,1]$ *is the probability to obtain* $\mathscr{D}$*.* □

The rules (en), (es), (ep), $(e\beta)$, $(e\mathtt{Y})$, $(e\mathtt{if}_l)$, $(e\mathtt{if}_r)$ at the top of Table 2 are standard, that are used in the evaluation of PCF (e.g., see [19]). They are enriched by a store that can be eventually used in the evaluation of their sub-terms (involving side-effects).

The rules (esk), $(e;)$, $(eVar)$, $(ew1)$, $(ew0)$, (ecA), (ecR), (ecN) in the middle of Table 2 formalize the standard evaluation of first order references and of usual imperative instructions that we find in Idealized Algol. The evaluation (ecA) begins by evaluating the expression whose result must be stored. The right-hand expression is expected to yield an classical variable. The rule (ecA) is the only one that changes the content of classical variable. W.l.o.g., we assume that $\mathtt{x} \notin \mathrm{dom}(\overline{\mathtt{s}})$ holds for the rule (ecN). Otherwise, we had to replace $\overline{\mathtt{s}}''\restriction_{\mathtt{x}},\mathtt{V}$ with $\overline{\mathtt{s}}''[\mathtt{x} \leftarrow \overline{\mathtt{s}}(\mathtt{x})],\mathtt{V}$ in its conclusion. Finally, we observe that only the rule that changes the classical variables in the domain of the store is (ecN).

Example 2.
- *Let* $\mathtt{P}$ *be the well-typed term* $B \cup \{\mathtt{x} : \mathsf{cVar}\} \vdash \mathtt{if}\,(\mathsf{read}\,\mathtt{x})\,\mathtt{M}_0\,\mathtt{M}_1 : \beta$ *such that* $B \cup \{\mathtt{x} : \mathsf{cVar}\} \vdash \mathtt{M}_i : \beta$ $(i = 0,1)$*. Let us evaluate* $\mathtt{P}$ *by using the store* $\overline{\mathtt{s}}$*: if* $\overline{\mathtt{s}}(\mathtt{x}) = 0$ *then we have to evaluate* $\mathtt{M}_0$ *(and we ignore* $\mathtt{M}_1$*), otherwise we evaluate* $\mathtt{M}_1$*.*
- *In the above term, let* $\mathtt{M}_0 = \mathtt{z}_0$*,* $\mathtt{M}_1 = \mathtt{z}_1$*, and* $\beta = \mathsf{cVar}$*. Starting with the store* $\overline{\mathtt{s}}[\mathtt{x} \leftarrow 0]$*, there is a unique derivation describing the evaluation of* $(\mathtt{if}\ \mathsf{read}(\mathtt{x})\,\mathtt{z}_0\,\mathtt{z}_1) := 5$*. This derivation concludes* $\overline{\mathtt{s}}[\mathtt{x} \leftarrow 0],(\mathtt{if}\ \mathsf{read}(\mathtt{x})\,\mathtt{z}_0\,\mathtt{z}_1) := 5 \Downarrow_1 \overline{\mathtt{s}}[\mathtt{x} \leftarrow 0,\mathtt{z}_0 \leftarrow 5],\mathtt{skip}$*.* □

Extending Idealized Algol The remaining rules formalizes our original extension of IQu. Rules about circuit evaluations are inspired by similar operator of qPCF (see [22]).

It is worth to remark that some of these rules rest on some auxiliary definitions (cf. Definitions 4, 5 and 6). Definition 4 has been formalized only for simplicity reasons: we isolated the function wr that counts the number of wires of an evaluated quantum circuit (since it is used in many rules of Table 2). Definitions 5 and 6 formalize the quantum co-processor as an external black-box.

The rules $(e\mathtt{u}_1)$, $(e\mathtt{u}_2)$, $(e\mathtt{u}_3)$ are used to bring the evaluation of a circuit on subexpressions, in order to reach (possible) side-effects (assignments and measurements) embedded in it. The rule $(e\mathtt{u}_2)$ exploits the function wr here below. The rule $(e\mathtt{u}_2)$ applies only when the wr yields the same value on the two circuit components. On the other hand, $\|$ is not subject to arity restriction (cf. rule $(e\mathtt{u}_3)$).

Definition 4. wr *is a partial function from circuits to numerals. It is defined by cases as follows:*

- $\mathsf{wr}(\mathtt{U}^{\underline{\mathtt{k}}}) = \underline{\mathtt{k}}$*;*
- *if* $\mathsf{wr}(\mathtt{M}_0)$*,* $\mathsf{wr}(\mathtt{M}_1)$ *are defined and* $\mathsf{wr}(\mathtt{M}_0) = \mathsf{wr}(\mathtt{M}_1)$ *then* $\mathsf{wr}(\mathtt{M}_0 :: \mathtt{M}_1) = \mathsf{wr}(\mathtt{M}_0)$*;*
- *if* $\mathsf{wr}(\mathtt{M}_0)$*,* $\mathsf{wr}(\mathtt{M}_1)$ *are defined then* $\mathsf{wr}(\mathtt{M}_0 \parallel \mathtt{M}_1) = \mathsf{wr}(\mathtt{M}_0) + \mathsf{wr}(\mathtt{M}_1)$*.*

The function is undefined in all other cases. □

It is easy to check that evaluated circuits (viz. circuits resulting from the evaluation of circuit expression) are strings of the grammar $\mathtt{C} ::= \mathtt{U}^{\underline{\mathtt{k}}} \mid \mathtt{C} :: \mathtt{C} \mid \mathtt{C} \parallel \mathtt{C}$ for which $\mathsf{wr}(\mathtt{C})$ is defined. Note that: (i) if $\mathtt{C}$ is $\mathtt{C}_0 :: \mathtt{C}_1$, then $\mathsf{wr}(\mathtt{C}_0) = \mathsf{wr}(\mathtt{C}_1)$, cf. rule $(e\mathtt{u}_2)$; and, (ii) if $\mathtt{M}$ is a circuit expression such that $\mathsf{wr}(\mathtt{M})$ is undefined then its evaluation diverges.

We remark that the evaluation of circuits evolves rightward once the possible side-effects in sub-terms are concerned. We mean that $(e\mathtt{u}_2)$ and $(e\mathtt{u}_3)$ update the store $\overline{\mathtt{s}}'$ by first evaluating the side-effects

in expression of the left-hand circuit and, then, by evaluating the side-effects in the expression of the right-hand circuit.

Recent quantum programming languages [8, 22, 23, 27, 28] include the possibility to manipulate quantum circuits and, in particular, of reversing circuits as originally advocated by [10]. `reverse` is expected to produce the adjoint circuit of its input: it is implemented by rewiring gates in reverse order (by means of rules (er_1), (er_2) and (er_3)) and, then, by replacing each gate by its adjoint. Its definition rests on the choice of a total endo-function (mapping each gate of arity k to a gate of arity k) that we denote with the symbol ‡. As usual, we assume that |Qu is endowed with a universal set of gates and that ‡ gives back an adjoint gate for each gate. If those latter assumptions do not hold, then ‡ can be chosen as the identity (so that, `reverse` does not reverse anymore the corresponding quantum transformations, but just rewires in reverse order).

The rule $(ecsz)$ yields the arity of the quantum (evaluated) circuit. The rule $(ersz)$ yields the number of qubits that the quantum register it involves stores.

Quantum State Updates and Quantum Measurements

We refer to [14] as a standard and comprehensive reference about quantum computation. Here we just recall what we need to introduce the interaction with quantum co-processors. The information of $n \in \mathbb{N}$ qubits is usually formalized by means of a normalized vector in $\mathscr{H}^n$ which is a Hilbert space of dimension $N = 2^n$ with orthonormal basis:

$$\overbrace{|\underbrace{0\ldots0}_{n}\rangle,\ldots,|\underbrace{1\ldots1}_{n}\rangle}^{N} \ .$$

The binary representation $x^\flat$ of any value x in the interval $[0,\ldots,N-1]$ identifies a vector in $\mathscr{H}^n$. The binary representation is handy to represent every state $|\psi\rangle$ of the Hilbert space as a linear combination:

$$|\psi\rangle = c_0|0^\flat\rangle + \ldots + c_{N-1}|(N-1)^\flat\rangle$$

with $c_0,\ldots,c_{N-1} \in \mathbb{C}$. Quantum transformations and measurement can transform quantum states [34, 35, 13]. We recall that a measurement reduces a quantum state partially, or totally, to a classical state. More precisely, given a state $|\phi\rangle \in \mathscr{H}^n$, we can measure a subset of qubits in $|\phi\rangle$ (i.e. a partial measurement). The result of the measurement is a residual state vector with a given probability (cf. Definition 6).

Our co-processor is a black-box, so we can formalize the interaction of |Qu with the quantum co-processor in an abstract way. We update the quantum state associated to a variable that can be updated by means of a function (see the next definition) that maps a circuit in the corresponding unitary transformation.

Definition 5. *Let $Circ^n$ be the set of evaluated circuits with type* circ*, with arity $\underline{\mathtt{n}}$ such that $N = 2^n$. Let $\mathscr{H}^n$ be a Hilbert space of finite dimension N. Let $\{|\varphi_i\rangle\}$ be an orthonormal basis on $\mathscr{H}^n$ and let $\mathscr{H}^n \to \mathscr{H}^n$ be the set of unitary operators on $\mathscr{H}^n$. The map from evaluated circuits to their corresponding unitary transformations is $cEval^n : Circ^n \to (\mathscr{H}^n \to \mathscr{H}^n)$, which we define as follows:*

- $cEval^n(\mathtt{U}^{\underline{\mathtt{n}}}) ::= \mathbf{U}$ *where* $\mathbf{U} : \mathscr{H}^n \to \mathscr{H}^n$ *is the unitary transformation associated to the gate* `U`*;*
- $cEval^n(\mathtt{C}_0 :: \mathtt{C}_1) ::= cEval^n(\mathtt{C}_1) \circ cEval^n(\mathtt{C}_0)$*;*
- $cEval^n(\mathtt{C}_0 \parallel \mathtt{C}_1) ::= cEval^{n_0}(\mathtt{C}_0) \otimes cEval^{n_1}(\mathtt{C}_1)$ *where* $\mathsf{wr}(\mathtt{C}_i) = \underline{\mathtt{n}}_i$ *and* $n = n_0 + n_1$. □

The rules (eqA_0) and (eqA_1) in Table 2 evaluate two expressions. If these evaluations converge then, the first one returns a circuit `C` and the second one returns a quantum variable `x` . If $\mathsf{wr}(\mathtt{C}) = \overline{\mathtt{s}}_\sharp(\mathtt{x})$ then

the evaluation proceeds by rule (eqA_0) that uses the function of Definition 5 to update the corresponding quantum state; otherwise, the rule (eqA_1) forgot the circuit transformations, in order to ensure the type-safety.

IQu allows for *partial measurements* of an arbitrary subset with k qubits of a n-qubits state.

Definition 6. *For all $k,n \in \mathbb{N}$, let $n\!\upharpoonleft_k = k\%(n+1)$ and $n\!\upharpoonright_k = n-(k\%(n+1))$, thus $n\!\upharpoonleft_k + n\!\upharpoonright_k = n$ where $\%$ denotes the modulo arithmetic operation. If $x < 2^j$ then, we use $\flat^j(x)$ to denote the binary representations of x deployed on j bits (i.e. binary digits). Moreover, let $S(j) = \{\flat^j(x) | 0 \le x < 2^j\}$ and $i \cdot j$ to denote the juxtaposition of i and j. Following [9], we formalize $pMeas^n : \mathscr{H}^n \times \mathbb{N} \longrightarrow \wp\big(\mathbb{N} \times \mathscr{H}^n \times \mathbb{R}\big)$ as follows:*

$$pMeas^n(|\phi\rangle,k) = \left\{ \big(m^\natural, |\psi_m\rangle, p_m\big) \;\middle|\; \begin{array}{l} |\phi\rangle = \sum_{i\in S(n\upharpoonleft_k)} \sum_{j\in S(n\upharpoonright_k)} c_{i\cdot j}|i\rangle \otimes |j\rangle \text{ and,} \\ m \in S(n\upharpoonleft_k) \text{ s.t. } |\psi_m\rangle = \sum_{j\in S(n\upharpoonright_k)} \frac{c_{m\cdot j}}{\sqrt{p_m}}|m\rangle \otimes |j\rangle \\ \qquad\qquad \text{where } p_m = \sum_{j\in S(n\upharpoonright_k)} |c_{i\cdot j}|^2 \end{array} \right\}$$

where $x^\natural$ is the natural number encoded in the sequence of bits x. The first argument of $pMeas^n$ is a quantum state $|\phi\rangle$ of $\mathscr{H}^n$. The second argument is the number of qubits we want to measure, modulo $n+1$. The result of $pMeas^n(|\phi\rangle,k)$ is a set of triples. The first component of the triple is a partial measure executed on $|\phi\rangle$: its value $m \in \mathbb{N}$ is the measurement of its first $k\%(n+1)$ qubits. The second component is the collapsing state (still in $\mathscr{H}^n$) and it is obtained from $|\phi\rangle$ by collapsing its measured sub-state to m. The third component is the probability of measuring the value m. □

We remark that $\text{pMeas}^n(|\phi\rangle,0)$ measures 0 qbit and $\text{pMeas}^n(|\phi\rangle,n)$ measures all n qbits.

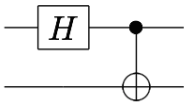
Example 3. *Let us consider the state $|\phi\rangle = \frac{1}{\sqrt{2}}|0010\rangle + \frac{1}{\sqrt{4}}|1011\rangle + \frac{1}{\sqrt{4}}|0010\rangle$ with 4-qubits in it. Let us measure its two first quantum bits. We remark that $pMeas^n(|\phi\rangle,k) = pMeas^n(|\phi\rangle,n+k)$, so that $pMeas^4(|\phi\rangle,2) = pMeas^4(|\phi\rangle,6)$. The partial observation of the two first qbits $pMeas^4(|\phi\rangle,2)$ has two possible outcomes. The first triple is $(m_0, \sqrt{\frac{2}{3}}|0010\rangle + \sqrt{\frac{1}{3}}|0001\rangle, \frac{3}{4})$ where $m_0^\flat = 00$. The second one is $(m_1, |1011\rangle, \frac{1}{4})$ where $m_1^\flat = 10$.* □

The rule (eqM) first evaluates two expressions in order to obtain a numeral $\underline{\mathtt{k}}$ and the quantum variable $\mathtt{x}$ subject of the measurement. The numeral $\underline{\mathtt{k}}$ (modulo the number of qubits stored in $\mathtt{x}$) identifies the number of qubits we want to measure. Then, it measures the quantum state that the store associates to by using the abstract function of Definition 6 which describes the expected behavior of the co-processor and that works in accordance with quantum computations laws. It is worth to note that the only non-deterministic rule of IQu evaluation is (eqM); that is, the probability-label in the conclusion of programs that never perform quantum measurement is 1. We here do not focus on any analysis about the probabilistic behavior of IQu. We simply remark that IQu is an extension of the probabilistic Idealized Algol in [5] and of the quantum language in [18]. The probabilistic operational equivalence notions can be easily adapted to IQu.

W.l.o.g., we assume that $\mathtt{x} \notin \text{dom}(\overline{\mathbb{s}})$ in (eqN). Otherwise $\overline{\mathbb{s}}''\restriction_{\mathtt{x}}, \mathtt{V}$ in its conclusion, should be replaced with $\overline{\mathbb{s}}''[\mathtt{x} \xleftarrow{\mathtt{n}} \overline{\mathbb{s}}(\mathtt{x})], \mathtt{V}$ in its conclusion. Both (ecN) and (eqN) are the only rules that change the domain of the store. A programmer can ask for a new quantum co-processor for manipulating a quantum state by means of (eqN) at run-time. We notice that no limit exists on the number of quantum registers that a program in IQu manipulates.

Example 4 (Bell state circuit)**.** *We show how IQu can encode a circuit description and evaluation.*

The Bell states (or EPR states or EPR pairs) are the simplest examples of entanglement of quantum states [14]. The following circuit applies a Hadamard gate on the top wire followed by a controlled-not:



It can be used to generate the Bell states by feeding it with a bases state $|00\rangle$, $|01\rangle$, $|10\rangle$, $|11\rangle$. *For example, the circuit returns the state* $\beta_{00} = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$ *on input* $|00\rangle$.

Let $H^{\underline{1}} : \mathsf{circ}$ *be the (unary) Hadamard gate,* $\mathtt{Id}^{\underline{1}} : \mathsf{circ}$ *be identity and* $\mathtt{CNOT}^{\underline{2}} : \mathsf{circ}$ *be the controlled-not operator. Let* Bell *be the closed term* $(H^{\underline{1}} \parallel \mathtt{Id}^{\underline{1}}) :: \mathtt{CNOT}^{\underline{2}}$ *that straightforwardly describes the above circuit. It is easy to check, by typing rules, that* $\vdash (H^{\underline{1}} \parallel \mathtt{Id}) :: \mathtt{CNOT}^{\underline{2}} : \mathsf{circ}$.

We can simulate an EPR experiment by using the (closed) term $\mathsf{qnew}^2\ \mathtt{r}\ \mathtt{in}\ (\mathtt{r} \lhd \mathsf{Bell}; \mathsf{meas}^{\underline{1}}\mathtt{r})$*: it requires that a fresh co-processor (locally, named* r*) is made available for the computation of its body, i.e.* $\mathtt{r} \lhd \mathsf{Bell}; \mathsf{meas}^{\underline{1}}\mathtt{r}$. *This latter applies the gates in* Bell *to the state stored in* r *and then performs a measurement.*
Clearly, $\vdash \mathsf{qnew}^2\ \mathtt{r}\ \mathtt{in}\ (\mathtt{r} \lhd \mathsf{Bell}; \mathsf{meas}^{\underline{1}}\mathtt{r}) : \mathsf{Nat}$ *and* $\{(\mathtt{r}, |00\rangle)\}, \mathtt{r} \lhd \mathsf{Bell} \Downarrow_1 \{(\mathtt{r}, \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle))\}, \mathtt{skip}$.
Moreover, since $\mathsf{pMeas}^2(\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle), 1) = \{(0, |00\rangle, \frac{1}{2}), (1, |11\rangle, \frac{1}{2})\}$, *either*

$$\{(\mathtt{r}, \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle))\}, \mathsf{meas}^{\underline{1}}\mathtt{r} \Downarrow_{\frac{1}{2}} \{(\mathtt{r}, |00\rangle)\}, \underline{0} \ \textit{or}\ \{(\mathtt{r}, \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle))\}, \mathsf{meas}^{\underline{1}}\mathtt{r} \Downarrow_{\frac{1}{2}} \{(\mathtt{r}, |11\rangle)\}, \underline{1}.$$

□

3.1 Type-safety

IQu enjoys of usual properties of programming languages such as Preservation and Progress [24]. These results follow by adapting, quite straightforwardly, the standard techniques used for PCF and Idealized Algol.

Theorem 1 (Preservation)**.** *Let* $\mathtt{x}_1 : \mathsf{cVar} \ldots, \mathtt{x}_n : \mathsf{cVar}, \mathtt{z}_1 : \mathsf{qVar} \ldots, \mathtt{z}_m : \mathsf{qVar} \vdash \mathtt{M} : \beta$ *be a term. Let* $\overline{\mathbb{s}}$ *be a store such that* $\{\mathtt{x}_1, \ldots, \mathtt{x}_n\} \subseteq \mathsf{cVar} \cap dom(\overline{\mathbb{s}})$ *and* $\{\mathtt{z}_1, \ldots, \mathtt{x}_n\} \subseteq \mathsf{qVar} \cap dom(\overline{\mathbb{s}})$.
If $\overline{\mathbb{s}}, \mathtt{M} \Downarrow_\alpha \overline{\mathbb{s}}', \mathtt{V}$ *then* $\mathtt{x}_1 : \mathsf{cVar} \ldots, \mathtt{x}_n : \mathsf{cVar}, \mathtt{z}_1 : \mathsf{qVar} \ldots, \mathtt{z}_m : \mathsf{qVar} \vdash \mathtt{V} : \beta$.

Proof. The proof is by induction on the derivation concluding $\overline{\mathbb{s}}, \mathtt{M} \Downarrow_\alpha \overline{\mathbb{s}}', \mathtt{V}$. The proof immediately holds on (en), (es), (ep) while it is true on $(e\beta)$, $(e\mathtt{Y})$ by arguments related to the inductive hypothesis and the subject reduction. The inductive hypothesis straightforwardly applies to $(e\mathrm{if}_l)$, $(e\mathrm{if}_r)$, so we can skip to consider the imperative part of the language. If the last rule is one among (esk), $(eVar)$, $(ew1)$, $(ew0)$, (ecA), (ecR) the proof is once again immediate. If the last rule is $(e;)$, (ecN) it is simple to apply the inductive argument. The rules $(e\mathrm{u}_1)$, $(e\mathrm{r}_1)$, $(e\mathrm{csz})$, $(e\mathrm{rsz})$, (eqA_0), (eqA_1), (eqM) do not pose any specific difficulties. Finally, the inductive principle applies also to $(e\mathrm{u}_2)$, $(e\mathrm{u}_3)$, $(e\mathrm{r}_2)$, $(e\mathrm{r}_3)$ and (eqN). □

Theorem 2 (Progress)**.** *Let* $\mathtt{x}_1 : \mathsf{cVar} \ldots, \mathtt{x}_n : \mathsf{cVar}, \mathtt{z}_1 : \mathsf{qVar} \ldots, \mathtt{z}_m : \mathsf{qVar} \vdash \mathtt{M} : \beta$ *be a term. Let* $\overline{\mathbb{s}}$ *be a store such that* $\{\mathtt{x}_1, \ldots, \mathtt{x}_n\} \subseteq \mathsf{cVar} \cap dom(\overline{\mathbb{s}})$ *and* $\{\mathtt{z}_1, \ldots, \mathtt{x}_n\} \subseteq \mathsf{qVar} \cap dom(\overline{\mathbb{s}})$.

1. *If* $\beta = \mathsf{Nat}$ *and* $\overline{\mathbb{s}}, \mathtt{M} \Downarrow_\alpha \overline{\mathbb{s}}', \mathtt{V}$ *then* V *is a numeral.*
2. *If* $\beta = \mathsf{circ}$ *and* $\overline{\mathbb{s}}, \mathtt{M} \Downarrow_\alpha \overline{\mathbb{s}}', \mathtt{C}$ *then* C *is a string of the grammar* $\mathtt{C} ::= \mathtt{U}^{\underline{\mathtt{k}}} \mid \mathtt{C} :: \mathtt{C} \mid \mathtt{C} \parallel \mathtt{C}$ *such that* $\mathsf{wr}(\mathtt{C})$ *is defined, and moreover if* C *has shape* $\mathtt{C}_0 :: \mathtt{C}_1$ *then* $\mathsf{wr}(\mathtt{C}_0) = \mathsf{wr}(\mathtt{C}_1)$.
3. *If* $\beta = \mathsf{cVar}$ *and* $\overline{\mathbb{s}}, \mathtt{M} \Downarrow_\alpha \overline{\mathbb{s}}', \mathtt{V}$ *then* V *is the name of a classical variable.*
4. *If* $\beta = \mathsf{qVar}$ *and* $\overline{\mathbb{s}}, \mathtt{M} \Downarrow_\alpha \overline{\mathbb{s}}', \mathtt{V}$ *then* V *is the name of a quantum variable.*
5. *If* $\beta = \mathsf{cmd}$ *and* $\overline{\mathbb{s}}, \mathtt{M} \Downarrow_\alpha \overline{\mathbb{s}}', \mathtt{V}$ *then* V *is* skip.

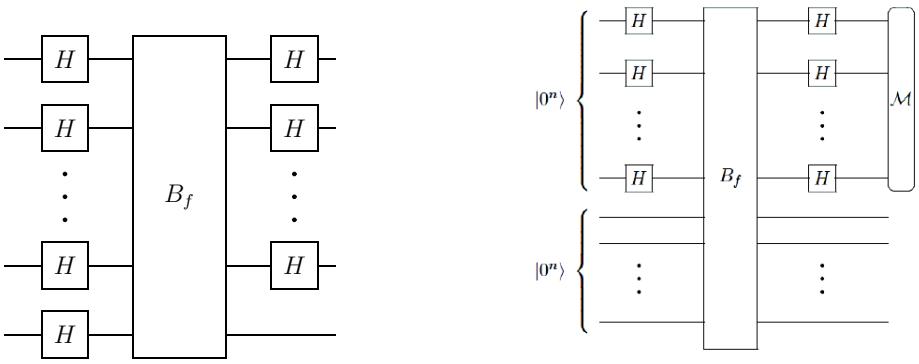


Table 3: Deutsch-Jozsa circuit (left) and the quantum subroutine of th Simon's algorithm (right).

Proof. All the proofs are by induction on the evaluation, proceeding by cases the subject of the last rule applied.

1. The proof is standard for the PCF core of IQu. The proof is immediate for (en), (es), (ep), (ecR), $(ecsz)$, $(ersz)$, (eqM). The proof for $(e\beta)$, $(e\mathtt{Y})$, $(e\mathrm{if}_l)$, $(e\mathrm{if}_r)$, $(e;)$, (ecN) and (eqN) follows from the inductive argument. The other cases are not possible because of the typing rules.
2. The proof is immediate for $(e\mathrm{u}_1)$, $(e\mathrm{r}_1)$. The proof for $(e\beta)$, $(e\mathtt{Y})$, $(e\mathrm{if}_l)$, $(e\mathrm{if}_r)$, $(e;)$, (ecN), $(e\mathrm{u}_2)$, $(e\mathrm{u}_3)$, $(e\mathrm{r}_2)$, $(e\mathrm{r}_3)$, (eqN) follows from the inductive argument. The other cases are not possible because of the typing rules.
3. The proof is immediate for $(eVar)$. The proof for $(e\beta)$, $(e\mathtt{Y})$, $(e\mathrm{if}_l)$, $(e\mathrm{if}_r)$, $(e;)$, (ecN), (eqN) follows from the inductive argument. The other cases are not possible because the typing rules.
4. The proof is immediate for $(eVar)$. The proof for $(e\beta)$, $(e\mathtt{Y})$, $(e\mathrm{if}_l)$, $(e\mathrm{if}_r)$, $(e;)$, (ecN), (eqN) follows from the inductive argument. The other cases are not possible because of the typing rules.
5. The proof is immediate for (esk), $(ew0)$, $(ew1)$, (ecA), (eqA_0), (eqA_1), The proof for rules $(e\beta)$, $(e\mathtt{Y})$, $(e\mathrm{if}_l)$, $(e\mathrm{if}_r)$, $(e;)$, (ecN), (eqN) follows from the inductive argument. The other cases are not possible because of the typing rules.

□

4 Examples

Two further examples of programming in IQu follow. One implements Deutsch-Jozsa. The other one is (a subroutine of) Simon's algorithm.

Example 5 (Deutsch-Jozsa Circuit)**.** *In this example we show how a IQu term can represent an infinite family of quantum programs which encode Deutsch-Jozsa algorithm [14]. Deutsch-Jozsa is a generalization of Deutsch algorithm that, given a black-box B_f implementing a function $f : \{0,1\} \to \{0,1\}$, determines whether f is constant or balanced*[1] *by means of a single call to B_f, something impossible in the classical case that requires two calls. Deutsch-Jozsa solves the parametrized version of the original problem because it applies to functions $f : \{0,1\}^n \to \{0,1\}$. The leftmost circuit in Table 3 is a possible implementation of Deutsch-Jozsa in IQu. When fed with a classical input state $|\underbrace{0\ldots0}_{n}1\rangle$, the output can be partially measured. Measuring the first n bits tells if the function f is constant or not. If all n qubits of*

[1] A function is balanced if exactly half of the inputs goes to 0 and, the other half, goes to 1.

such a unique measurement are 0, then f is constant. Otherwise, i.e., if at least one of the measurement outcomes is 1, then f is balanced. See [14] for further details.

Let $\mathtt{H}^{\underline{1}} : \mathsf{circ}$ *be the Hadamard gate and* $\mathtt{Id}^{\underline{1}} : \mathsf{circ}$ *be the Identity gate. We implement Deutsch-Jozsa in IQu by sequentially composing* M_1, $\mathtt{x}$ *and* M_3, *where* $\mathtt{x} : \mathsf{circ}$ *is expected to be substituted by the black-box circuit that implements the function f, while both* M_1 *and* M_3 *are defined in the coming lines.*

- *Let* M_{par} *be a term that applied to a circuit* $\mathtt{C} : \mathsf{circ}$ *and to a numeral* $\underline{\mathtt{n}}$ *puts* $n+1$ *copies of* $\mathtt{C}$ *in parallel. It is defined as* $\mathsf{M}_{par} = \lambda \mathtt{u}^{\mathsf{circ}}.\lambda \mathtt{k}^{\mathsf{Nat}}.Y\mathsf{W}_1\mathtt{uk} : \mathsf{circ} \to \mathsf{Nat} \to \mathsf{circ}$, *where* W_1 *is the term* $\lambda \mathtt{w}^{\sigma}.\lambda \mathtt{u}^{\mathsf{circ}}.\lambda \mathtt{k}^{\mathsf{Nat}}.\mathtt{if\ k\ (u)\ (u}\ \|\ \mathtt{(wupred(k)))}$ *whose type is* $\sigma \to \sigma$ *with* $\sigma = \mathsf{circ} \to \mathsf{Nat} \to \mathsf{circ}$.
- *The circuit* $\mathsf{M}_1 : \mathsf{circ}$ *is obtained by feeding the term* M_{par} *with two inputs: the (unary) Hadamard gate* $\mathtt{H}^{\underline{1}}$ *and the input dimension* $\mathtt{rsize(r)}$ *where r is a co-processor register with suitable dimension. It should be evident that it generates* $n+1$ *parallel copies of the gate* $\mathtt{H}^{\underline{1}}$.
- *The circuit* $\mathsf{M}_3 : \mathsf{circ}$ *can be defined as* $(\mathsf{M}_{par}\mathtt{H}^{\underline{1}}\,\mathtt{pred}(\mathtt{rsize}(\mathtt{r}))) \parallel \mathtt{Id}^{\underline{1}} : \mathsf{circ}$, *i.e. it is obtained by the parallel composition of the term* M_{par} *fed by the gate* $\mathtt{H}^{\underline{1}}$ *and the dimension* $\mathtt{pred}(\mathtt{rsize}(\mathtt{r}))$ *(generating n parallel copies of the gate* $\mathtt{H}^{\underline{1}}$*) and a single copy* $\mathtt{Id}^{\underline{1}}$ *of the identity gate.*

Fixed an arbitrary n, the generalization of Deutsch-Jozsa is obtained by using the quantum variable binder $\mathsf{qnew}^n\ \mathtt{r\ in}\ P$ *that makes the quantum variable* $\mathtt{r}$ *available in P. In this picture, it is necessary to recall that the local variable declaration* $\mathsf{qnew}^n\ \mathtt{r\ in}\ P$ *creates a quantum register which is fully initialized to 0 (Section 2). Since the expected input state of Deutsch-Jozsa is* $|\overbrace{0\ldots0}^{n}1\rangle$, *we define and use an initializing circuit* $\mathsf{M}_{\mathsf{init}} = (\mathsf{M}_{par}\mathtt{Id}^{\underline{1}}(\mathtt{pred}(\mathtt{rsize}(r)))) \parallel \mathtt{Not}^{\underline{1}} : \mathsf{circ}$ *that complements the last qubit, setting it to* 1. *Let* DJ^+ *be the circuit* $\mathsf{M}_{\mathsf{init}} :: \mathsf{M}_1 :: \mathtt{x} :: \mathsf{M}_3$. *The (parametric) IQu encoding of Deutsch-Jozsa can be defined as* $\lambda \mathtt{x}^{\mathsf{circ}}.\ \mathsf{qnew}^{n+1}\ \mathtt{r\ in}\ ((\mathtt{r} \lhd \mathsf{DJ}^+); \mathsf{meas}^{\mathtt{n}}\mathtt{r})$. *The program solves any instance of Deutsch-Jozsa fixed by the value of its dimension parameter n and by providing an encoding of the function f to evaluate.*

Let M_{B_f} *be a black-box closed circuit implementing the function f that we want to check and let* $\mathsf{DJ}^\star$ *be* $\mathsf{DJ}^+[\mathsf{M}_{B_f}/\mathtt{x}]$ *namely the circuit obtained by the substitution of* M_{B_f} *to* $\mathtt{x}$ *in* DJ^+. *By means of the evaluation rule* (EqA_0), *we have* $\{(\mathtt{r}, |\underbrace{0\ldots0}_{n}\rangle)\}, \mathtt{r} \lhd \mathsf{DJ}^\star \Downarrow_1 \{(\mathtt{r}, |\phi\rangle)\}, \mathtt{skip}$ *where* $|\phi\rangle$ *is the computational state after the evaluation of* $\mathsf{DJ}^\star$. *To (partially) measure the state* $|\phi\rangle$ *we use the rule* (eqM) *to conclude* $\{\mathtt{r}, |\phi\rangle\}, \mathsf{meas}^{\underline{\mathtt{n}}}\mathtt{r} \Downarrow_1 \{\mathtt{r}, |\phi'\rangle\}, \underline{\mathtt{k}}$, *where* $(k, |\phi'\rangle, 1) \in pMeas^n(|\phi\rangle, n)$, *i.e.* $\underline{\mathtt{k}}$ *is the (deterministic) output of the measurement and* 1 *is the associated probability.* □

Example 6 (Simon's algorithm)**.** *In [32], Simon exhibited a quantum algorithm that solves in a polynomial time a problem for which the best known classical algorithm takes exponential time [1]. Simons quantum algorithm is an important precursor to Shors Algorithm for integer factorization (both algorithms are both examples of the Hidden Subgroup Problem over Abelian groups).*

We here focus on the the inherently quantum relevant fragment of Simon's algorithm [9].

Simon's problem can be formulated as follows. Let be $f : \{0,1\}^n \to X$ *(X finite) a black-box function. Determine the string* $\mathtt{s} = s_1 s_2 \ldots s_n$ *such that* $f(x) = f(y)$ *if and only if* $x = y$ *or* $x = y \oplus \mathtt{s}$. *Simon's algorithm requires an intermediate, partial measure of the quantum state. The measurement is embedded in a quantum subroutine that can be eventually iterated at most n times, where n is the input size. See [9] for further details and a careful complexity analysis.*

The rightmost circuit of Table 3 implements the quantum subroutine of Simon's algorithm and has an encoding in IQu, due to the support of both partial measurement and persistent store of quantum measurements.

Simon's subroutine sequentially composes M_1, $\mathtt{x}$ *and* M_3, *where* $\mathtt{x} : \mathsf{circ}$ *is expected to be substituted by the black-box circuit that implements the function* f *(denoted as* B_f *in Table3).* M_1 *and* M_3 *are defined by letting* $\mathsf{M}_1 = \mathsf{M}_3 = (\mathsf{M}_{par}(\mathtt{H}^{\underline{1}})\, \mathit{rsize}(\mathtt{r})) \parallel (\mathsf{M}_{par}(\mathtt{Id}^{\underline{1}})\, \mathit{rsize}(\mathtt{r})) : \mathsf{circ}$ *where: (i)* M_{par} *is the term defined in Example 5, (ii)* $\mathtt{r}$ *is a quantum register; and, (iii)* $\mathtt{H}^{\underline{1}} : \mathsf{cmd}$, $\mathtt{Id}^{\underline{1}} : \mathsf{cmd}$ *are the unary Hadamard and Identity gates, respectively.*

Let $\mathsf{M}_{\mathsf{SP}}{}^{+}$ *be the circuit* $\mathsf{M}_1 :: x :: \mathsf{M}_3 : \mathsf{circ}$ *. Let n be the arity of f we want to check. The program that implements Simon's subroutine can be* $\lambda \mathtt{x}^{\mathsf{circ}}.\ \mathit{qnew}^{2*n}\ \mathtt{r}\ \mathtt{in}\ ((\mathtt{r} \lhd \mathsf{M}_{\mathsf{SP}}{}^{+}); \mathit{meas}^{\underline{n}}\, \mathtt{r})$, *where the abstracted variable* $x : \mathsf{circ}$ *will be replaced by a suitable encoding of the black-box function that implements* f.

Let $\mathsf{M}_{B_f} : \mathsf{circ}$ *be the encoding of the circuit implementing* f *and let* $\mathsf{M}_{\mathsf{SP}}{}^{\star}$ *be* $\mathsf{M}_{\mathsf{SP}}{}^{+}[\mathsf{M}_{B_f}/\mathtt{x}]$, *namely the circuit obtained by the substitution of* M_{B_f} *for* $\mathtt{x}$ *in* $\mathsf{M}_{\mathsf{SP}}{}^{+}$.

It is easy to check that the following evaluation respects the IQu *semantics (rule* (EqA_0)*):*

$$\{(\mathtt{r}, |\underbrace{0 \ldots 0}_{2*n}\rangle)\}, \mathtt{r} \lhd \mathsf{M}_{\mathsf{SP}}{}^{\star} \Downarrow_1 \{(\mathtt{r}, |\phi\rangle)\}, \mathit{skip} \quad ,$$

where $|\phi\rangle$ *is the state after the evaluation of the circuit* $\mathsf{M}_{\mathsf{SP}}{}^{\star}$. *We can measure the first* $\underline{\mathtt{n}}$ *quantum bits as follows:* $\{(\mathtt{r}, |\phi\rangle)\}, \mathit{meas}^{\underline{\mathtt{n}}}\, \mathtt{r} \Downarrow_{\alpha} \{(\mathtt{r}, |\phi'\rangle)\}, \underline{\mathtt{k}}$, *where* $(k, |\phi'\rangle, \alpha) \in \mathit{pMeas}^{2*n}(\overline{\mathbb{s}}'(\mathtt{r}), n)$.

The classical output $\underline{\mathtt{k}}$ *can be used as a feedback from the quantum co-processor by the classical program, in this way it can decide how to proceed in the computation. In particular, it can use the measurement as guard-condition in a loop that iterates the subroutine. So we can easily re-use the Simon-circuits above as many times as we want, by arbitrarily reducing the probability error.* □

5 Conclusions and future work

IQu is a higher-order programming language that manages quantum co-processors. We formalize co-processors as quantum registers that store quantum states. This approach is radically new w.r.t. the existing proposals due to the following distinctive features: (i) each quantum variable is associated to a unique quantum state, we can duplicate such a name at will without invalidate the linear constraints that the quantum state has to satisfy; (ii) we formalize an elegant hiding mechanism that provides a natural approach to multiple co-processors internalized in the language; and, (iii) the classical programming constructs included in IQu can be used naturally by a traditional programmer, because they are unaffected by the generally quite restrictive requirements about the management of quantum data. This approach introduces a neat separation between the description of the directives to manipulate states in quantum co-processors from the names for quantum states. The reason is that directives are circuits that we consider as classical data that are freely duplicable and erasable. Since the wide expressiveness of Idealized Algol is preserved in IQu, we think we are proposing a programming tool which represent a further step in the design of quantum programming languages, coherently with directions that Knill advocates in [10].

Current ongoing work focuses on semantics and typing systems of IQu. First, we plan to add dependent types for circuits and registers, in analogy to [22, 23]. Second, we are studying a mature approach to the representation of quantum circuits by making explicit the linear management of their wires (namely, a revised and restricted version of the circuits considered in [23]). Third, we are interested in the formalization of a call-by-value version of IQu. The goal is to further ease the embedding of quantum programming in traditional programming frameworks. Fourth, we are interested in developing a denotational semantics for IQu, maybe a not complete one, but suitable to tackle the equivalence between programs involving (meaningful) quantum, non-deterministic [2, 3], probabilistic and reversible [20, 21] aspects [4].

References

[1] Sanjeev Arora & Boaz Barak (2009): *Computational Complexity: A Modern Approach*, 1st edition. Cambridge University Press, New York, NY, USA, doi:10.1017/CBO9780511804090.

[2] Federico Aschieri & Margherita Zorzi (2013): *Non-determinism, Non-termination and the Strong Normalization of System T*. In: *Typed Lambda Calculi and Applications, 11th International Conference, TLCA 2013, Eindhoven, The Netherlands, June 26-28, 2013. Lecture Notes in Computer Science, 7941*, pp. 31–47, doi:10.1007/978-3-642-38946-7_5.

[3] Federico Aschieri & Margherita Zorzi (2016): *On natural deduction in classical first-order logic: Curry-Howard correspondence, strong normalization and Herbrand's theorem*. *Theoretical Computer Science* 625, pp. 125–146, doi:10.1016/j.tcs.2016.02.028.

[4] Ugo Dal Lago & Margherita Zorzi (2012): *Probabilistic operational semantics for the lambda calculus*. *RAIRO - Theoretical Informatics and Applications* 46(3), pp. 413–450, doi:10.1051/ita/2012012.

[5] Vincent Danos & Russell S. Harmer (2002): *Probabilistic Game Semantics*. *ACM Trans. Comput. Logic* 3(3), pp. 359–382, doi:10.1145/507382.507385.

[6] Alejandro Díaz-Caro, Pablo Arrighi, Manuel Gadella & Jonathan Grattage (2011): *Measurements and Confluence in Quantum Lambda Calculi With Explicit Qubits*. *Electr. Notes Theor. Comput. Sci.* 270(1), pp. 59–74, doi:10.1016/j.entcs.2011.01.006.

[7] Jonathan Grattage (2011): *An Overview of QML With a Concrete Implementation in Haskell*. *Electr. Notes Theor. Comput. Sci.* 270(1), pp. 165–174, doi:10.1016/j.entcs.2011.01.015.

[8] Alexander S. Green, Peter LeFanu Lumsdaine, Neil J. Ross, Peter Selinger & Benoît Valiron (2013): *Quipper: A Scalable Quantum Programming Language*. In: *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, ACM, New York, NY, USA, pp. 333–342, doi:10.1145/2491956.2462177.

[9] Phillip Kaye, Raymond Laflamme & Michele Mosca (2007): *An introduction to quantum computing*. Oxford University Press, Oxford.

[10] E. Knill (1996): *Conventions for quantum pseudocode*. Technical Report, Los Alamos National Laboratory. Technical Report.

[11] Ugo Dal Lago & Margherita Zorzi (2014): *Wave-Style Token Machines and Quantum Lambda Calculi*. In: *Proceedings Third International Workshop on Linearity, LINEARITY 2014, Vienna, Austria, 13th July, 2014, Electronic Proceedings in Theoretical Computer Science 176*, pp. 64–78, doi:10.4204/EPTCS.176.6.

[12] Mohamed Yousri Mahmoud & Amy P. Felty (2018): *Formal Meta-level Analysis Framework for Quantum Programming Languages*. *Electr. Notes Theor. Comput. Sci.* 338, pp. 185–201, doi:10.1016/j.entcs.2018.10.012.

[13] Andrea Masini, Luca Viganò & Margherita Zorzi (2011): *Modal Deduction Systems for Quantum State Transformations*. *Multiple-Valued Logic and Soft Computing* 17(5-6), pp. 475–519. Available at `https://www.oldcitypublishing.com/journals/mvlsc-home/mvlsc-issue-contents/mvlsc-volume-17-number-5-6-2011/mvlsc-17-5-6-p-475-519/`.

[14] Michael A. Nielsen & Isaac L. Chuang (2010): *Quantum computation and quantum information, 10h Anniversary Edition*. Cambridge University Press, Cambridge, doi:10.1017/CBO9780511976667.

[15] Peter W. O'Hearn & Robert D. Tennent, editors (1997): *Algol-like Languages.* Progress in Theoretical Computer Science, Birkhauser. Two volumes.

[16] C.-H.L. Ong (2004): *An approach to deciding the observational equivalence of Algol-like languages*. *Annals of Pure and Applied Logic* 130(1), pp. 125 – 171, doi:10.1016/j.apal.2003.12.006.

[17] Michele Pagani, Peter Selinger & Benoît Valiron (2014): *Applying quantitative semantics to higher-order quantum computing*. In: *Proceedings of POPL '14*, ACM, pp. 647–658, doi:10.1145/2535838.2535879.

[18] L. Paolini, M. Piccolo & M. Zorzi (2019): *QPCF: higher order languages and quantum circuits*. Journal of Automated Reasoning. To appear.

[19] Luca Paolini (2006): *A stable programming language*. Information and Computation 204(3), pp. 339 – 375, doi:10.1016/j.ic.2005.11.002.

[20] Luca Paolini, Mauro Piccolo & Luca Roversi (2016): *A Class of Reversible Primitive Recursive Functions*. Electronic Notes in Theoretical Computer Science 322(18605), pp. 227–242, doi:10.1016/j.entcs.2016.03.016.

[21] Luca Paolini, Mauro Piccolo & Luca Roversi (2018): *On a Class of Reversible Primitive Recursive Functions and Its Turing-Complete Extensions*. New Generation Computing 36(3), pp. 233–256, doi:10.1007/s00354-018-0039-1.

[22] Luca Paolini & Margherita Zorzi (2017): *qPCF: a language for quantum circuit computations*. In: Theory and Applications of Models of Computation, Lecture Notes in Computer Science 10185, Springer, pp. 455–469, doi:10.1007/978-3-319-55911-7_33.

[23] Jennifer Paykin, Robert Rand & Steve Zdancewic (2017): *QWIRE: A Core Language for Quantum Circuits*. In: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, ACM, New York, NY, USA, pp. 846–858, doi:10.1145/3009837.3009894.

[24] Benjamin C. Pierce (2002): *Types and Programming Languages*. The MIT Press.

[25] Andrew M. Pitts (1997): *Reasoning About Local Variables with Operationally-Based Logical Relations*. In: Algol-like Languages, chapter 17, Progress in Theoretical Computer Science, Birkhäuser, pp. 165–185, doi:10.1007/978-1-4757-3851-3_7.

[26] Robert Rand, Jennifer Paykin & Steve Zdancewic (2017): *QWIRE Practice: Formal Verification of Quantum Circuits in Coq*. In: Proceedings 14th International Conference on Quantum Physics and Logic, Nijmegen, The Netherlands, 3-7 July 2017, *EPTCS* 266, pp. 119–132, doi:10.4204/EPTCS.266.8.

[27] Francisco Rios & Peter Selinger (2018): *A Categorical Model for a Quantum Circuit Description Language*. In: Proceedings 14th International Conference on Quantum Physics and Logic, Nijmegen, The Netherlands, 3-7 July 2017, *EPTCS* 266, Open Publishing Association, pp. 164–178, doi:10.4204/EPTCS.266.11.

[28] Neil J. Ross (2015): *Algebraic and Logical Methods in Quantum Computation*. Ph.D. thesis, Department of Mathematics and Statistics, Dalhousie University. Available from arXiv:1510.02198.

[29] Peter Selinger (2004): *Towards a Quantum Programming Language*. Mathematical Structures in Computer Science 14(4), pp. 527–586, doi:10.1017/S0960129504004256.

[30] Peter Selinger & Benoit Valiron (2006): *A lambda calculus for quantum computation with classical control*. Mathematical Structures in Computer Science 16, pp. 527–552, doi:10.1017/S0960129506005238.

[31] Peter Selinger & Benoît Valiron (2009): *Semantic Techniques in Quantum Computation*, chapter Quantum lambda calculus, pp. pp. 135–172. Cambridge University Press, doi:10.1017/CBO9781139193313.

[32] Daniel R. Simon (1994): *On the Power of Quantum Computation*. SIAM Journal on Computing 26, pp. 116–123, doi:10.1137/S0097539796298637.

[33] Benoît Valiron, Neil J. Ross, Peter Selinger, D. Scott Alexander & Jonathan M. Smith (2015): *Programming the Quantum Future*. Commun. ACM 58(8), pp. 52–61, doi:10.1145/2699415.

[34] Luca Viganò, Marco Volpe & Margherita Zorzi (2014): *Quantum state transformations and branching distributed temporal logic*. In: Logic, Language, Information, and Computation - 21st International Workshop, WoLLIC 2014, Valparaíso, Chile, September 1-4, 2014, Lecture Notes in Computer Science 8652, Springer, pp. 1–19, doi:10.1007/978-3-662-44145-9_1.

[35] Luca Viganò, Marco Volpe & Margherita Zorzi (2017): *A branching distributed temporal logic for reasoning about entanglement-free quantum state transformations*. Information & Computation 255, pp. 311–333, doi:10.1016/j.ic.2017.01.007.

[36] Margherita Zorzi (2016): *On quantum lambda calculi: a foundational perspective*. Mathematical Structures in Computer Science 26(7), pp. 1107–1195, doi:10.1017/S0960129514000425.