qPCF: A Language for Quantum Circuit Computations

Luca Paolini[1(✉)] and Margherita Zorzi[2(✉)]

[1] Department of Computer Science, University of Torino, Turin, Italy
paolini@di.unito.it

[2] Department of Computer Science, University of Verona, Verona, Italy
margherita.zorzi@univr.it

Abstract. We propose qPCF, a functional language able to define and manipulate quantum circuits in an easy and intuitive way. qPCF follows the tradition of "quantum data & classical control" languages, inspired to the QRAM model. Ideally, qPCF computes finite circuit descriptions which are offloaded to a quantum co-processor (i.e. a quantum device) for the execution. qPCF extends PCF with a new kind of datatype: quantum circuits. The typing of qPCF is quite different from the mainstream of "quantum data & classical control" languages that involves linear/exponential modalities. qPCF uses a simple form of dependent types to manage circuits and an implicit form of monad to manage quantum states via a destructive-measurement operator.

1 Introduction

In the last fifteen years, the definition and the development of quantum programming languages catalyzed the attention of a part of the computer science research community. Quantum computers are a long term but concrete reality. Even if physicists and engineers have to continuously face tricky problems in the realization of quantum devices, the advance of these innovative technologies promises a noticeable speedup.

A calculus for quantum computable functions should present two different facets. On the first hand there is the unitary aspect of the calculus, that captures the essence of quantum computing as algebraic transformations of state vectors by means of unitary operators. On the other hand, it should be possible to *control* the quantum steps by means of classical computational steps, "embedding" the pure quantum evolution in a classical computation. Behind this second perspective we have the usual idea of computation as a sequence of discrete steps on (the mathematical description of) an abstract machine. The relationship between these different aspects gives rise to different approaches to quantum functional calculi (as observed in [1]). If we divide the two features, i.e. we separate data from control, we adopt the so called *quantum data & classical control* (*qd&cc*) approach. This means that classical computation is *hierarchical dependent* from the quantum part: a classical program (ideally in execution on a classical machine) computes some "directives": these directives are sent to a

T.V. Gopal et al. (Eds.): TAMC 2017, LNCS 10185, pp. 455–469, 2017.
DOI: 10.1007/978-3-319-55911-7_33

hypothetical device which apply them to quantum data. Therefore classical program controls quantum data or, in other words, classical computational steps control the unitary part of the calculus. In general, the classical control acts on the quantum side of the computation in two way: by means of the selection of unitary transformations to be applied and by means of data observations, i.e. by means of measurements. A different approach based on the quantum control is the *superposition-of-programs paradigm*. See [34], Part III, for details. This idea is inspired to an architectural model called Quantum Random Access Machine (QRAM). The QRAM has been defined in [10] and can be viewed as a classically controlled machine enriched with a quantum device. On the grounds of the QRAM model, Selinger defined the first functional language based on the quantum data-classical control paradigm [28]. This work represents a milestone in the development of quantum functional calculi and inspired a number of different investigations. A key research line tried to retrace, in the quantum setting, foundational results about computability. In this direction, calculi for quantum computable functions have been defined, and equivalence results with other computational models, such as Quantum Turing Machine and Quantum Circuit Families, have been proved [4,11,12,35]. Moreover, interesting proposals to provide satisfactory denotational semantics for *qd&cc* functional calculi have been proposed in [8,19,29]. Many quantum programming languages [7,28,29] implementing the (*qd&cc*) approach have been proposed in literature. A recent and interesting proposal is Quipper which is an embedded, scalable functional programming language for quantum computing proposed in [28]. Quipper is essentially a high-level circuit description language: circuits can be created, manipulated, evaluated in a mixture of procedural and declarative programming styles. The most important quantum algorithms can be easily encoded thanks to a number of programming tools, macros, and extensive libraries of quantum functions. The idea of the separation between control and data is definitely reformulated in terms of *quantum-coprocessor* [31]. Quipper has been mainly developed as a *concrete* language. Authors are not interested in the foundational study of it. Quipper is based on the lambda calculus with classical control proposed in [28], and this relationship is discussed in [26], by means of a suitable calculus named Proto-quipper. In [14], the semantics of Proto-Quipper is further formalized by means of the linear specification logic SL. The type system is based on a linear logic approach that ensures the correct interaction of classical and quantum types. The "*qd&cc*" philosophy, in particular the circuit generation oriented approach, has been also adopted in the purely linear core-language QWire, introduced in [22]: a low-level quantum language developed to be a "quantum plugin" for a hosting classical language like Haskell. QWire and qPCF are based on some similar ideas. Differently from qPCF, QWire retains the focus on quantum states which is typical of the *qd&cc* tradition. In qPCF quantum states are not more atomic data, they are replaced by quantum circuits. In this paper we advance in the research on the languages for *qd&cc* paradigm by formalizing a flexible quantum language. We propose qPCF, a simple extension of PCF. We quickly list the main features of qPCF.

– *Absence of explicit linear constraints*: the management of linear resources is radically different from the ones proposed in languages inspired to Linear Logic such as [4,11,12,14,26,28,35]; so, we do not use linear/exponential modalities.
– *Use of dependent types*: we decouple the classic control from the quantum computation by adopting a simplified form of dependent types that provide a sort of linear interface.
– *Emphasis of the Standardization Theorem*: the Standardization Theorem, proved in [4,35], and largely used in circuit description languages such as Quipper, decomposes computations in different phases, according to the quantum circuit construction by classical instructions and the successive, independent, evaluation phase involving quantum operations.
– *Unique measurement at the end of the computation*: following the "principle of deferred measurement" which states that any quantum circuit is equivalent to one where all measurements are performed as the very last operations (see, e.g., [17]), we add an explicit measurement operator to qPCF syntax that models the *(von Neumann) Total Measurement* [17], a kind of measurement that reduces a quantum state to a classical one (a sequence of classical bit). Essentially we are using a monad-style programming, and we "embed" both, quantum evaluation and measurement into the operator dmeas (see Sect. 3 for dmeas operational behavior).
– *Implicit representation of quantum states*: differently from other proposals (e.g. [4,28,35]), we hide quantum states we are working on. This can be achieved thanks to the monadic-style approach we mentioned above.
– *Turing Completeness*: qPCF retains PCF expressive power. So, a qPCF term can represent an infinite class of circuits.

Synopsis: Section 2 introduces syntax and typing system of qPCF; the operational semantics of qPCF is in Sect. 3; Sect. 4 sketches some properties of qPCF; Sect. 5 contains some examples of qPCF circuit encodings; Sect. 6 is devoted to discuss conclusions and future work.

2 qPCF

In this section we describe qPCF, a programming language that pursue seriously the application of the standardization theorem of [4,35]: it states that, in the "quantum-data & classic control languages", the quantum evaluation can always be postponed after the classical execution. On the other hand, the classical evaluation designs a quantum circuit that can be evaluated in a second time. Ideally, qPCF computes a finite circuit description which is offloaded to a quantum co-processor for the execution. qPCF is definitively more flexible than the languages presented in [4,19,27–29]. It extends PCF with quantum circuits, viz. a new kind of classical data. Indeed, as observed in [22], quantum circuits can be freely duplicated and erased. We realized that the linearity of mainstream typing systems of "quantum-data & classic control" languages has been used to impose constraints on both the management of quantum-data and the management of classic control. qPCF neatly splits these linear facets by using two different solutions. On the one hand, qPCF shows that linearity for quantum control can be completely confined to atomic datatypes by using a simplified form of dependent types [23]. A dependent type picks up a family of types that bring in the type auxiliary information (just circuit arity in our case). On the other hand, the linearity needed to manage quantum state is hidden in a destructive measure operator (by means of an implicit form of monad) that model von Neumann Measurement [17] and allows us to avoid the explicit management of intermediate quantum states.

In the rest of the paper, we assume some familiarity with notions as quantum bits (the quantum equivalent of classical data), quantum states [15,16,32,33] (systems of n quantum bits), quantum circuit and quantum circuit families [18]. A quantum circuit generalizes the idea of classical circuit, replacing the elementary classical gates (AND, OR, NOT...) by elementary quantum gates [17], that enjoy matemathical descriptions in terms of *unitary operators* on suitable Hilbert spaces. A quantum circuit family can be (quite informally) considered as a function $\mathbf{K} : \mathbb{N} \rightarrow \mathcal{K}$ (denotable as $(K_i)_{i<\omega}$), where $\mathcal{K}$ is the set of circuit descriptions; $\mathbf{K}(n)$ returns K_n, i.e. the circuit of ariety n. See [35] for a friendly introduction to quantum computing. We remand to [17] for a complete overview about the topic. See also [9,18] for details about quantum circuit families and some crucial discussions about the universality of sets of quantum gates. Finally, see [25] for a rigorous algebraic characterisation of quantum computing.

2.1 Syntax

Dependent types have been widely used in strongly normalizing settings (usually, with logic goals) where the evaluation of expressions is always terminating. But in programming settings the strong normalization [21] is not realistic. Unfortunately, to allow types that embeds undefined terms (viz. not strongly normalizing ones) requires the management of "undefined types" [3]. We circumvent this issue by identifying a subclass of terms (always normalizing) that we use in our dependent types. qPCF extends PCF [5,6,24] to manage some additional atomic data structures: *indexes* (always normalizing number expressions) and *circuits*.

The row syntax of qPCF follows.

$$\begin{aligned}\mathtt{M},\mathtt{N} ::= {} & \mathtt{x} \mid \lambda\mathtt{x}.\mathtt{M} \mid \mathtt{MN} \mid \underline{\mathtt{n}} \mid \mathtt{pred} \mid \mathtt{succ} \mid \mathtt{if} \mid \mathtt{Y}_\sigma \mid \mathtt{set} \mid \mathtt{get} \\ & \mid \odot\mathtt{EE}' \mid \underline{\mathtt{s}} \mid \mathtt{append} \mid \mathtt{iter} \mid \mathtt{reverse} \mid \mathtt{size} \mid \mathtt{dMeas}\ .\end{aligned}$$

In the first row we extended PCF with syntactic sugar to facilitate the bitwise access to digit: `get` allows us to read the i-th digit of the binary representation of a numeral, i.e. its i-th bit; `set` allows us to modify the i-th bit of a numeral.

Index expressions (ranged over by E) are completely formalized via the typing (cf. Table 1). They include numerals and some *total* operations on expressions: $\odot \in \{+, *\}$ (viz. sum, product).

We assume U to range on a given set of selected gates (i.e. unitary operator, see [17]): if $\mathcal{U}$ is a fixed set of computable unitary operators then, we associate to each computable operator $\mathbf{U} \in \mathcal{U}$ a symbol U. We represent circuits by means of *strings*, viz. the Kleene-closure of the following symbols: the parallel composition of circuits denoted $\|$ (i.e. side-by-side placing of circuits), the sequential composition of circuits $\fatsemi$ and gate-names `U`. `append` sequentializes two circuits of the same arity. `iter` produces the parallel composition of a first circuit with a given numbers of a second one. The goal of the operator `reverse` is to transform a circuit in its adjoint one (in case the gate-base has been chosen closed under adjunction, otherwise it will be meaningless).

`size` is an operator that applied to a circuit returns its arity: an index information. It is worth to notice that `size` do not add any expressivity to qPCF, because the programmer can explicitly manage pairs of "circuit together with arity": so that, a projection provides the arity of the circuit. We added `size` to qPCF to emphasize the gain that dependent type can provide in a concrete context, although this makes the proofs of the language properties more complex.

Last, but not least, we use `dMeas` (a.k.a. destructive measure) to evaluate circuits initialized via a numeral (representing a binary classical state): `dMeas` returns the classic state (encoded in the binary representation of a numeral) obtained measuring the final quantum state of the considered circuit. Traditionally *qd&cc* languages focus on quantum states, while qPCF focuses on circuits (hiding states in an monadic measure).

Typically, $\mathcal{U}$ will include a universal base for quantum circuits (see [9]). We like to remark that we can instance $\mathcal{U}$ to interesting family of gate as reversible ones: in these cases we are not properly building a quantum programming language. If $k \in \mathbb{N}$ then we denote $\mathcal{U}(k)$ the gates in $\mathcal{U}$ having arity $k+1$, so $\mathcal{U} = \bigcup_0^\omega \mathcal{U}(k)$. Notice that we do not introduced explicit permutations, because they can be provided by means of a convenient choice of quantum gates (see, e.g. [30]). Thus, the choice $\mathcal{U}$ determines whose permutations our circuits can use. We also notice that the gate identity is a particular permutation.

2.2 Typing System

Standard PCF types are extended to manage circuits and their dependencies. We use types decorated by numerals to define a denumerable family of circuits. Our approach is closely inspired to that mentioned in [23, Sect. 30.5] to manage types of vectors (with dependencies): the decoration carry around some arity information. We avoided general dependent types systems (see [3] for a survey) because their great expressiveness is exceeding our need, we preferred to maintain the qPCFtype system as simple as possible by aiming to show the feasibility of the approach and its concrete benefits. Our approach to dependent types is based on a special kind of numeric expressions that can be managed in a limited way: index. Summing up, types of PCF (i.e. integers and arrows) are flanked by two new types: circuits (viz. strings typed with dependent types that carry around numeric information about arities) and indexes (that grasp a subset of numeric expressions that express only terminating expressions).

Types of qPCF are formalized by the following grammar:

$$\sigma, \tau ::= \mathsf{Nat} \mid \mathsf{Idx} \mid \sigma \to \tau \mid \mathsf{circ}(\mathtt{E}) \mid \Pi \mathtt{x}.\tau$$

where $\mathtt{E}$ is an index expressions (morally, a strongly normalizing numeric expression). As in standard dependent type system, we replace arrows involving dependencies by quantified types, namely an arrow $\sigma \to \tau$ is replaced by $\Pi \mathtt{x}^{\sigma}.\tau$ whenever $\mathtt{x}^{\sigma}$ occurs in τ in order to emphasize that τ *depends* from $\mathtt{x}$.

Index. Idx aims to pick up a subset of expressions on natural numbers being strongly normalizing, i.e. we want to cut out undefined PCF-expressions as $\mathtt{Y}_{\mathsf{Nat}}(\lambda \mathtt{x}^{\mathsf{Nat}}.\mathtt{x})$ (viz. a looping forever term). The leading use of Idx is to type terms $\mathtt{M}$ embodied in dependent types (i.e. used in types via $\mathsf{circ}(\mathtt{M})$). The goal is to select numeric expressions that made the equivalence decidable (when such expressions are closed). We focus on a restricted, but revealing, syntax of index expressions is $\mathtt{E} ::= \mathtt{x}^{\mathsf{Idx}} \mid \underline{\mathtt{n}} \mid \odot\,\mathtt{E}\,\mathtt{E}'$ where $\odot \in \{+, *\}$ viz. operators denoting addition and multiplication. We are considering a very basic set of binary operators that can be conveniently extended in a concrete case, e.g. by adding the (positive subtraction) $\dot{-}$, or the %, or a selection $\mathtt{if}^{\mathtt{x}}$ and so on.

Above expressions are typed Idx by the following rules:

$$\frac{}{B[\mathsf{x} : \mathsf{Idx}] \vdash \mathsf{x}^{\mathsf{Idx}} : \mathsf{Idx}}\ (i_1) \qquad \frac{}{B \vdash \underline{\mathtt{n}} : \mathsf{Idx}}\ (i_2) \qquad \frac{B \vdash \mathrm{E}_0 : \mathsf{Idx} \quad B \vdash \mathrm{E}_1 : \mathsf{Idx}}{B \vdash \odot\,\mathrm{E}_0\,\mathrm{E}_1 : \mathsf{Idx}}\ (i_3)$$

where B denotes a standard typing base, i.e. sets of pairs (variable and type).

Index expression are closed when they do not contain any free variable. As usual for PCF, the evaluation is focused on closed expressions, and formalized by the following rules:

$$\frac{}{\underline{\mathtt{n}} \Downarrow \underline{\mathtt{n}}}\ (n) \qquad \frac{\mathrm{E}_0 \Downarrow \underline{\mathtt{m}} \quad \mathrm{E}_1 \Downarrow \underline{\mathtt{n}}}{\odot\,\mathrm{E}_0\,\mathrm{E}_1 \Downarrow \underline{\mathtt{m} \odot \mathtt{n}}}\ (\mathrm{op})$$

where we use the $\odot$ to denote both its name and its straighforward semantics. We remark that we are considering a strict subset of the index expressions of qPCF in order to increase some intuition (e.g. by neglecting $\mathtt{size}$).

It is immediate that the above index expressions are normalizing with the proposed evaluation strategy, when we focus on closed terms. Moreover, we can informally claim that they are strongly normalizing in the straightforward lambda-calculus behind our semantics, that can be obtained as usual by including some δ-rules for constants.

The most basic property of paradigmatic typing system is that well-typed terms do not "go wrong", i.e. types are preserved by the evaluation and, if the evaluation stops then the result is a value.

Theorem 1 (Preservation & Progress). *(i) If* $\vdash E : \mathsf{Idx}$ *and* $E \Downarrow E'$ *then* $\vdash E' : \mathsf{Idx}$*. (ii) If* $\vdash E : \mathsf{Idx}$ *and* $E \Downarrow E'$ *then* E' *is a numeral.*

Remaining typing. We can now extend the typing to the whole qPCF: the typing system is given in Table 1 (be careful to implicit assumption remarked in the caption).

Table 1. Typing Rules. Each typing rule contains implicit premises: (i) all occurrences of circuits types (also those in bases) embody a term typed Idx with the given base; (ii) all free variables (occurring in terms and types) are typed in the base.

$$\frac{}{B[\mathtt{x}:\sigma]\vdash \mathtt{x}:\sigma}\,(p_1) \qquad \frac{}{B\vdash \underline{\mathtt{n}}:\mathsf{Nat}}\,(p_2) \qquad \frac{}{B\vdash \underline{\mathtt{n}}:\mathsf{Idx}}\,(i_2)$$

$$\frac{B[\mathtt{x}:\sigma]\vdash \mathtt{N}:\tau \qquad \mathtt{x}\notin \mathrm{FV}(\mathtt{N})}{B\vdash \lambda\mathtt{x}^{\sigma}.\mathtt{N}:\sigma\to\tau}\,(p_3) \qquad \frac{B\vdash \mathtt{P}:\sigma\to\tau \quad B\vdash \mathtt{Q}:\sigma}{B\vdash \mathtt{PQ}:\tau}\,(p_4)$$

$$\frac{B[\mathtt{x}:\sigma]\vdash \mathtt{N}:\tau \qquad \mathtt{x}\in \mathrm{FV}(\mathtt{N})}{B\vdash \lambda\mathtt{x}.\mathtt{N}:\Pi\mathtt{x}^{\sigma}.\tau}\,(x_1) \qquad \frac{B\vdash \mathtt{P}:\Pi\mathtt{x}^{\sigma}.\tau \quad B\vdash \mathtt{E}:\sigma}{B\vdash \mathtt{PE}:\tau[\mathtt{E}/\mathtt{x}]}\,(x_2)$$

$$\frac{}{B\vdash \mathtt{succ}:\mathsf{Nat}\to\mathsf{Nat}}\,(p_5) \qquad \frac{\sigma\in\{\mathsf{Nat},\mathsf{circ}(\mathtt{E})\}}{B\vdash \mathtt{if}:\mathsf{Nat}\to\sigma\to\sigma\to\sigma}\,(p_6)$$

$$\frac{}{B\vdash \mathtt{pred}:\mathsf{Nat}\to\mathsf{Nat}}\,(p_7) \qquad \frac{\sigma=\tau_1\to\ldots\to\tau_n\to\tau \quad (n\geq 0) \quad \tau\in\{\mathsf{Nat},\mathsf{circ}(\mathtt{E})\}}{B\vdash \mathtt{Y}_{\sigma}:(\sigma\to\sigma)\to\sigma}\,(p_8)$$

$$\frac{}{B\vdash \mathtt{get}:\mathsf{Nat}\to\mathsf{Nat}\to\mathsf{Nat}}\,(b_1) \qquad \frac{}{B\vdash \mathtt{set}:\mathsf{Nat}\to\mathsf{Nat}\to\mathsf{Nat}}\,(b_2)$$

$$\frac{B\vdash \mathtt{M}:\mathsf{Idx}}{B\vdash \mathtt{M}:\mathsf{Nat}}\,(x_3) \qquad \frac{B\vdash \mathtt{M}:\mathsf{circ}(\mathtt{E})}{B\vdash \mathtt{size}\ \mathtt{M}:\mathsf{Idx}}\,(x_4) \qquad \frac{B\vdash \mathtt{E}_0:\mathsf{Idx} \quad B\vdash \mathtt{E}_1:\mathsf{Idx}}{B\vdash \odot\,\mathtt{E}_0\,\mathtt{E}_1:\mathsf{Idx}}\,(i_3)$$

$$\frac{\mathtt{U}\in\mathcal{U}(k)}{B\vdash \mathtt{U}:\mathsf{circ}(\underline{\mathtt{k}})}\,(c_1) \qquad \frac{}{B\vdash \mathtt{iter}:\Pi\mathtt{x}^{\mathsf{Idx}}.\,\mathsf{circ}(\mathtt{E}_0)\to\mathsf{circ}(\mathtt{E}_1)\to\mathsf{circ}(\mathtt{E}_0+((1+\mathtt{E}_1)*\mathtt{x}))}\,(c_2)$$

$$\frac{}{B\vdash \fatsemi:\mathsf{circ}(\mathtt{E})\to\mathsf{circ}(\mathtt{E})\to\mathsf{circ}(\mathtt{E})}\,(c_1') \qquad \frac{}{B\vdash \|:\mathsf{circ}(\mathtt{E}_0)\to\mathsf{circ}(\mathtt{E}_1)\to\mathsf{circ}(\mathtt{E}_0+\mathtt{E}_1)}\,(c_1'')$$

$$\frac{}{B\vdash \mathtt{append}:\mathsf{circ}(\mathtt{E})\to\mathsf{circ}(\mathtt{E})\to\mathsf{circ}(\mathtt{E})}\,(c_3) \qquad \frac{}{B\vdash \mathtt{reverse}:\mathsf{circ}(\mathtt{E})\to\mathsf{circ}(\mathtt{E})}\,(c_4)$$

$$\frac{}{B\vdash \mathtt{dMeas}:\mathsf{Nat}\to\mathsf{circ}(\mathtt{E})\to\mathsf{Nat}}\,(c_5)$$

Finite sets of pairs *variable, type* are called bases whenever variable-names are disjoint: we use B to range on them. We write $B[\mathtt{x}:\sigma]$ to denote the set where the pair $\mathtt{x}:\sigma$ is added (possibly replacing an pair involving $\mathtt{x}$. As usual, dependent type systems include a typing rule making explicit some type inter-convertibility. We consider types up to a congruence $\simeq$. We define $\simeq$ as the smaller equivalence that includes: (i) the α-conversion of bound variables and β-conversion, (ii) sum and product are associative and commutative; (iii) product distributes over the sum, i.e. $(*\,\mathtt{E}\,(+\,\mathtt{E}_0\,\mathtt{E}_1)) \;\simeq\; (+\,(*\,\mathtt{E}\,\mathtt{E}_0)(*\,\mathtt{E}\,\mathtt{E}_1))$; (iv) $\underline{\mathtt{0}}$ is the neuter element for the sum; and, (v) $\underline{\mathtt{1}}$ is the neuter element for the product. We use the type equivalence often implicitly. In particular in the typing system (cf. Table 1) types (containing dependencies) are considered up to equivalence.

Rules $(p_1),(p_2),(p_3),(p_4),(p_5),(p_6),(p_7),(p_8)$ are directly inherited from PCF and do not require special care. We also note that (p_1) can be instantiated to (i_1) (which has not been included in the system). Rules (p_6), (p_8) are restricted to excludes undefined index expressions. This restriction avoid types containing terms (i.e. index expressions) being not normalizing. The cases excluded by (p_3), (p_4) are managed by rules (x_1), (x_2). Rules (x_1), (x_2) reflect the usual approach of dependent types.

Rule $(b_1),(b_2)$ type `get` and `set` that use the second numeral to select a bit in the binary representation of the first argument: `get` extract such bit, `set` modify it. The rule (x_3) allows us to transform an index in a numeral typed Nat.

The rule (x_4) is typing an operator that allows us to recover in the computation the index information carried around by the circuit type.

Rules (c_1), (c_1'), (c_1''),(c_2), (c_3), (c_4), (c_5) conclude our type-equipment. (c_1), (c_1'), (c_1'') type strings representing circuits. (c_2) allows to parallel compose circuits: a base circuit $\mathtt{M}$ and some copies of a circuit $\mathtt{N}$. (c_3) allows to sequentialize circuits of the same arity. (c_4) transforms a circuit in its adjoint one.

Example 1. An interesting example of typed term that provides evidence of the circularity arising from dependent types is: $\mathtt{x} : \Pi \mathsf{z}^{\mathsf{Idx}}.\,\mathsf{circ}(\mathsf{z}) \vdash \mathtt{x}\,\mathtt{size}\,(\mathtt{M}) : \mathsf{circ}(\mathtt{size}\,(\mathtt{M}))$ where M can be any term of qPCF typed by a circuit.

The above example shows that in types can occur undefined terms, maybe containing open variables not typed Idx. In particular, $\mathtt{size}$ can contain any term (that can be typed as a circuit of a given arity). Luckily, the evaluation of $\mathtt{size}$ does not need the normalization of its argument: it just requires the normalization of its type.

3 Operational Semantics

As standard for PCF, the evaluation focuses on closed terms of ground types, viz. Nat, Idx, $\mathtt{circ}(\mathtt{E})$. Because the inclusion of dependent types and the presence of the operator $\mathtt{size}$, we assume that an evaluated terms brings implicitly in it, its whole typing information. We denote $\mathtt{V}$ the closed values of ground types, namely numerals (typed either Nat or Idx), and strings (typed as circuits of a given arity). The operational evaluation is formalized by means of the evaluation predicate $\mathtt{M} \Downarrow \mathtt{V}$: it holds whenever it is the conclusion of a derivation built on the rules presented in Table 2 (we included also the rule for the evaluation of index expressions). Table 2 includes the standard call-by-name semantics of PCF, namely the first two lines of rules. Since they are well-known, we do not insist further on them. The rules (sz), (op) compute some index expressions. In particular, (sz) recovers the numeral decorating the type circuit of a closed term. Since the involved expressions do not contain open variables, the evaluation does not pose any problem.

Let $\lceil \underline{\mathtt{m}} \rceil^{\underline{\mathtt{n}}}$ be notation for $(\underbrace{(\underline{\mathtt{m}}\,/\,\underline{2})\ldots/\,\underline{2}}_{\underline{\mathtt{n}}})\%\underline{2}$ where / is the integer division and % is the modulo. Thus, $\lceil \underline{\mathtt{m}} \rceil^{\underline{\mathtt{0}}}$ is the rightmost bit of the binary representation of $\underline{\mathtt{m}}$. Moreover, if $\underline{\mathtt{k}}$ is the logarithm (base 10) of $\underline{\mathtt{m}}$ then $\lceil \underline{\mathtt{m}} \rceil^{\underline{\mathtt{k-1}}} = \underline{1}$ and, for all $\underline{\mathtt{h}}$ greater than $\underline{\mathtt{k}}$, $\lceil \underline{\mathtt{m}} \rceil^{\underline{\mathtt{h}}} = \underline{0}$. The rule (gt) and (st) get/set a bit of the first argument (the one selected by the second argument). Notice that $\mathtt{set}$, $\mathtt{get}$ are syntactic sugar managing classical input states. In particular, the numeral $\mathtt{set}\,\underline{0}\,\underline{\mathtt{n}+1}$ represents the state $1\underbrace{0\ldots0}_{n}$.

The rules (u), (u'), (u''), (r_0), (r_1), (r_2), (a), (d) build circuits, i.e. strings on $\mathtt{\S}$, $\|$ and the gate-names $\mathtt{U}$. The semantics of $\mathtt{append}$ is simply the sequential post-position of circuits. The semantics of $\mathtt{iter}$ is the parallel composition of circuits, driven by an argument of type Idx: thus the arity of the generated circuit is well

Table 2. Operational Semantics.

$$\frac{}{\underline{\mathtt{n}} \Downarrow \underline{\mathtt{n}}}\ (n) \qquad \frac{\mathtt{M} \Downarrow \underline{\mathtt{n}}}{\mathtt{s(M)} \Downarrow \underline{\mathtt{n}+\mathtt{1}}}\ (s) \qquad \frac{\mathtt{M} \Downarrow \underline{\mathtt{n}+\mathtt{1}}}{\mathtt{p(M)} \Downarrow \underline{\mathtt{n}}}\ (p) \qquad \frac{\mathtt{M[N/x]P}_1 \cdots \mathtt{P}_m \Downarrow \mathtt{V}}{(\lambda\mathtt{x.M})\mathtt{NP}_1 \cdots \mathtt{P}_m \Downarrow \mathtt{V}}\ (\beta)$$

$$\frac{\mathtt{M} \Downarrow \underline{\mathtt{0}} \quad \mathtt{L} \Downarrow \mathtt{V}}{\mathtt{if\ M\ L\ R} \Downarrow \mathtt{V}}\ (\mathrm{if}_l) \qquad \frac{\mathtt{M} \Downarrow \underline{\mathtt{n}+\mathtt{1}} \quad \mathtt{R} \Downarrow \mathtt{V}}{\mathtt{if\ M\ L\ R} \Downarrow \mathtt{V}}\ (\mathrm{if}_r) \qquad \frac{\mathtt{M(YM)P}_1 \cdots \mathtt{P}_i \Downarrow \mathtt{V}}{\mathtt{YMP}_1 \cdots \mathtt{P}_i \Downarrow \mathtt{V}}\ (Y)$$

$$\frac{\mathtt{M} : \mathsf{circ}(\mathtt{E}) \qquad \mathtt{E} \Downarrow \underline{\mathtt{n}}}{\mathtt{size\,M} \Downarrow \underline{\mathtt{n}}}\ (\mathrm{sz}) \qquad \frac{\mathtt{E}_0 \Downarrow \underline{\mathtt{m}} \quad \mathtt{E}_1 \Downarrow \underline{\mathtt{n}}}{\odot\, \mathtt{E}_0\, \mathtt{E}_1 \Downarrow \underline{\mathtt{m} \odot \mathtt{n}}}\ (\mathrm{op}) \qquad \frac{\mathtt{M} \Downarrow \underline{\mathtt{m}} \qquad \mathtt{N} \Downarrow \underline{\mathtt{n}}}{\mathtt{get\,M\,N} \Downarrow \lceil \underline{\mathtt{m}} \rceil^{\underline{\mathtt{n}}}}\ (\mathrm{gt})$$

$$\frac{\mathtt{M} \Downarrow \underline{\mathtt{m}} \quad \mathtt{N} \Downarrow \underline{\mathtt{n}} \quad \text{and } \underline{\mathtt{m}}' \text{ is such that } \lceil \underline{\mathtt{m}}' \rceil^{\underline{\mathtt{n}}} = 1 \text{ and } \forall \underline{\mathtt{k}} \neq \underline{\mathtt{n}} \lceil \underline{\mathtt{m}}' \rceil^{\underline{\mathtt{k}}} = \lceil \underline{\mathtt{m}} \rceil^{\underline{\mathtt{k}}}}{\mathtt{set\,M\,N} \Downarrow \underline{\mathtt{m}}'}\ (\mathrm{st})$$

$$\frac{}{\mathtt{U} \Downarrow \mathtt{U}}\ (\mathrm{u}) \qquad \frac{\mathtt{M}_0 \Downarrow \mathtt{C}_0 \quad \mathtt{M}_1 \Downarrow \mathtt{C}_1}{\mathtt{M}_0 \fatsemi \mathtt{M}_1 \Downarrow \mathtt{C}_0 \fatsemi \mathtt{C}_1}\ (\mathrm{u}') \qquad \frac{\mathtt{M}_0 \Downarrow \mathtt{C}_0 \qquad \mathtt{M}_1 \Downarrow \mathtt{C}_1}{\mathtt{M}_0 \parallel \mathtt{M}_1 \Downarrow \mathtt{C}_1 \parallel \mathtt{C}_0}\ (\mathrm{u}'')$$

$$\frac{\mathtt{M}_0 \Downarrow \mathtt{C}_0 \quad \mathtt{M}_1 \Downarrow \mathtt{C}_1}{\mathtt{append\,M}_0\mathtt{M}_1 \Downarrow \mathtt{C}_0 \fatsemi \mathtt{C}_1}\ (\mathrm{a}) \qquad \frac{\mathtt{E} \Downarrow \underline{\mathtt{n}} \qquad \mathtt{M}_0 \Downarrow \mathtt{C}_0 \qquad \mathtt{M}_1 \Downarrow \mathtt{C}_1}{\mathtt{iter\ E\ M}_0\,\mathtt{M}_1 \Downarrow \underbrace{\mathtt{C}_1 \parallel \ldots \parallel \mathtt{C}_1}_{\underline{\mathtt{n}}} \parallel \mathtt{C}_0}\ (\mathrm{it})$$

$$\frac{\mathtt{M} \Downarrow \mathtt{U} \quad (\ddagger\mathtt{U}) = \mathtt{U}'}{\mathtt{reverse\,M} \Downarrow \mathtt{U}'}\ (\mathrm{r}_0) \qquad \frac{\mathtt{M} \Downarrow \mathtt{C}_0 \fatsemi \mathtt{C}_1 \quad \mathtt{reverse\,C}_0 \Downarrow \mathtt{C}_0' \quad \mathtt{reverse\,C}_1 \Downarrow \mathtt{C}_1'}{\mathtt{reverse\,M} \Downarrow \mathtt{C}_1' \fatsemi \mathtt{C}_0'}\ (\mathrm{r}_1)$$

$$\frac{\mathtt{M} \Downarrow \mathtt{C}_0 \parallel \mathtt{C}_1 \quad \mathtt{reverse\,C}_0 \Downarrow \mathtt{C}_0' \quad \mathtt{reverse\,C}_1 \Downarrow \mathtt{C}_1'}{\mathtt{reverse\,M} \Downarrow \mathtt{C}_0' \parallel \mathtt{C}_1'}\ (\mathrm{r}_2)$$

$$\frac{\mathtt{M} \Downarrow \underline{\mathtt{m}} \quad \mathtt{N} \Downarrow \mathtt{C} \quad \mathtt{N} : \mathsf{circ}(\underline{\mathtt{k}}) \quad \mathrm{circuitEval}(\underline{\mathtt{m}}\restriction_{\underline{\mathtt{k}+\mathtt{1}}}, \mathtt{C}) = \underline{\mathtt{n}}}{\mathtt{dMeas(M,N)} \Downarrow \underline{\mathtt{n}}}\ (\mathrm{m})$$

defined. The semantics of `reverse` is build to produce the adjoint circuits when a suitable endo-function $\ddagger$ is provided by the co-processor. If the co-processor do not provide it (for instance, because the set $\mathcal{U}$ of unitary gate is not closed under adjunction) then we let $\ddagger$ be the identity, so that `reverse` is well-defined but uninteresting.

Let $\underline{\mathtt{m}}\restriction_{\underline{\mathtt{k}}}$ to denote the numeral $\underline{\mathtt{k}}$ such that the binary representation of $\underline{\mathtt{n}}$ is the restriction of the binary representation of $\underline{\mathtt{m}}$ on the first $\underline{\mathtt{k}}$ bits. It is worth to recall that, conventionally, each classic state is represented via an integer having (implicitly) a number of relevant bits as the arity of the circuit. The rule (m) evaluates the `dMeas` arguments and uses the results of these evaluations to feed an external evaluation: morally, a quantum co-processor [31]. The co-processor call is done by using the auxiliary evaluation circuitEval. It executes the quantum circuits on the provided classical initialization, then it returns the measure of the whole final state. The rule explicitly restricts the evaluation of the first argument to the relevant number of bits (i.e. the arity of the circuit).

In order to define our co-processor we need two ingredients. The first one is the semantic for the evaluation of the circuit. We denote *Circ* the valid strings of circuits, and $\mathcal{O}$ the set of unitary operators on finite dimensional Hilbert spaces (informally, we are mapping circuit descriptions into their mathematical denotation, i.e. into corresponding algebraic operators). So that we can define the circuit semantic by using the function Hilb : $Circ \to \mathcal{O}$ defined as follows: $\mathrm{Hilb}(\mathtt{U}) ::= \mathbf{U}$; $\mathrm{Hilb}(\mathtt{C}_0 \parallel \mathtt{C}_1) ::= \mathrm{Hilb}(\mathtt{C}_0) \otimes \mathrm{Hilb}(\mathtt{C}_1)$; $\mathrm{Hilb}(\mathtt{C}_0 \,\mathring{9}\, \mathtt{C}_1) ::= \mathrm{Hilb}(\mathtt{C}_0) \circ \mathrm{Hilb}(\mathtt{C}_1)$. The second one is the semantic of the measure that rest on the von Neumann Measurement [9], here dubbed vMeas. We define circuitEval($\underline{\mathtt{n}}$, $\mathtt{C}$) to be the measure of the application of the circuit to our initial state (in the assumed base), namely $\mathrm{circuitEval}(\underline{\mathtt{n}}, \mathtt{C}) ::= \mathrm{vMeas}(\mathrm{Hilb}(\mathtt{C}), \underline{\mathtt{n}})$.

Equivalence. The operational equivalence can be defined by just considering closed terms of type Nat because the operational differences in the other types can be traced back to Nat (the reverse can be easily proved be false).

For many reasons, we are remarking the relevance of the notion of program of PCF, i.e. a closed term of type Nat. First, the result of a circuit-measure is a list of bits, viz. a natural number. Second, the circuits are represented by using strings on a finite alphabet, that still (in a Turing-complete setting) can be straightforwardly represented by numbers (at worst, paying some code-obfuscation). These remarks should make clear why the evaluation of qPCF is focused on natural numbers and the standard notion of program, as any PCF-like programming language.

4 Properties of qPCF

qPCF is, morally, a PCF-like language endowed with a quantum co-processor. This co-processor allows us to execute a quantum circuits that has been designed by executing a classical control. The co-processor returns a measure (total, in the sense that we measure the whole quantum state) of a run of the given circuit on a given input, to our classical processor.

A first property of a paradigmatic programming language as qPCF is some form of subject reduction. Moreover, we prove preservation, i.e. if a well-typed term takes a step of evaluation then the resulting term is also well typed. A second property expected for a programming language is progress [23]: well-typed terms evaluation do not stuck. Roughly, a term P is stuck whenever the evaluation of P ends in a normal form, which is not a ground value.

The main complexity in this proof comes from the fact that we have infinite (two plus a family) ground types (viz. Nat, Idx, $\mathtt{circ}(\mathtt{E})$). Example 1 shows that each term can occurs in a type (in an index expression using $\mathtt{size}$). To prove preservation and progress we must unravel the mutual relationship that holds between them.

Lemma 1. *If* $B, \mathtt{x} : \tau \vdash \mathtt{M} : \sigma$ *and* $B \vdash \mathtt{N} : \tau$ *then* $B \vdash \mathtt{M}[\mathtt{N}/\mathtt{x}] : \sigma[\mathtt{N}/\mathtt{x}]$ *and, moreover, if* $\sigma = \Pi \mathtt{z}^{\tau}.\sigma'$ *then* $B \vdash \mathtt{MN} : \sigma'[\mathtt{N}/\mathtt{x}]$.

Proof. The proof follows by induction on the derivation $B, \mathtt{x} : \tau \vdash \mathtt{M} : \sigma$.

Theorem 2 (Idx-safety). *If* $\vdash \mathtt{M} : \mathsf{Idx}$ *then* $\mathtt{M} \Downarrow \underline{\mathtt{n}}$.

Proof. The proof is quite complex but it can be done by defining a suitable predicate of computability à la Tait.

Remark that Theorem 2 is stronger than both preservation and progress, in fact it immediately implies both: (i) if $\mathtt{M}$ is closed, $\vdash \mathtt{M} : \mathsf{Idx}$ and $\mathtt{M} \Downarrow \mathtt{N}$ then $\mathtt{N}$ is closed and $\vdash \mathtt{N} : \mathsf{Idx}$, and (ii) if $\mathtt{M}$ is closed, $\vdash \mathtt{M} : \mathsf{Idx}$ and $\mathtt{M} \Downarrow \mathtt{N}$ then $\mathtt{N}$ is a numeral.

Theorem 3 (Preservation). *If* $\mathtt{M}$ *is a closed term such that* $\vdash \mathtt{M} : \sigma$ *and* $\mathtt{M} \Downarrow \mathtt{N}$ *then* $\mathtt{N}$ *is a closed term such that* $\vdash \mathtt{N} : \sigma$.

Proof. The evaluation is applied only to terms typed by ground types, viz. Nat, Idx, $\mathtt{circ}(\mathtt{E})$. Indeed, the rule in Table 2 are applied to them. The proof follows by proving by mutual induction on the following statements: (i) if $\mathtt{M}$ is closed, $\vdash \mathtt{M} : \mathsf{Nat}$ and $\mathtt{M} \Downarrow \mathtt{N}$ then $\mathtt{N}$ is closed and $\vdash \mathtt{N} : \mathsf{Nat}$; and, (ii) if $\mathtt{M}$ is closed, $\vdash \mathtt{M} : \mathsf{circ}(\mathtt{E})$ and $\mathtt{M} \Downarrow \mathtt{N}$ then $\mathtt{N}$ is closed and $\vdash \mathtt{N} : \mathsf{circ}(\mathtt{E})$. The fact that we are restricting our attention on closed terms typed by ground types simplify our proof by conveniently restricting the possible cases. The proof of (i) involves only the rules (n), (s), (p), (β), ($\mathrm{if_l}$), ($\mathrm{if_r}$), (Y), (gt), (st), (m), because the others are excluded by hypothesis. The proof of (ii) involves only the rules (β), ($\mathrm{if_l}$), ($\mathrm{if_r}$), (Y), (u), (u$'$), (u$''$), ($\mathrm{r_0}$), ($\mathrm{r_1}$), ($\mathrm{r_2}$), (a), (it), because the others are excluded by hypothesis.

Likewise, a form of progress can be proved.

Theorem 4 (Progress)

- *If* $\mathtt{M}$ *is a closed term such that* $\vdash \mathtt{M} : \mathsf{Nat}$ *and* $\mathtt{M} \Downarrow \mathtt{N}$ *then* $\mathtt{N}$ *is a numeral.*
- *If* $\mathtt{M}$ *is a closed term such that* $\vdash \mathtt{M} : \mathsf{circ}(\mathtt{E})$ *and* $\mathtt{M} \Downarrow \mathtt{N}$ *then there is a numeral* $\underline{\mathtt{k}}$ *such that* $\mathtt{E} \Downarrow \underline{\mathtt{k}}$ *and* $\mathtt{N}$ *is a circuit of arity* $\underline{\mathtt{k}}$.

Proof. The proof is similar to the that of the Preservation Theorem.

Progress and preservation together tell us that a well-typed term can never reach a stuck state during evaluation.

We conclude this section with some preliminary comments about confluence. It is well-known that quantum-measures break the deterministic evolution of a quantum system. As a consequence, in presence of a measurement operator in a quantum language (equipped with an universal basis of quantum gates), one necessarily lost confluence. This loose of standard properties is typical in presence of “non classical” operators (this holds for examples also in languages including non deterministic or probabilistic choices [2,13]). Given an evaluation of a program $\mathtt{P}$, a second evaluation can ends with a different result; in particular, the results of two evaluations of a same program can be different natural numbers. Clearly, the “measurement-free” fragment of qPCF, i.e. the whole calculus minus `dMeas` is patently deterministic. Finally, one can observe that the presence of the measure does not imply the loss of the determinism, if we limits the use of qPCF to deterministic circuits (by a suitable choice of unitary operators included in $\mathcal{U}$, e.g. only swaps).

5 Examples

In this section we propose some higher-order encoding of quantum circuit families. In the following examples, we exploit the full expressive power of the language. A qPCF term can be parametric, viz. it can represent an entire (infinite) quantum circuit family. In general, given an input numeral $\underline{\mathtt{n}}$ we define a term that generates the description of the n-dimensional circuit of the family. Notice that, in some sense, circuits can be parameterized both in "horizontal" and in "vertical", that correspond to the two basic ways to built greater circuits from smaller ones, i.e. by sequential and parallel composition.

Example 2. The following term, applied to a circuit $\mathtt{C} : \mathsf{circ}(\underline{\mathtt{k}})$ and a numeral $\underline{\mathtt{n}}$, concatenates $n+1$ copies of $\mathtt{C}$.

$\mathsf{M}_{\mathsf{seq}} = \lambda \mathtt{u}^{\mathsf{circ}(\underline{\mathtt{k}})}.\lambda \underline{\mathtt{k}}^{\mathsf{Nat}}.Y \mathsf{W} \mathtt{u} \underline{\mathtt{k}} : \mathsf{circ}(\underline{\mathtt{k}}) \to \mathsf{Nat} \to \mathsf{circ}(\underline{\mathtt{k}})$, where

$\mathsf{W} = \lambda \mathtt{w}^{\sigma}.\lambda \mathtt{u}^{\mathsf{circ}(\underline{\mathtt{k}})}.\lambda \underline{\mathtt{k}}^{\mathsf{Nat}}.\mathtt{if\ x\ (u)}\ \big(\mathsf{append}\,(\mathtt{u})\,((\lambda \mathtt{y}^{\mathsf{circ}(\underline{\mathtt{k}})}.\lambda \mathtt{z}^{\mathsf{Nat}}.\mathtt{wyz})\,\mathtt{u}\,\mathtt{pred}(\mathtt{x}))\big)$

has type $\sigma \to \sigma$, with $\sigma = \mathsf{circ}(\underline{\mathtt{k}}) \to \mathsf{Nat} \to \mathsf{circ}(\underline{\mathtt{k}})$. It is easy to show that $\mathsf{M}_{\mathsf{seq}}\mathtt{C}\underline{\mathtt{0}}$ generates the circuit built upon a single copy of the circuit $\mathtt{C}$ and so on.

It is straightforward to parameterize the above term in order to transform it in a template for a circuit-builder that can be used for any arity. It suffices to replace $\underline{\mathtt{k}}$ with the variable $\mathtt{k}^{\mathsf{Idx}}$ and to abstract it; so that, the resulting term is typed $\Pi \mathtt{k}^{\mathsf{Idx}}.\,\mathsf{circ}(\mathtt{k}) \to \mathsf{Nat} \to \mathsf{circ}(\mathtt{k})$.

Example 3. The following term, applied to a numeral $\underline{\mathtt{n}}$ and two unitary gates $\mathtt{U}_1$ and $\mathtt{U}_2$ of arity $\underline{\mathtt{k}}$ and $\underline{\mathtt{h}}$ respectively, generates a simple circuit built upon a copy of gate $\mathtt{U}_1$ in parallel with n copies of gate $\mathtt{U}_2$:

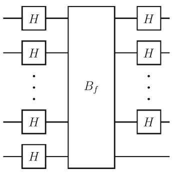
$$\mathsf{M}_{\mathsf{par}} = \lambda \mathtt{x}^{\mathsf{Idx}}.\lambda \mathtt{u}^{\mathsf{circ}(\underline{\mathtt{k}})} \lambda \mathtt{w}^{\mathsf{circ}(\underline{\mathtt{h}})}.\,\mathtt{iter\ x\,u\,w} : \Pi \mathtt{x}^{\mathsf{Idx}}.\,\mathsf{circ}(\underline{\mathtt{k}}) \to \mathsf{circ}(\underline{\mathtt{h}}) \to \mathsf{circ}(\mathtt{x} * \underline{\mathtt{h}} + \underline{\mathtt{k}})$$

It is straighforward to parameterize the above example. It suffices to replace numerals $\underline{\mathtt{k}}$ and $\underline{\mathtt{h}}$ in the above example by variables and to abstract, by obtaining a term typed $\Pi \mathtt{k}^{\mathsf{Idx}}.\Pi \mathtt{h}^{\mathsf{Idx}}.\Pi \mathtt{x}^{\mathsf{Idx}}.\,\mathsf{circ}(\mathtt{k}) \to \mathsf{circ}(\mathtt{h}) \to \mathsf{circ}(\mathtt{x} * \mathtt{h} + \mathtt{h})$.

Example 4 (Deutsch-Jozsa). We provide the qPCF encoding of the circuit that implements the generalised version of the Deutsch's problem [17].

The "simple case" of Deutsch's problem can be formulated as follows. Given a block box B_f implementing some function $f : \{0,1\} \to \{0,1\}$, determine whether f is constant or balanced. The classical computation to determine whether f is constant or balanced is very simple: one computes $f(0)$ and $f(1)$, and then check if $f(0) = f(1)$. This requires two different calls to B_f (i.e. one to compute $f(0)$ and one to compute $f(1)$) in the classical computing model. By means of the "quantum superpower", Deutsch showed how to achieve this result with a single call of B_f in the quantum case.

The problem can be generalised considering a function $f : \{0,1\}^n \to \{0,1\}$ which acts on many input bits. This yields the n-bit generalization of Deutsch's algorithm, known as the Deutsch-Josza algorithm. The following picture represents the circuit, up to the last, measurement phase, for the Deutsch-Josza problem.



When fed with a classical input state of the form $|0\dots01\rangle$, the output measurement of the first $n-1$ bits reveals if the function f is constant or not. If all $n-1$ measurement results are 0, we can conclude that the function was constant. Otherwise, if at least one of the measurement outcomes is 1, we conclude that the function was balanced. See [17] for details about Deutsch and Deutsch-Josza algorithm.

Consider now the following qPCF terms. They easily show how to encode different levels of the measurement-free parametric circuit for the Deutsch's problem. The last, evaluation-measurement phase, will be performed by our evaluation `dMeas`, suitably fed with the representation of the classical input state (i.e. $\mathtt{set}\,\underline{0}\,\underline{0}$). Let $\mathtt{H} : \mathsf{circ}(\underline{0})$ and $\mathtt{I} : \mathsf{circ}(\underline{0})$ be the (unary) Hadamard and Identity gates respectively (so the index is 0). Suppose $\mathsf{M}^{B_f} : \mathsf{circ}(\underline{\mathtt{n}})$ is given for some n such that $\mathsf{M}^{B_f} \Downarrow \mathsf{U}_f$ where $\mathsf{U}_f : \mathsf{circ}(\underline{\mathtt{n}})$ is the qPCF§-circuit that represents the function f.

Observe that $\lambda \mathrm{x}^{\mathsf{Idx}}.\,\mathtt{iter\ xHH} : \Pi \mathrm{x}^{\mathsf{Idx}}.\,\mathsf{circ}(\mathtt{x})$ clearly generates $x+1$ parallel copies of unary Hadamard gates H, and $\lambda \mathrm{x}^{\mathsf{Idx}}.\,\mathtt{iter\ xIH} : \Pi \mathrm{x}^{\mathsf{Idx}}.\,\mathsf{circ}(\mathtt{x})$ concatenates in parallel x copies of unary Hadamard gates H and one copy of the unary identity gate I.

Thus the generator term of the parametric measurement-free Deutsch-Jozsa circuit, here dubbed DJ^- can be defined as
$\mathsf{DJ}^- = \lambda \mathrm{x}^{\mathsf{Idx}}.\lambda \mathrm{y}^{\mathsf{circ(x)}}.\,\mathtt{append}(\mathtt{append}(\mathtt{iter}\ \mathtt{x\,H\,H})\mathtt{y})(\mathtt{iter}\ \mathtt{x\,I\,H}) : \sigma$ where
$\sigma = \Pi \mathrm{x}^{\mathsf{Idx}}.\,\mathsf{circ}(\mathtt{x}) \to \mathsf{circ}(\mathtt{x})$.

We finally evaluate DJ^- by means of `dMeas`, providing the encoding
$\mathsf{M}^{B_f}\ \mathsf{circ}(\underline{\mathtt{n}})$ of the black-box function f having arity $n+1$. Let us assume that the term $\mathtt{dMeas}(\mathtt{set}\underline{0}\underline{0}, \mathsf{DJ}^-\underline{\mathtt{n}}\mathsf{M}^{B_f})$ evaluated by means of $\Downarrow$, yields the numeral $\underline{\mathtt{m}}$: the rightmost $\underline{\mathtt{n}}$ digit of the binary representation of $\underline{\mathtt{m}}$ are the result. Notice that $\mathsf{DJ}^-\underline{0}\mathsf{M}^{B_f}$ returns the circuit description of Deutsch algorithm.

6 Conclusions and Future Work

We introduced qPCF, an extension of PCF for quantum circuit generation and evaluation. In this seminal work, we introduced qPCF syntax, typing rules and evaluation semantics. We started to study its properties and we provided some examples of parametric circuit families encoding. The presented research is the first step of some works in progress and for several short time investigations. First, we are further investigating qPCF properties sketched in Sect. 4. Second, we aim to deepen qPCF flexibility, e.g. studying *specialization* of qPCF: for example, we aim to focus on the (still "silent") reverse operator (of the calculus), also in different settings w.r.t quantum computing. We like to remark that gates can range on different interesting sets. Since *reversibility* is nowadays one of the most interesting trend in computer science [20], a reversible specialization of qPCF seems to be intriguing. Even if the use of total measurement does not represent a theoretical limitation, a partial measurement operator can represent an useful programming tool. Therefore, another interesting task will be to integrate in qPCF the possibility to perform partial measures of computation results.

References

1. Arrighi, P., Dowek, G.: Linear-algebraic lambda-calculus: higher-order, encodings and confluence. In: Proceedings of the 19th Annual Conference on Term Rewriting and Applications, pp. 17–31 (2008)
2. Aschieri, F., Zorzi, M.: Non-determinism, non-termination and the strong normalization of system T. In: Hasegawa, M. (ed.) TLCA 2013. LNCS, vol. 7941, pp. 31–47. Springer, Heidelberg (2013). doi:10.1007/978-3-642-38946-7_5
3. Aspinall, D., Hofmann, M.: Dependent types. In: Pierce, B. (ed.) Advanced Topics in Types and Programming Languages, Chap. 2, pp. 45–86. MIT Press (2005)
4. Dal Lago, U., Masini, A., Zorzi, M.: On a measurement-free quantum lambda calculus with classical control. Math. Struct. Comput. Sci. **19**(2), 297–335 (2009)
5. Gaboardi, M., Paolini, L., Piccolo, M.: Linearity and PCF: a semantic insight! In: Proceeding of ICFP 2011, pp. 372–384 (2011)
6. Gaboardi, M., Paolini, L., Piccolo, M.: On the reification of semantic linearity. Math. Struct. Comput. Sci. **26**(5), 829–867 (2016)
7. Grattage, J.: An overview of QML with a concrete implementation in haskell. Electron. Not. Theor. Comput. Sci. **270**(1), 165–174 (2011)
8. Hasuo, I., Hoshino, N.: Semantics of higher-order quantum computation via geometry of interaction. In: LICS 2011, pp. 237–246 (2011)
9. Kaye, P., Laflamme, R., Mosca, M.: An Introduction to Quantum Computing. Oxford University Press, Oxford (2007)
10. Knill, E.: Conventions for quantum pseudocode. Technical Report, Los Alamos National Laboratory (1996)
11. Lago, U.D., Masini, A., Zorzi, M.: Quantum implicit computational complexity. Theor. Comput. Sci. **411**(2), 377–409 (2010)
12. Lago, U.D., Masini, A., Zorzi, M.: Confluence results for a quantum lambda calculus with measurements. Electron. Not. Theor. Comput. Sci. **270**(2), 251–261 (2011)
13. Lago, U.D., Zorzi, M.: Probabilistic operational semantics for the lambda calculus. RAIRO - Theor. Inf. Appl. **46**(3), 413–450 (2012)
14. Mahmoud, M., Felty, A.P.: Formalization of Metatheory of the Quipper Programming Language in a Linear Logic. University of Ottawa, Canada (2016)
15. Masini, A., Viganò, L., Zorzi, M.: A qualitative modal representation of quantum register transformations. In: Proceedings of the 38th International Symposium on Multiple-Valued Logic (ISMVL), pp. 131–137 (2008)
16. Masini, A., Viganò, L., Zorzi, M.: Modal deduction systems for quantum state transformations. J. Multiple-Valued Logic Soft Comput. **17**(5–6), 475–519 (2011)
17. Nielsen, M.A., Chuang, I.L.: Quantum Computation and Quantum Information. Cambridge University Press, Cambridge (2000)
18. Nishimura, H., Ozawa, M.: Perfect computational equivalence between quantum turing machines and finitely generated uniform quantum circuit families. Quant. Inf. Process. **8**(1), 13–24 (2009)

19. Pagani, M., Selinger, P., Valiron, B.: Applying quantitative semantics to higher-order quantum computing. In: Proceedings of POPL 2014, pp. 647–658. ACM (2014)
20. Paolini, L., Piccolo, M., Roversi, L.: A class of reversible primitive recursive functions. Electron. Not. Theor. Comput. Sci. **322**(18605), 227–242 (2016)
21. Paolini, L., Pimentel, E., Rocca, S.R.: An operational characterization of strong normalization. In: Aceto, L., Ingólfsdóttir, A. (eds.) FoSSaCS 2006. LNCS, vol. 3921, pp. 367–381. Springer, Heidelberg (2006). doi:10.1007/11690634_25
22. Payikin, J., Rand, R., Zdancewic: QWire: A Core Language for Quantum Circuits. University of Pennsylvania, USA (2016)
23. Pierce, B.C.: Types and Programming Languages. The MIT Press, Cambridge (2002)
24. Plotkin, G.D.: LCF considered as a programming language. Theor. Comput. Sci. **5**, 223–255 (1977)
25. Roman, S.: Advanced Linear Algebra. Graduate Texts in Mathematics, vol. 135, 3rd edn. Springer, New York (2008)
26. Ross, N.: Algebraic and Logical Methods in Quantum Computation. Ph.D. thesis, Dalhousie University Halifax, Nova Scotia (2015)
27. Selinger, P.: Towards a quantum programming language. Math. Struct. Comput. Sci. **14**(4), 527–586 (2004)
28. Selinger, P., Valiron, B.: A lambda calculus for quantum computation with classical control. Math. Struct. Comput. Sci. **16**, 527–552 (2006)
29. Selinger, P., Valiron, B.: Quantum lambda calculus. In: Semantic Techniques in Quantum Computation, pp. 135–172. Cambridge University Press (2009)
30. Shende, V.V., Prasad, A.K., Markov, I.L., Hayes, J.P.: Synthesis of reversible logic circuits. IEEE Trans. Comput.-Aided Des. Integr. Circ. Syst. **22**(6), 710–722 (2003)
31. Valiron, B., Ross, N.J., Selinger, P., Alexander, D.S., Smith, J.M.: Programming the quantum future. Commun. ACM **58**(8), 52–61 (2015)
32. Viganò, L., Volpe, M., Zorzi, M.: Quantum state transformations and branching distributed temporal logic. In: Kohlenbach, U., Barceló, P., Queiroz, R. (eds.) WoLLIC 2014. LNCS, vol. 8652, pp. 1–19. Springer, Heidelberg (2014). doi:10.1007/978-3-662-44145-9_1
33. Viganò, L., Volpe, M., Zorzi, M.: A branching distributed temporal logic for reasoning about entanglement-free quantum state transformations. Inf. Comput., 1–24 (2017). http://dx.doi.org/10.1016/j.ic.2017.01.007
34. Ying, M.: Foundations of Quantum Programming. Morgan Kaufmann, Cambridge (2016)
35. Zorzi, M.: On quantum lambda calculi: a foundational perspective. Math. Struct. Comput. Sci. **26**(7), 1107–1195 (2016)