applied sciences

Article

Quantum Calculi—From Theory to Language Design

Margherita Zorzi

Department of Computer Science ,University of Verona, 37100 Verona, Italy; margherita.zorzi@univr.it

Received: 31 August 2019; Accepted: 4 December 2019; Published: 12 December 2019

Abstract: In the last 20 years, several approaches to quantum programming have been introduced. In this survey, we focus on the QRAM (Quantum Random Access Machine) architectural model. We explore the twofold perspective (theoretical and concrete) of the approach and we list the main problems one has to face in quantum language design. Moreover, we propose an overview of some interesting languages and open-source platforms for quantum programming currently available. We also provide the higher-order encoding in the functional languages qPCFand IQu of the well known Deutsch-Jozsa and Simon's algorithms.

Keywords: quantum language design; quantum computing; programming theory

1. Introduction

Quantum computers are nowadays a tangible reality. Even if physicists and engineers have to continuously address problems in the realization of quantum devices, the advance of these innovative technologies has presented a noticeable speedup. Examples of this are the universal architectures behind IBM Quantum Experience or Rigetti's Forest platform [1] that are potentially able to solve problems more efficiently than a classical computer.

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 [2–8]. The interest in the development of a quantum language of is twofold. On the one hand, it presents a theoretical challenge: a calculus able to characterize (all or an interesting class of) quantum computable functions provides the basis for a quantum computability theory. On the other hand, a solid quantum programming design is essential to design robust, powerful and easy languages, which effectively allow users to program emerging quantum architectures. In quantum language design, a computer scientist has to address (at least) the following questions: What is the Architectural Model the language refers to?; How to manage quantum data (which are non-duplicable to ensure the no-cloning property, so need a form of linear treatment)?; what class of quantum functions one aim to program (expressive power)?

In this survey, we provide some possible answers, starting from the state of art and proposing some examples of quantum programming in two functional calculi, called qPCF and IQu respectively.

Moreover, we browse the state of art of some important quantum languages, mainly focusing on the functional programming style.

The purpose of this article is not to reconstruct the entire history of quantum languages but to provide an agile introduction to (part of) the topic and the tools for further study.

Note

In this paper, we assume familiarity with basic concepts about quantum computing such as quantum bits [9], quantum state/registers [10–13], quantum data properties and quantum algorithms [14]. For the mathematical background, in particular the algebraic characterization of quantum computational spaces and operators, see for example, Reference [15].

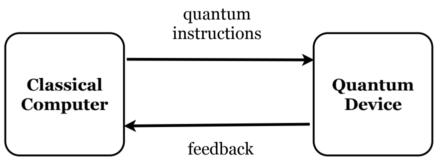
2. On Quantum Language Design: Main Features

2.1. The Architectural Model

In quantum software design, the choice of an architectural model is not as easy as for the classical case. This holds today and, in particular, held two decades ago, when quantum programming theory moves the first steps.

In this report, we mainly focus on the so-called *quantum data* & *classical control* (qd&cc) [3,16], approach, which separates the classical and quantum parts of the computation. A classical program (ideally in execution on a classical machine) computes some "directives," sent to a hypothetical quantum device that applies them to quantum data.

This idea is inspired by an architecture called Quantum Random Access Machine (QRAM), introduced by Knill [17], who noticed that the quantum circuit families [9,18], the most interesting computational model, is not enough to describe quantum algorithms. In fact, as clearly observed in Reference [19] a quantum algorithm assume an explicit "control flow" that permits to execute quantum operations, eventually measures quantum data and runs classical computation on the measured results. The following picture represents the QRAM in a simplified and intuitive way:



The QRAM model can be formulated in several version—for example, one can consider a simplified architecture, where measurement is postponed at the end of the computation [8,20]; otherwise, one can admit general measurements (in this case part of the computation possibly depends on intermediate measurement results) [3,21,22]. In a more general perspective, the natural unidirectionality of the QRAM (the classical "client" morally depends on the quantum " server") can be relaxed. This has been done in Reference [7], where the notion of *dynamic lifting* is introduced—oversimplifying the story, the classical computer is able to receive a query and send back to the quantum device the continuation of the computation. Dynamic lifting is supported, for example, by the language Quipper.

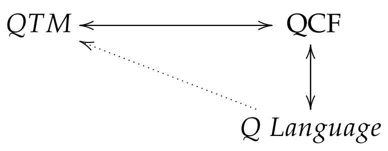
We focus on the QRAM model since the only superposition of data reflects what physicists are realizing in the development of quantum devices. Moreover, this choice based on classical control makes easier the sound description of quantum programs. In fact, it is quite easy to observe that if the set of unitary transformation clearly forms an algebra, they do not form a Hilbert space (the same holds for the set of Quantum Turing Machines).

Projects as Quipper [7], where the QRAM is definitively reformulated in terms of *quantum-coprocessor* [23], or the recently developed quantum circuit definition language QWIRE [21,24] and the Q# language by Microsoft [25] follow the qd&cc approach.Following the same direction, in Section 4.5.3 we propose two higher-order encoding examples according to syntax and operational semantics of the paradigmatic languages qPCF and IQu.

In literature can be found other approaches to quantum programming that can be considered as orthogonal to the QRAM based one. Oversimplifying, the concept of quantum superposition has been extended from data to programs, introducing notions of "sums of programs" and "quantum control". Roughly speaking, given two correct programs M and N also the combination $\alpha M + \beta N$ (where α and β are scalar coefficients) is considered meaningful as its semantical and denotational account. The origin of this approach can be traced in the lambda calculus introduced by L. Vaux in Reference [26], where the author, out from the quantum setting, extended the pure lambda-calculus by endowing the set of terms with a structure of vector spaces to study main properties and the relationship with the ordinary lambda-calculus. Based on Vaux's investigations, some versions of the (linear) algebraic and vectorial lambda calculus have been successively studied, also establishing connections with quantum computation (see for example, Reference [27]). The related notions of *superposition of program* and *quantum control* (where the control flow notions of sequencing, conditionals, loops and recursion are not entirely classical as for ordinary quantum algorithms) have been addressed in some recent papers. Even if the effective practical utility of these proposals is not yet visible, they offer new interesting perspectives on quantum computability and quantum programming theory. See for example, Reference [28] by Mingsheng and Reference [29] by Sabry, Kaiser-Vizzotto and Valiron.

2.2. On the Expressive Power of a Quantum Language

What is the expressive power of a given quantum programming language? Even then, the question can be addressed from two perspectives. For a quantum computability viewpoint, it is interesting to characterize the class of the *quantum computable functions encodable in the language*. This can be reached, for example, by establishing an expressive equivalence with another computational model, such as the Quantum Turing Machines (QTM). Since QTMs are very complex and programming on they is very challenging, the equivalence can be retraced passing through the equivalence (on total functions) between QTM and *finitely generated* Quantum Circuit families (QCF), a subclass of QCF. This has been proved, for example, in Reference [8] for the quantum lambda calculi Q. The proof is complex and the result can be roughly summarized as in Figure 1, where the continuous arrow represents a direct equivalence (for which at least a proof has been showed in literature) and the dotted arrow represents an indirect equivalence (obtained as a byproduct):



$$\begin{tikzcd} QTM \arrow[rr, leftrightarrow] & & \mathrm{QCF} \arrow[d, leftrightarrow] \\ & & Q\ Language \arrow[ull, dotted] \end{tikzcd}$$

Figure 1. Direct and indirect equivalence between quantum computational models.

The equivalence with a (sub)classes of QCFs is interesting also from a programming perspective. Some recent research trends suggests that quantum languages can be conveniently formulated and viewed as (higher-order) circuit definition languages [20,21,23,30] (see also Section 3).

Following this idea, a higher-order program parametrically encodes an entire family of circuits: when it is fed with an input dimension, it returns the correct circuit description for the given arity.

In the Quipper language [7], a neat distinction between the notion of *parameters* and *inputs* is given. Informally, parameters capture compile-time information (e.g., problem size), whereas inputs concern the run-time circuit generation.

In Section 4.5.3 we see two higher-order examples, respectively the encoding of the Deutsch-Jozsa algorithm [9] in the languages qPCF and IQu and the encoding of Simon's quantum subroutine (the central part of Simon's algorithm [9]) in the language IQu.

For other parametric encodings of quantum circuit families see also Quipper's literature and documentation in References [31,32].

2.3. On the Linearity of Quantum Data

It is well-known that quantum data are not duplicable (*no cloning property*) [33]. Therefore, the design of the type system for a quantum programming language necessarily has to address the problem of data linearity. In this section, we specifically take a look at this aspect through a quick overview of some languages in literature. A more general discussion about the state of art of quantum programming is in Section 3.

Following Selinger's suggestion in his seminal work about quantum programming theory [16], higher-order language can be equipped with some kind of linear type system along the lines of Girard's linear logic (LL), in order to account for the non-duplicability of quantum data [3,8,34–38].

Several versions of LL have been used in literature.For example, Selinger and Valiron's quantum lambda calculus [3] (on which the programming language Quipper is partially inspired) is based on the multiplicative fragment of *affine* LL, that is, contraction structural rule is not allowed. This choice is justified by the no-cloning property of quantum states and, logically, signifies that one can not duplicate resources but can discard them.

Instead, Altenkirch and Grattage [4,39] chose a different approach to the (central) problem of copy and discard quantum data. In their language QML (that can not be completely considered QRAM based but is relevant for this discussion) is possible to write expressions like $\mathsf{let}\ \mathsf{x} = (\mathsf{false} + \mathsf{true})\mathsf{in}\langle \mathsf{x}, \mathsf{x}\rangle$ which (apparently) violates no-cloning properties, since the variable x is duplicate in the let expression (notice also that a limited superposition "+" between ground value is admitted). The previous expression does not clone quantum data (this is guaranteed by the formal semantics the authors provide): it shares a copy of the quantum datum among different references. This is captured through a type system based on multiplicative *strict* LL, that is, linear logic with contraction and without weakening. We spend some further words about these calculi in Section3.

In Dal Lago et al. [34] the untyped quantum lambda calculus Q is introduced and studied. The set of well-formation rules (w.f.r.) is based on Wadler's formulation of LL and the ! ("bang") operator allows distinguishing between duplicable (classical) and non-duplicable (quantum) resources. Classical variables and quantum variables (variables "pointing to" qubits) belong to different syntactical classes. These choices statically respect the linearity of quantum data, that can neither be duplicated or erased (duplication and erasing of classical data are allowed). Moreover, the *surface reduction* is adopted, to avoid dynamic duplication of linear resources: informally, a term of the form $!(M)$ can not contain quantum variables (this is guaranteed by w.f.r.) and can not be reduced. Lambda calculus' β-reduction is modified accordingly (in particular, β-reduction is split into three different rules). A small example clarifies the design of the language and the operational behavior of programs.

Example 1 (How to violate quantum linearity)**.** *In* Q *computational steps are defined between configurations that is, triples of the form [Q,QV,M] where Q is "the quantum memory" (it keeps information about the mathematical state we are computing and qubit names), QV is the set of qubit names contained in the lambda term M. Q inherits configurations from Reference [3] (and this tradition can be retraced in Idealized ALGOL programming style [40]).*

Let us consider the following well–formed expression, where 1 represent a empty quantum state (a memory containing 0 qubits), the subterm new (1) *create a qubit and its name and cnot is the controlled not operator [9]:* $[1, \emptyset, (\lambda!x.cnot\langle x, x\rangle)!(\mathsf{new}\ (1))]$.

Observe that the sub-term $!(\mathsf{new}\ 1)$ *is a duplicable term because it is marked with the bang and, by w.f.r., does not yet contain references to quantum data. The operational behavior of* new *c, where c is a constant* 0 *or* 1 *is the following: the contraction creates a new quantum bit in the quantum register q and a new quantum variable in the lambda term (this corresponds to the preparation of the input state). We show now a correct computation, where: (i) we do not reduce in the scope of the bang; (ii) we perform firstly the c_β reduction, corresponding to classical β reduction (we are passing a duplicable argument to a λ binding a classical variable $!x$); (iii) we separately (in two steps) create two new quantum bits and the associated quantum variables by contracting subterms* new 1 *(each of them creates a different, fresh quantum data) (iv) finally we evaluate the cnot on the two qubits created in the reductions by the evaluation $\to_U$. The "link" between a qubits name and the related qubit ϕ is represented in the first element of the configuration by the notation $p \mapsto |\phi\rangle$).*

$$\begin{aligned}
[1, \emptyset, (\lambda!x.cnot\langle x, x\rangle)!(\mathsf{new}\ 1)] \quad &\to_{c_\beta} \quad [1, \emptyset, cnot\langle \mathsf{new}\ 1, \mathsf{new}\ 1\rangle] \\
&\xrightarrow{2}_{\mathsf{new}} \quad [|p \mapsto 1\rangle \otimes |q \mapsto 1\rangle, \{p, q\}, cnot\langle p, q\rangle)] \\
&\to_U \quad [|p \mapsto 1\rangle \otimes |q \mapsto 0\rangle, \{p, q\}, \langle p, q\rangle].
\end{aligned}$$

The the quantum state $|p \mapsto 1\rangle \otimes |q \mapsto 0\rangle$ represents the output of the application of the controlled-not on the state $|10\rangle$ (see Reference [9]).

However, if we reduce under the scope of the bang (namely reducing the subterm new (1) *before executing the c_β–reduction and then create a single quantum variable), we would obtain the following computation:*

$$\begin{aligned}[1, \emptyset, (\lambda!x.cnot\langle x, x\rangle)!(\mathsf{new}\,1)] &\to_{\mathsf{new}} [|p \mapsto 1\rangle, \{p\}, (\lambda!x.cnot\langle x, x\rangle)!(p)] \\ &\to_{q_\beta} [|p \mapsto 1\rangle, \{p\}, \mathbf{cnot}\langle p, p\rangle)].\end{aligned}$$

Notice that we have duplicated the quantum variable p, creating a double reference *to the same qubit. As a consequence, we could apply a binary unitary transform (cnot) to a single qubit (the one referenced by p), which is not compatible with the basic principles of quantum computing.*

The use of Linear Logic is not the only way to control the linearity of quantum data. In the last years, different solutions partially or completely based on *dependent types* have been useful in the definition of safe quantum languages. See Section3 and References [20,41].

3. Quantum Programming Languages

In recent years, we have seen an increase in quantum software platforms. Mainstream industries, such as IBM and Microsoft, invested a huge amount of resources both in software development and in the technological advance.

The world of quantum programming can claim two decades of interesting research and is growing day-by-day. In this section, we only focus on a fragment of this world and remand to the literature for a complete overview. Quantum languages can be divided into three classes: practical-minded tools for generating quantum circuit description; quantum languages with a limited (or still underdeveloped) theoretical account; theoretically attractive languages with a focus on semantics and denotational models.

To the first category belongs **Forest**, the quantum software platform developed by `Rigetti` and **Qiskit** (**Quantum Information Software Kit**), an opensource platform developed by `IBM` for the 20-qubits quantum computer *IBM Q Experience*. **Forest** includes an opensource quantum language called PyQuil embedded in Pyton and offers to the user both a small online real quantum architecture named Quantum Processing Unit (QPU) that support computations up to 8 qubits and a remote Quantum Virtual Machine (QVM) for simulation up to 30 qubits. **Qiskit** native language is OpenQASM, a low-level quantum assembly language that provides instruction to the actual quantum devices, similarly to PyQuil. `IBM` provides local and cloud-based quantum simulators (up to 25 qubits). For a complete account and comparison of open-source quantum programming platforms, see Reference [1].

The second category hosts languages such as Quipper by Selinger, Valiron et al. [42] and the **Quantum developers KIT**, based on the language Q# by `Microsoft`. Quipper represents one of the more advanced academic reality in the landscape of quantum languages. Quipper represents one of the more advanced academic reality in the landscape of quantum languages. Quipper is embedded (an embedded language provide a "quantum core" sophistically interfaced with a host "classical" language) in Haskell and is based on the QRAM model, successively rephrased in terms of *quantum co-processors* in Reference [23]. Quipper is higher-order, scalable, permits to easily write programs using ancillae qubits [9], allows to compile classical programs into quantum circuits and simulate in Haskell quantum computations. Moreover, the language pursues the goal to optimize the number of resources needed for computations. Also to this end, in Quipper a neat separation between compile-time inputs (called *parameters*) and run-time inputs (sometimes called *states*) is explicit. Quipper does not have a dedicated type system and so lacks both linear and dependent types (see Section 2.3) and does not support a denotational semantics. For this reason, in the present survey, we will focus only on a class of Quipper sub-languages called Proto − Quipper.

Q#, recently introduced by `Microsoft`, is a stand-alone (i.e., non-embedded) domain-specific programming language used for expressing quantum algorithms. It allows us to write sub-programs that execute on an adjunct quantum processor under the control of a classical host program (in this sense, Q# is a QRAM based language) and also includes a resource estimator. Literature about theoretical properties of Q# is still missing. Despite some feature such as the use of polymorphic types being very interesting, in this paper we do not go deeper in the discussion and remand to the on-line documentation [25].

The third category is the one we focus on in Section 4.

4. Foundational Quantum Programming Languages

In this section, we discuss some quantum languages called here "foundational" since they have been deeply studied from a theoretical viewpoint. In particular, as previously said, we focus on some QRAM based functional calculi, reserving Section 4.6 to provide references to other (interesting or historically important) languages we do not address here.

4.1. Quantum Lambda Calculi

In Reference [16], Selinger rigorously defined a first-order quantum functional language. This paper represents a milestone for a part of the calculi developed in the following years and definitively founds the QRAM-based programming model, where only data (qubits) are superposed and where programs live in a standard classical world. Subsequently the author, in joint work with Valiron [3], defined a quantum λ-calculus with classical control, here dubbed λ_{sv}, that we have briefly discussed in Section 2.3.

The main goal of Selinger and Valiron's work is to give the basis of a *typed* quantum functional language, based on a call-by-value λ–calculus enriched with constants for unitary transformations and an explicit measurement operator which allows the program to observe the value of one qubit.

Reductions are defined between *program states*, whose definition is similar to the one of configuration (Section 2.3). Because of the presence of measurement, the authors provide an operational semantics based on a probabilistic call-by-value evaluation. The type system of λ_{sv} is based on linear logic (linearity is extended to higher types), by distinguishing between duplicable and non-duplicable resources. The type system of λ_{sv} is based on linear logic (linearity is extended to higher types), by distinguishing between duplicable and non-duplicable resources.

Selinger and Valiron's type syntax is the following: $A, B ::= \alpha | X | !A | A \multimap B | \top | A \otimes B$ where α ranges over a set of type constants, X ranges over a countable set of type variables, $\multimap$ represents the linear implication and $\top$ is the linear unit type.

As previously said the authors restrict the system to be *affine*, that is, *contraction structural rule is not allowed*. The type system avoids run-time errors and is also equipped with subtyping rules, which provide a more refined control of the resources. λ_{sv} enjoys some good properties such as subject reduction and progress and a strong notion of *safety*. The authors also define a new interesting quantum type inference algorithm, based on the idea that a linear (quantum) type can be viewed as a decoration of an intuitionistic one.

An example of λ_{sv} well-typed term, encoding the EPR circuit is λ_{sv} as $EPR = \lambda x.CNOT\langle H(\mathsf{nw}(0)), \mathsf{nw}(0)\rangle$ and has type $!(\top \multimap (\mathsf{qbit} \otimes \mathsf{qbit}))$ where qbit is the base type of qubits. The `nw` constant behaves like the new constant used in Section 2.3.

λ_{sv}'s reductions are performed in a call-by-value setting. This is a central characteristic since different strategies are not equivalent (see the original paper or Reference [8]). Moreover, a mixed strategy can produce an hill-formed term.

In Reference [43], the authors provide a categorical semantics for λ_{sv}. For some recent investigations on semantic models, see Reference [44]. See also Reference [45], where the authors define a particle-style geometry of interaction [46] for λ_{sv}.

The pure quantum lambda calculus Q we employ in Example 1 is morally an heir of λ_{sv}, despite the fact that it has been introduced for different scopes. In particular, the "quantum Turing completeness" (i.e., the equivalence with the Quantum Turing Machines) of Q has been proved, in joint with properties of the set of well forming rules (such as Subject Reduction) and a notion of *standardization*, that ideally aims to capture the separation between classical and quantum part of the computation. Oversimplifying, Standardization Theorem says that a correct computation of a well-formed program always corresponds a computation where reduction steps are performed in the following order: first, one reduces classical redexes (this corresponds to the generation of the suitable quantum circuit among the entire infinite quantum circuit family according to the input dimension one provides); second, one reduces subterms of the shape $\mathbf{new}(c)$ (see Example 1); finally, subterms of the shape $U\langle q_1, \dots, q_k\rangle)$, where U is a constant representing a unitary operator and $\langle q_1, \dots, q_k\rangle$ is a pattern of quantum variable representing qubits in the quantum register (this corresponds to the evaluation of the circuit generated at stage 1 on the input prepared at stage 2). Different versions of this idea have been implicitly used in several following proposals, in particular for systems designed as quantum circuit generator languages. For the quantum lambda calculus Q two other sibling systems have been defined: SQ [35], a sub-calculus of Q intrinsically quantum polytime (where *soft linear logic* is used as a basis for the w.f.r. set) and for Q* [36] and extension of Q with a partial measurement operator. Differently from λ_{sv}, in Q^*, no strategy is imposed and a form of confluence (based on a notion of probabilistic distributions of observable results) is proved. For a complete account about the "Q-Family", see Reference [8].

An interesting development of Selinger and Valiron's quantum lambda calculus has been developed in Reference [37], where the author propose a fully expressive lambda calculus (λ_{sv} does not allow to encode an entire infinite quantum circuit family) and provide a denotational semantics for higher-order programs based on Girard's quantitative semantics.

4.2. QML and the QIO-Monad

Another seminal contribution in the definition of quantum functional calculi is QML, the typed quantum language for finite quantum computations we briefly compared with λ_{sv} in Section 2.3. QML has been developed in References [4,39,47,48].

QML permits to encode quantum algorithms easily. More precisely, a single term can encode a quantum circuit, that is, captures an algorithm of type $Q^k \to Q^k$, where Q is the type of the qubits for a given $k \in \mathsf{Nat}$. The syntax allows building expressions such as αt, where α is an amplitude and t is a term or expressions like $t + u$ where the symbol "+" represents the superposition of terms. However, superposition is controlled by a restrictive notion of "orthogonality" between terms, defined employing a notion of inner product between judgments [4]. Intuitively, $t \perp u$ holds when t and u are "distinguishable in some way" [39]: in other words, one can derive the judgement $\mathsf{true} \perp \mathsf{false}$ from $t \perp u$ by means of orthogonality rules. Orthogonality of judgments is automatically inferred by static analysis of QML's terms [48].

Quantum superposition can be combined with the conditional construct. The syntax includes both the if then else and the quantum conditional if° then else ; the quantum conditional if° then else is allowed when the values in the branches are orthogonal. For example, the following term represents the encoding of the Hadamard gate in QML: $\mathsf{had}\ x = \mathsf{if}^\circ x\ \mathsf{then}\ (\frac{1}{\sqrt{2}}\mathsf{false} + (-\frac{1}{\sqrt{2}})\mathsf{true})\ \mathsf{else}\ (\frac{1}{\sqrt{2}}\mathsf{false} + \frac{1}{\sqrt{2}}\mathsf{true})$. The denotational semantics of QML has been defined in terms of *superoperators*.

Starting from QML and as a step toward higher-order quantum programming, Altenkirch et.al successively introduced the Quantum IO-monad [44]. The QIO-monad is a functional interface to quantum programming, implemented as a Haskell library and provides a constructive semantics for quantum programming. The main idea to Quantum IO- monad is to split reversible (i.e., unitary, purely quantum) and irreversible (i.e., probabilistic) computations and provides a reversible let operation, supports the use of ancillas (i.e., auxiliary qubits). Various central quantum procedures (e.g., the complete implementation of Shor's algorithm) are implemented.

As pointed out in Reference [19], the QIO-monad represents the early step towards embedding quantum computation inside a functional host language, the key idea behind languages as Quipper and QWIRE. See also Green's Ph.D. thesis, where the author embedded the Quantum IO-Monad inside Agda, a dependently-typed programming language, showing the usefulness of dependent types in the setting of quantum languages. Dependent types prevent some instances of qubit cloning and represent an important tool to address quantum data linearity (see the language QWIRE and qPCF below).

4.3. Proto − Quipper

Ross defines the first version of Proto − Quipper in Reference [49] (see also Section 2.3, where we spend some word about its LL-based type system). Proto − Quipper has been introduced as a "bridge" between quantum lambda calculi Quipper, to isolate a well-typed and safe core of the practical language. The syntax and the type system of Ross' Proto − Quipper are inspired to λ_{sv}, with some important additions. A central feature is a boxing-unboxing mechanism for quantum circuits. Informally, the *box* function turns a function describing a linear operation into a circuit regarded as a classical data (a dual function *unbox* performs the reverse operation). A boxed circuit can be used multiple times as a part of a larger circuit. This avoids useless circuit duplication and has a positive impact on resource consumptions. Moreover, the syntax is extended with constant terms to capture useful circuit-level operations, like reversing.

Linear types are extended with a specific type $Circ(T, U)$ (where T and U represents the input-output set of wires) of circuits. Proto − Quipper is type-safe (Subject Reduction and Progress Theorems hold) and allows a notion of subtyping similar to the one defined for λ_{sv}. The operational semantics of Proto − Quipper is defined between configurations, as in λ_{sv} and Q. Proto − Quipper can be considered a quantum circuit description language, as QWIRE, qPCF and IQu we describe in the following Sections. Differently from QWIRE, qPCF and IQu, Proto − Quipper is not Turing complete but expressive enough to be an interesting formal core of Quipper. Another version of Proto − Quipper called Proto − Quipper-M has been introduced by Selinger and Rios in Reference [30]. Proto − Quipper-M mainly focuses on denotational models (an aspect we did not address in this paper), so we remand to the original paper.

Proto − Quipper has been also investigated in Reference [50], where Mahmoud and Felty formalize the semantics of the language by encoding the typing and evaluation rules in linear description logic, to reason about the linear type system of the "father" language Quipper.

For a further insight on Proto − Quipper-like languages denotation see also the Combined Linear/Non Linear calculus (CLNL) by Lindenhovious et al. [51], a version of Benton's LNL calculus enriched with string diagrams, a tool that has found applications across a range of areas in computer science and in particular in quantum circuit description languages. LNL calculus also inspired the language QWIRE we introduce in the following section and the denotational model for a language similar to QWIRE based on enriched categories proposed by Staton and Renella in Reference [52].

4.4. QWIRE

QWIRE (*"choir"*) [19,21,24,53] is a powerful and flexible language for writing *verified* quantum programs. As Quipper, QWIRE is based on the QRAM. The "quantum core" (called *circuit language*) of QWIRE can be treated as a "quantum plugin" for a host classical language. In its current version it is embedded in COQ proof assistant [19,24] and is a verified circuit generation language. This is reflected by the type system, inspired to Benton's LNL Logic that partitions the exponential data into a purely linear fragment and a purely non-linear fragment connected via a categorical adjunction (notice that this fully reflects also the QRAM structure). This choice also makes the "quantum core" strongly independent from the host language. The type system of the circuit language essentially controls the well formation of expressions concerning *wires*, that is, the circuit's inputs/outputs. The type system of the host language also controls the boxing mechanism (similarly to Quipper, a circuit can be "boxed" and then promoted as a classical resource/code). QWIRE supports *dinamic lifting*, that allows the quantum computer to initialize a residual computation in a continuation-passing style. Dynamic lifting is an integral part of many quantum algorithms, including quantum error correction but it is also inefficient because the quantum computer must remain suspended (and must continuously undergo error correction to prevent degradation), waiting for the remainder of the circuit to be computed.

For this reason, QWIRE also supports *static* lifting, which models the fact that there is no residual quantum state left on the quantum computer after a run and can be used when dynamic lifting is no essential (as in the quantum teleportation encoding).

Differently from Quipper, QWIRE is type-safe and supports both linear and dependent types. A strong type system guarantees the safety of generated circuits. In some sense, QWIRE design inherit and develop the better of the two perspectives we often pointed out in this paper: on the one hand good properties of quantum lambda calculi/paradigmatic languages are preserved; on the other hand, as Quipper it is designed for realistic computations.

QWIRE is an ongoing project and nowadays the main effort is devoted to efficiency, formal verification and optimization of quantum programs, a research area still underdeveloped. See Reference [19] for a complete account about the use of lifting operations and several techniques (safe semantics, automatic type-checking, circuit denotation. . .) designed for QWIRE.

4.5. qPCF *and* IQu

In this section we describe two (stand-alone) functional languages called qPCF and IQu respectively. See References [20,41] for the complete study of qPCF and see Reference [22] for the current version of IQu. Both qPCF and IQu follow the QRAM-based approach, support the direct manipulation of quantum circuits and enjoy properties such as *Preservation* and *Progress* Theorems. W.r.t. our previous proposals [8], strongly oriented to the characterization of quantum computable functions, qPCF IQu are programming oriented calculi. They have been designed to make quantum programming as easier as possible also for a programmer with low/medium skills in quantum computing theory. This scope is eased by the gate based subtended model of computation, that allows representing the quantum part of the program (the one offloaded to the quantum coprocessor in the QRAM architecture) as a quantum circuit. Both qPCF and IQu can be foundational for concrete languages. In particular, as a medium time goal, we are proceeding with the implementation of the ALGOL-like language IQu.

4.5.1. qPCF

The language qPCF is based on a version of the QRAM model where measurements are all performed at the end of the computation. qPCF is a simple extension of Plotkin's PCF: essentially, it extends PCF with a new kind of classical data, quantum circuits, that can be, thanks to the type system, treated as classical objects (i.e., freely duplicated). In qPCF linear logic-based typing has been completely avoided, in favor of the use of a simple version of *dependency*, that permits to soundly encode quantum circuits families directly controlling arities of subcircuits, avoiding illegal erasing or duplication of wires. qPCF types includes: the types of quantum circuits $\mathsf{circ}(E)$ (where E that morally represents a circuit dimension), the type of *indexes* (i.e., strongly normalizing expressions) Idx and the (standard) quantification over types $\Pi x.\tau$.

The syntax of qPCF allows us to easily manipulate circuits through operations such as sequentialization and parallelization of an arbitrary number of sub-circuits. For example, consider the program $\mathsf{M_{seq}} = \lambda \mathtt{u}^{\mathsf{circ}(\underline{\mathtt{k}})}.\lambda \mathtt{x}^{\mathsf{Nat}}.Y\mathsf{W}\mathtt{ux} : \mathsf{circ}(\underline{\mathtt{k}}) \to \mathsf{Nat} \to \mathsf{circ}(\underline{\mathtt{k}}) : \sigma \to \sigma$ where Y is the recursion operation, $\mathsf{W} = \lambda \mathtt{w}^{\sigma}.\lambda \mathtt{u}^{\mathsf{circ}(\underline{\mathtt{k}})}.\lambda \mathtt{y}^{\mathsf{Nat}}.\,\mathtt{if}\ \mathtt{y}\ (\mathtt{u})\ (\fatsemi\,(\mathtt{u})\,(\mathtt{w}\,\mathtt{u}\,(\mathtt{pred}\,\mathtt{y})))$ and $\sigma = \mathsf{circ}(\underline{\mathtt{k}}) \to \mathsf{Nat} \to \mathsf{circ}(\underline{\mathtt{k}})$. Given a term representing a circuit (of ariety $\underline{\mathtt{k}}$) $\mathtt{C} : \mathsf{circ}(\underline{\mathtt{k}})$, it is easy to observe that $\mathsf{M_{seq}}$ applied to $\mathtt{C}$ and $\underline{\mathtt{n}}$ concatenates $n+1$ copies of $\mathtt{C}$.

The parallel composition of quantum circuit can be achieved by using the operator `iter` (that has type $\Pi \mathtt{x}^{\mathsf{Idx}}.\,\mathsf{circ}(\mathtt{E}_0) \to \mathsf{circ}(\mathtt{E}_1) \to \mathsf{circ}(\mathtt{E}_0 + ((1+\mathtt{E}_1) * \mathtt{x})))$.

Consider the program $\mathsf{M_{par}} = \lambda \mathtt{x}^{\mathsf{Idx}}.\lambda \mathtt{u}^{\mathsf{circ}(\underline{\mathtt{k}})}\lambda \mathtt{w}^{\mathsf{circ}(\underline{\mathtt{h}})}.\,\mathtt{iter}\ \mathtt{x}\,\mathtt{u}\,\mathtt{w} : \Pi \mathtt{x}^{\mathsf{Idx}}.\,\mathsf{circ}(\underline{\mathtt{k}}) \to \mathsf{circ}(\underline{\mathtt{h}}) \to \mathsf{circ}(\underline{\mathtt{k}} + (\mathtt{x} * (\underline{\mathtt{h}} + 1)))$. When applied to a numeral $\underline{\mathtt{n}}$ and two unitary gates $\mathtt{U}_1 : \mathsf{circ}(\underline{\mathtt{k}})$ and $\mathtt{U}_2 : \mathsf{circ}(\underline{\mathtt{h}})$, $\mathsf{M_{par}}$ generates a circuit built upon a copy of gate $\mathtt{U}_1$ in parallel with n copies of gate $\mathtt{U}_2$. Notice that the term is driven by an argument of type Idx, to ensure that iteration is strong normalizing and, consequently, that the arity of the generated circuit is always a numeral.

qPCF makes especially convenient the programming of quantum algorithms in *deferred form*, where measurements are all postponed at the end of the computation. qPCF quantum states are not stored. This is possible since the interaction with the quantum co-processor is neatly decoupled using the operator dmeas. It offloads a quantum circuit to a co-processor for the evaluation which is immediately followed by a (von Neumann) Total Measurement. This means that partial measures are forbidden. Thanks to the deferred measurement principle [9], this restriction does not represent a theoretical limitation. Nevertheless, general measurement is an useful programming tool and we model it in the language IQu described in Section 4.5.2.

4.5.2. IQu

IQu (read "Haiku" as the Japanese poetic form) extends Reynold's Idealized Algol, the core of Algol-like languages [40]. Idealized Algol combines the fundamental features of procedural languages, that is, local stores and assignments, with a fully-fledged higher-order procedure mechanism which, in its turn, conservatively includes PCF. We exploited Idealized Algol features to provide a (as much as possible) minimal extension to capture higher-order circuit generation.

In IQu enjoys a simple type theory: classical types (for natural numbers, command and classical variables) are extended with two new types.

The first one, circ, is the type of quantum circuits. Quantum circuits are classical data, so their type allows to operate on them without any special care. In IQu all circuits/gates $U^{\underline{k}}$, of any dimension k, are typed with the unique circuit type circ. The second one, qVar, types quantum variables (representing quantum content of registers). Since the manipulation of quantum registers requires care, we adapt Idealized Algol's original de-reference mechanism to access the content of classical and quantum variables. Registers can not be duplicated but we can access their classical content via suitable methods and, if interested in that, we can duplicate that content. On the other and, the type of "quantum variables" prevents to read quantum states but allows to measure them.

This reference mechanism allows modeling the no-cloning property of quantum data therefore in IQu we use neither dependent types or linear typer for describing quantum circuits, to keep it as simple as possible.

As for qPCF, basic circuit manipulations (concatenation and parallelization) are easy encodable thanks to recursion and ad hoc operators denoted as $\fatsemi$ and $\|$ respectively.

Evaluation in IQu, as in Idealized ALGOL, and in quantum languages such as References [3,8], is defined between pairs $(\overline{s}, M)$, where s is a store, that is, a function that links quantum and classical variables to classical data and quantum registers respectively and M is a term. We possibly represent quantum stores as pairs $\{r, |\phi\rangle\}$, where r is the name of the register and $|\phi\rangle$ is the related quantum state. A fresh quantum register of an arbitrary dimension k can be easily created and bound in a program M by means of the command $\mathsf{qnew}^{\mathtt{N}}$ `x in M`, where `N` is expected to reduce to $\underline{k}$.

The function rsize returns the arity of a quantum register. The expression $\mathtt{x} \lhd \mathtt{N}$ evaluates the application of the circuit `N` to the quantum state stored in `x`, then it stores the resulting state in `x`.

Finally, $\mathsf{meas}^{\mathtt{N}}$ `x` measures `N` qubits of a quantum state which `x` stores (and, update such state, in accordance with the quantum measurement rules).

Formally, IQu does not extend qPCF. However, such an extension is possible, thus we state that IQu extends qPCF with classical and quantum stores. Moreover, since IQu supports general measurement, the programmer does not necessarily encode algorithms in deferred form as, for qPCF.

4.5.3. Programming Quantum Algorithms: Three Higher-Order Examples

In this section, we use qPCF and IQu to provide three higher-order examples of quantum circuit family encodings. The examples are parametric, whereas the majority of the examples proposed in the literature are linear: this means that a term (the program) represents an entire (infinite) quantum circuit family. When fed with the input dimension, the evaluation of the program morally first yields the description of the circuit of the right size among the whole family and then returns to the user the evaluation of the circuit on the quantum state provided as the input of the computation.

We show the encodings of the well-known Deutsch-Jozsa and Simon's algorithms. We implement the circuit representing the Deutsch-Jozsa procedure in qPCF and IQu, to see at work the main peculiarities of the languages and the different solutions we chose in the definition of the syntax, the type and the reduction systems. Finally, we propose the encoding of the circuit family implementing Simon's quantum subroutine in IQu, exploiting its partial measurement operator and the quantum store.

Example 2 (Deutsch-Jozsa Circuit in qPCF)**.** *Figure 2 represents, up to the last phase (measurement of the output state), the circuit implementing the well-known Deutsch-Josza algorithm (the generalization of the Deutsch's algorithm) [9], that considers a function $f : \{0,1\}^n \to \{0,1\}$ which acts on many input bits:*

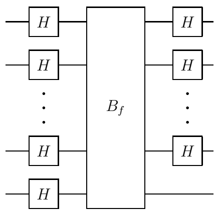


Figure 2. Circuit representation of Deutsch-Jozsa algorithm.

Given a classical input state of the form $|0\ldots01\rangle$, the circuit returns a state that, once measured, reveals in the first $n-1$ bits if the function f is constant or balanced. If all $n-1$ measurement results are 0, we can conclude that the function is constant. Otherwise, if at least one of the measurement outcomes is 1, we conclude that the function is balanced.

In qPCF, unitary gates of arity $k+1$ are typed with the (dependent) type $circ(\underline{k})$, where the dependency parameter $\underline{i}$ in type $circ(\underline{i})$ is the numeral representation of the integer i. Moreover, the special type Idx is reserved for strongly normalizing expressions that represent dimensions of operation on circuits, such as sequentialization and parallelization. Sequential composition of two circuits C_1 and C_2 (of th same arity k) is denoted as $C_1 \fatsemi C_2$. The qPCF term $\mathtt{iter}$ put in parallel an arbitrary number of quantum gates of arbitrary ariety: for example, $\mathtt{iter}\,\underline{k}UV$ create a parallel composition of k copies of the gate V and a copy of the gate U.

Let $\mathtt{H} : circ(\underline{0})$ and $\mathtt{I} : circ(\underline{0})$ be the (unary) Hadamard and Identity gates respectively. Suppose $\mathsf{M}^{B_f} : circ(\underline{n})$ is given for some n such that $\mathsf{M}^{B_f} \Downarrow^{\alpha} \mathsf{U}_f$ where $\mathsf{U}_f : circ(\underline{n})$ is the qPCF-circuit that represents the the black-box function f having arity $n+1$ and $\Downarrow^{\alpha}$ represents the evaluation labelled with its probability (in this case, $\alpha = 1$).

The term $\lambda \mathrm{x}^{Idx}.\,\mathtt{iter}\ \mathrm{xHH} : \Pi \mathrm{x}^{Idx}.\,circ(\mathrm{x})$ generates $x+1$ parallel copies of Hadamard gates $\mathtt{H}$ and $\lambda \mathrm{x}^{Idx}.\,\mathtt{iter}\ \mathrm{xIH} : \Pi \mathrm{x}^{Idx}.\,circ(\mathrm{x})$ concatenates in parallel x copies of Hadamard gates $\mathtt{H}$ and one copy of the identity gate $\mathtt{I}$. Notice that we abstract on types by means of the standard abstraction Π of dependent types. Thus the parametric measurement-free Deutsch-Jozsa circuit can be defined as

$$\mathsf{DJ}^{-} = \lambda \mathrm{x}^{Idx}.\lambda \mathrm{y}^{circ(\mathrm{x})}.((\mathtt{iter}\ \mathrm{x}\,\mathtt{H}\,\mathtt{H}) \fatsemi \mathrm{y}) \fatsemi (\mathtt{iter}\ \mathrm{x}\,\mathtt{I}\,\mathtt{H}) : \sigma,$$

where $\sigma = \Pi \mathrm{x}^{Idx}.\,circ(\mathrm{x}) \to circ(\mathrm{x})$.

The last phase is performed by the operator $\mathtt{dMeas}$, the qPCF function that takes as inputs the representation (as numeral) of a classical state and the description of a circuit, applies the circuit on the state and returns a possible, probabilistic results. $\mathtt{dMeas}$ is here suitably fed with the $\mathbf{n}(\underbrace{0\ldots0}_{n}1)$ and $\mathsf{DJ}^{-}\underline{\mathrm{n}}\mathsf{M}^{B_f}$. Notice that $\mathsf{DJ}^{-}\underline{\mathrm{n}}\mathsf{M}^{B_f}$ returns the Deutsch-Jozsa circuit of dimension $n+1$ for the function f among the infinite family.

The evaluation yields $\mathtt{dMeas}(\mathbf{n}(\underbrace{0\ldots0}_{n}1), \mathsf{DJ}^{-}\underline{\mathrm{n}}\mathsf{M}^{B_f}) \Downarrow^{\alpha} \underline{\mathrm{m}}$ where $\underline{\mathrm{m}}$ is the result (with probability $\alpha = 1$).

We can make the term parametric also w.r.t. the sub-circuit represented by M^{B_f}*. It suffices to replace* $\underline{\mathtt{n}}$ *with the variable* $\mathtt{n}^{Idx}$*, to replace the black-box with a variable* $\mathtt{b}^{circ(\mathtt{n})}$ *so that, the resulting term is typed* $\Pi\mathtt{n}^{Idx}.\Pi\mathtt{b}^{circ(\mathtt{n})}.Nat$ *or more simply* $\Pi\mathtt{n}^{Idx}.\ circ(\mathtt{n}) \to Nat$ *(where* Nat *is the usual type of integers).*

Example 3 (Deutsch-Jozsa Circuit in IQu). *We show how an IQu term represents the infinite family of quantum programs that encode the Deutsch-Jozsa algorithm (Figure 2), as we have just done with qPCF.*

Let $\mathtt{H}^{\underline{1}}$: circ *be the Hadamard gate and* $\mathtt{Id}^{\underline{1}}$: circ *be the Identity gate. Notice that we "decorate" IQu constants representing unitary operators with their arities and, as described in the previous section, all gates/circuits are typed with the ground type* circ*. We implement Deutsch-Jozsa in IQu by sequentially composing the terms* M_1*,* $\mathtt{x}$ *and* M_3*, where* $\mathtt{x}$: 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* $\mathtt{C}$: circ *and to a numeral* $\underline{\mathtt{n}}$ *puts* $n+1$ *copies of* $\mathtt{C}$ *in parallel. It is defined as* $\mathsf{M}_{par} = \lambda \mathtt{u}^{\mathsf{circ}}.\lambda \mathtt{k}^{Nat}.Y\mathsf{W}_1\mathtt{uk} : circ(\to)\ Nat \to \mathsf{circ}$*, where* Y *is the recursion operator,* W_1 *is the term* $\lambda \mathtt{w}^{\sigma}.\lambda \mathtt{u}^{\mathsf{circ}}.\lambda \mathtt{k}^{Nat}.\ \mathit{if}\ \mathtt{k}\ (\mathtt{u})\ (\mathtt{u}\ \parallel\ (\mathtt{w\,u\,pred(k)}))$ *whose type is* $\sigma \to \sigma$ *with* $\sigma = \mathsf{circ} \to Nat \to \mathsf{circ}$.
- *The circuit* M_1 : circ *is obtained by feeding the term* M_{par} *with two inputs: the (unary) Hadamard gate* $\mathtt{H}^{\underline{1}}$ *and the input dimension* $\mathit{rsize}(\mathtt{r})$ *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* M_3 : circ *can be defined as* $(\mathsf{M}_{par}\mathtt{H}^{\underline{1}}\ \mathtt{pred}(\mathit{rsize}(\mathtt{r}))) \parallel \mathtt{Id}^{\underline{1}}$: circ*, that is, it is obtained by the parallel composition of the term* M_{par} *fed by the gate* $\mathtt{H}^{\underline{1}}$ *and the dimension* $\mathtt{pred}(\mathit{rsize}(\mathtt{r}))$ *(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* $qnew^{n}\ \mathtt{r}\ \mathtt{in}\ P$ *that makes the quantum variable* $\mathtt{r}$ *available in* P*. The local variable declaration* $qnew^{n}\ \mathtt{r}\ \mathtt{in}\ P$ *creates a quantum register which is fully initialized to 0. Since the expected input state of Deutsch-Jozsa circuit is* $|\underbrace{0\ldots0}_{n}1\rangle$*, we define and use an initializing circuit* $\mathsf{M}_{\mathsf{init}} = (\mathsf{M}_{par}\mathtt{Id}^{\underline{1}}(\mathtt{pred}(\mathit{rsize}(r)))) \parallel \mathtt{Not}^{\underline{1}}$: circ *that complements the last qubit, setting it to* 1 *(*$\mathtt{pred}$ *is the standard predecessor function of PCF and* $\mathit{rsize}(r)$ *extracts the size, that is, the numeber of quantum bits available in the register). Let* DJ^{+} *be the circuit* $\mathsf{M}_{\mathsf{init}} \fatsemi \mathsf{M}_1 \fatsemi \mathtt{x} \fatsemi \mathsf{M}_3$*. The (parametric) IQu encoding of the Deutsch-Jozsa algorithm can be defined as* $\lambda \mathtt{x}^{\mathsf{circ}}.\ qnew^{n+1}\ \mathtt{r}\ \mathtt{in}\ ((\mathtt{r} \lhd \mathsf{DJ}^{+}); meas^{\underline{n}}\ \mathtt{r})$*. Given an input dimension* n *and an encoding of the function* f *to evaluate, the program solves any instance of the Deutsch-Jozsa algorithm.*

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 suitable evaluation rule of IQu, we have* $\{(\mathtt{r}, |\underbrace{0\ldots0}_{n}\rangle)\}, \mathtt{r} \lhd \mathsf{DJ}^{\star} \Downarrow_1 \{(\mathtt{r}, |\phi\rangle)\}, \mathit{skip}$ *where* $|\phi\rangle$ *is the computational state after the evaluation of* $\mathsf{DJ}^{\star}$*. To measure the state* $|\phi\rangle$ *we use the operational rule for the constructor* $meas^{\underline{\mathtt{n}}}\ \mathtt{r}$ *to conclude* $\{\mathtt{r}, |\phi\rangle\}, meas^{\underline{\mathtt{n}}}\ \mathtt{r} \Downarrow_1 \{\mathtt{r}, |\phi'\rangle\}, \underline{\mathtt{k}}$*, where* $\underline{\mathtt{k}}$ *is the (deterministic) output of the measurement and* 1 *is the associated probability.*

Example 4 (Simon's algorithm in IQu). *Simon's quantum algorithm is an important precursor to Shor's algorithm for integer factorization. Simon's algorithm [54] solves in quantum polynomial-time a classically hard problem [55] which can be formulated as follows. Let be* $f : \{0,1\}^n \to X$ *(*X *finite) a black-box function. Determine the string* $\mathsf{s} = s_1 s_2 \ldots s_k$ *such that* $f(x) = f(y)$ *if and only if* $x = y$ *or* $x = y \oplus \mathsf{s}$*. 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. We here focus on the inherently quantum relevant fragment of Simon's algorithm [56]. The circuit in Figure 3 implements the quantum subroutine of Simon's algorithm.*

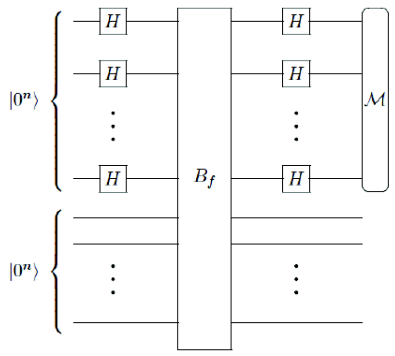


Figure 3. Circuit representation of Simon's algorithm.

The whole procedure can be easily encoded in IQu*, thanks to the partial measurement operator and the possibility to store intermediate quantum states.*

Simon's quantum 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 the figure above).* M_1 *and* M_3 *are defined by letting* $\mathsf{M}_1 = \mathsf{M}_3 = (\mathsf{M}_{par}(\mathtt{H}^{\underline{1}})\,\mathtt{rsize}(\mathtt{r})) \parallel (\mathsf{M}_{par}(\mathtt{Id}^{\underline{1}})\,\mathtt{rsize}(\mathtt{r})) : \mathsf{circ}$ *where: (i)* M_{par} *is the term that sequentializes an arbitrary number of copy the same gate (easily definable by the* Y *operator), (ii)* $\mathtt{r}$ *is a quantum register; and, (iii)* $\mathtt{H}^{\underline{1}} : \mathsf{circ}$*,* $\mathtt{Id}^{\underline{1}} : \mathsf{circ}$ *are the unary Hadamard and Identity gates, respectively.*

Let $\mathsf{M_{SP}}^{+}$ *be the circuit* $\mathsf{M}_1 \fatsemi x \fatsemi \mathsf{M}_3 : \mathsf{circ}$ *. Let* n *be the arity of* f *we want to check. The program that implements Simon's subroutine can be encoded as* $\lambda \mathtt{x}^{\mathsf{circ}}.\ \mathsf{qnew}^{2*n}\ \mathtt{r}\ \mathtt{in}\ ((\mathtt{r} \lhd \mathsf{M_{SP}}^{+}); \mathsf{meas}^{\underline{n}}\,\mathtt{r})$*, 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* $\mathsf{M_{SP}}^{\star}$ *be* $\mathsf{M_{SP}}^{+}[\mathsf{M}_{B_f}/\mathtt{x}]$*, namely the circuit obtained by the substitution of* M_{B_f} *for* $\mathtt{x}$ *in* $\mathsf{M_{SP}}^{+}$*.*

The following evaluation respects the IQu *semantics:*

$$\{(\mathtt{r}, |\underbrace{0\ldots0}_{2*n}\rangle)\}, \mathtt{r} \lhd \mathsf{M_{SP}}^{\star} \Downarrow_1 \{(\mathtt{r}, |\phi\rangle)\}, \mathtt{skip} \quad ,$$

where $|\phi\rangle$ *is the state after the evaluation of the circuit* $\mathsf{M_{SP}}^{\star}$*. We can measure the first* $\underline{\mathtt{n}}$ *quantum bits as follows:* $\{(\mathtt{r}, |\phi\rangle)\}, \mathsf{meas}^{\underline{n}}\,\mathtt{r} \Downarrow_\alpha \{(\mathtt{r}, |\phi'\rangle)\}, \underline{\mathtt{k}}$*, where* ϕ' *is one possible state after the partial measurement and* α *is the related probability.*

The classical output $\underline{\mathtt{k}}$ *can be used as 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.*

4.6. Other Quantum Calculi

In literature, several imperative approach to quantum computation have been proposed. The Quantum Computation Language (QCL), designed and implemented by Omer [57], is based on the syntax of C programming language. Its interpreter is implemented using a simulation library for executing quantum programs on a classical computer and it has been used as a code generator for classical machine controlling a quantum circuit. Along with QCL several other imperative quantum programming languages were proposed, see for example, References [58] and [2].

Starting from the seminal paper, Reference [59], some *measurement based* calculi have been defined. In particular, the so-called *measurement calculus* [5] has been developed as an efficient rewriting system for measurement-based quantum computation. In Reference [5], the authors defined a calculus of local equations for 1-qubit one-way quantum computing. Roughly speaking, the idea is that a computation is built out of three basic commands, *entanglement*, *measurement* and *local correction*. The authors define a suitable syntax for these primitives, which permits the description of patterns, that is, sequences of basic commands with qubits as input-output. By pattern composition, it is possible to implement quantum gates and quantum protocols. Moreover, a standardization theorem, which has some important consequences, is stated and proved.

5. Conclusions

Quantum programming theory is a dynamic and vital research field. In the last twenty years, several languages have been introduced and implemented. Quantum lambda calculi provided the basis for quantum computability, establishing the equivalence with other computational models. Foundational languages helped in developing ad hoc type theories for the sound management of data according to quantum mechanics principles. More practical languages showed how to program simulated and real small devices, in the perspective of the availability of a universal, powerful quantum device.

Different design choices bring different benefits and weaknesses. The choice of the QRAM architectural model, which assumes a strong separation between data and control, allows to efficiently manage the restrictions due to programming with quantum data. Moreover, it seems to be the more realistic architecture for physical implementation. Concerning the design of the type system, in this paper, we met several solutions. Linear Logic (LL) perhaps remains the most natural option for quantum type systems, since LL modalities allow to syntactically distinguish between duplicable and non-duplicable resources and reduction strategies can easily preserve the no-cloning properties of quantum data. Nevertheless, different solutions such as the use of dependent types are emerging as valid alternatives. This holds both for "stand-alone" languages (such as qPCF) and for embedded languages, in which the quantum core is purely linear and neatly separated from the "host" part, as happens in QWIRE and Quipper. This classification can be central to the scope of a language. On the one hand, a stand-alone calculus can represent a better solution if one is interested in theoretical studies since good properties (of the type system or the operational semantics) do not depend on the properties of the host language. On the other hand, an embedded language allows more direct control of quantum operations and is suitable for the implementation in automatic verification systems.

Concerning the denotational semantics of quantum languages, an argument we did not address in this paper, many fascinating theoretical results have been shown extending to the quantum setting different techniques and tools (as the Geometry of the Interaction or several categorical frameworks) and various challenging questions remain open [44,45,52,60].

We conclude that the theoretical results of the last decades and the current technological development seem to happily converge in the same direction. The advances of quantum technologists, that year-by-year provide more efficient architectures, also suggest how quantum computers have to be programmed. For example, the NISQ (Noisy Intermediate-Scale Quantum) technology, available shortly, is providing important steps toward bigger and fault-tolerant computers and could early become a concrete reference for realistic software design [61]. Thanks to this parallel effort, today we can design increasingly efficient languages and run quantum algorithms on both simulators and real prototypes [1]. At the same time, formal verification of quantum programs (e.g., based on logic [62,63] or on mechanizable tools [64]) became an urgent and exciting challenge. We are entering the quantum future and we are ready to program it.

Funding: This research received no external funding.

Acknowledgments: I wish to dedicate this paper, that embraces one of the passions of my life, to four very special people that, day-by-day, make me happy: my beloved-hippie-super-mom Franca (the genius of the family), my brother-friend-uncle Franco (the cub of the family), my friend-confidant-maître Andrea Masini (the Don Quixotte of our Department of Computer Science) and my best friend Gionata (who, in the last 20 years, has shared with me secrets, drinks and football-related joys and sorrows). Thanks, dears, for all. Long and prosper life.

Conflicts of Interest: The author declares no conflict of interest.

References

1. LaRose, R. Overview and Comparison of Gate Level Quantum Software Platforms. *Quantum* **2019**, *3*, 1–24.
2. Sanders, J.W.; Zuliani, P. Quantum Programming. In Proceedings of the 5th International Conference on Mathematics of Program Construction—MPC 2000, Ponte de Lima, Portugal, 3–5 July 2000; Lecture Notes in Computer Science; Backhouse, R.C., Oliveira, J.N., Eds.; Springer: Berlin/Heidelberg, Germany, 2000, Volume 1837, pp. 80–99. doi:10.1007/10722010_6.
3. Selinger, P.; Valiron, B. A lambda calculus for quantum computation with classical control. *Math. Struct. Comput. Sci.* **2006**, *16*, 527–552. doi:10.1017/S0960129506005238.
4. Altenkirch, T.; Grattage, J. A Functional Quantum Programming Language. In Proceedings of the 20th IEEE Symposium on Logic in Computer Science (LICS 2005), Chicago, IL, USA, 26–29 June 2005; pp. 249–258. doi:10.1109/LICS.2005.1.
5. Danos, V.; Kashefi, E.; Panangaden, P. The Measurement Calculus. *J. ACM* **2007**, *54*. doi:10.1145/1219092.1219096.
6. Díaz-Caro, A.; Arrighi, P.; Gadella, M.; Grattage, J. Measurements and Confluence in Quantum Lambda Calculi With Explicit Qubits. *Electr. Notes Theor. Comput. Sci.* **2011**, *270*, 59–74. doi:10.1016/j.entcs.2011.01.006.
7. Green, A.S.; Lumsdaine, P.L.; Ross, N.J.; Selinger, P.; Valiron, B. Quipper: A Scalable Quantum Programming Language. In Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation, Seattle, WA, USA, 16–19 June 2013; ACM: New York, NY, USA, 2013; pp. 333–342. doi:10.1145/2491956.2462177.
8. Zorzi, M. On quantum lambda calculi: a foundational perspective. *Math. Struct. Comput. Sci.* **2016**, *26*, 1107–1195. doi:10.1017/S0960129514000425.
9. Nielsen, M.A.; Chuang, I.L. *Quantum Computation and Quantum Information*, 10th Anniversary ed.; Cambridge University Press: Cambridge, UK, 2010; pp. xxvi+676. doi:10.1017/CBO9780511976667.
10. Masini, A.; Viganò, L.; Zorzi, M. Modal Deduction Systems for Quantum State Transformations. *Multiple-Valu. Logic Soft Comput.* **2011**, *17*, 475–519.
11. Viganò, L.; Volpe, M.; Zorzi, M. Quantum state transformations and branching distributed temporal logic. In *Logic, Language, Information, and Computation, Proceedings of the 21st International Workshop, WoLLIC 2014, Valparaíso, Chile, 1–4 September 2014*; Lecture Notes in Computer Science 8652; Springer: Berlin/Heidelberg, Germany, 2014; pp. 1–19. doi:10.1007/978-3-662-44145-9_1.
12. Viganò, L.; Volpe, M.; Zorzi, M. A branching distributed temporal logic for reasoning about entanglement-free quantum state transformations. *Inf. Comput.* **2017**, *255*, 311–333. doi:10.1016/j.ic.2017.01.007.
13. Masini, A.; Zorzi, M. A Logic for Quantum Register Measurements. *Axioms* **2019**, *8*, 25. doi:10.3390/axioms8010025.
14. Nakahara, M.; Ohmi, T. *Quantum Computing—From Linear Algebra to Physical Realizations*; CRC Press: Boca Raton, FL, USA, 2008.
15. Roman, S. *Advanced Linear Algebra*, 3rd ed.; Graduate Texts in Mathematics; Springer: New York, NY, USA, 2008; Volume 135, pp. xviii+522.
16. Selinger, P. Towards a Quantum Programming Language. *Math. Struct. Comput. Sci.* **2004**, *14*, 527–586. doi:10.1017/S0960129504004256.
17. Knill, E. *Conventions for Quantum Pseudocode*; Technical Report; Los Alamos National Laboratory: Los Alamos, NM, USA, 1996.
18. Nishimura, H.; Ozawa, M. Perfect computational equivalence between quantum Turing machines and finitely generated uniform quantum circuit families. *Quant. Inf. Process.* **2009**, *8*, 13–24. doi:10.1007/s11128-008-0091-8.
19. Rand, R. Formally Verified Quantum Programming. Ph.D. Thesis, University of Pennsylvania, Philadelphia, PA, USA, 2018. Available online: http://www.cis.upenn.edu/~rrand/thesis.pdf (accessed on 1 June 2019).
20. Paolini, L.; Zorzi, M. qPCF: A language for quantum circuit computations. In *Theory and Applications of Models of Computation*; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2017; Volume 10185, pp. 455–469. doi:10.1007/978-3-319-55911-7_33.
21. Paykin, J.; Rand, R.; Zdancewic, S. QWIRE: A Core Language for Quantum Circuits. In Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, Paris, France, 15–21 January 2017; ACM: New York, NY, USA, 2017; pp. 846–858. doi:10.1145/3009837.3009894.
22. Paolini, L.; Roversi, L.; Zorzi, M. Quantum programming made easy. In Proceedings of the Joint International Workshop on Linearity & Trends in Linear Logic and Applications, Linearity-TLLA@FLoC 2018, Oxford,

UK, 7–8 July 2018; EPTCS; Ehrhard, T., Fernández, M., de Paiva, V., de Falco, L.T., Eds.; 2018; Volume 292, pp. 133–147, Open Publishing Association, Waterloo, Australia . doi:10.4204/EPTCS.292.8.

23. Valiron, B.; Ross, N.J.; Selinger, P.; Alexander, D.S.; Smith, J.M. Programming the Quantum Future. *Commun. ACM* **2015**, *58*, 52–61. doi:10.1145/2699415.
24. Rand, R.; Paykin, J.; Zdancewic, S. QWIRE Practice: Formal Verification of Quantum Circuits in Coq. In Proceedings of the 14th International Conference on Quantum Physics and Logic, Nijmegen, The Netherlands, 3–7 July 2017; Volume 266, pp. 119–132. doi:10.4204/EPTCS.266.8.
25. Q# Reference Page, Technical Report, Microsoft. Available online: https://docs.microsoft.com/en-us/quantum/language/?view=qsharp-preview (accessed on 1st June, 2019).
26. Vaux, L. The algebraic lambda calculus. *Math. Struct. Comput. Sci.* **2009**, *19*, 1029–1059. doi:10.1017/S0960129509990089.
27. Arrighi, P.; Díaz-Caro, A. Scalar System F for Linear-Algebraic Lambda-Calculus: Towards a Quantum Physical Logic. *Log. Methods Comput. Sci.* **2012**, *8*. doi:10.2168/LMCS-8(1:11)2012.
28. Ying, M. *Foundations of Quantum Programming*; Morgan Kaufmann: Burlington, MA, USA, 2016.
29. Sabry, A.; Valiron, B.; Vizzotto, J.K. From Symmetric Pattern-Matching to Quantum Control. In *Foundations of Software Science and Computation Structures*; Baier, C., Dal Lago, U., Eds.; Springer International Publishing: Cham, Switzerland, 2018; pp. 348–364.
30. Rios, F.; Selinger, P. A Categorical Model for a Quantum Circuit Description Language. In Proceedings of the 14th International Conference on Quantum Physics and Logic, Nijmegen, The Netherlands, 3–7 July 2017; Open Publishing Association, Waterloo, Australia, 2018; Volume 266, pp. 164–178. doi:10.4204/EPTCS.266.11.
31. Quipper Reference Page; Technical Report. Available online: https://www.mathstat.dal.ca/~selinger/quipper/ (accessed on 1 June 2019).
32. Siddiqui, S.; Mohammed Jahirul Islam, O.S. *Five Quantum Algorithms Using Quipper*; Technical Report; Shahjalal University of Science and Technology: Sylhet, Bangladesh; University of Maryland: Baltimore County, MD, USA, 2014.
33. Park, J.L. The concept of transition in quantum mechanics. *Found. Phys.* **1970**, *1*, 23–33. doi:10.1007/BF00708652.
34. Dal Lago, U.; Masini, A.; Zorzi, M. On a Measurement-free Quantum Lambda Calculus with Classical Control. *Math. Struct. Comput. Sci.* **2009**, *19*, 297–335. doi:10.1017/S096012950800741X.
35. Dal Lago, U.; Masini, A.; Zorzi, M. Quantum implicit computational complexity. *Theor. Comput. Sci.* **2010**, *411*, 377–409. doi:10.1016/j.tcs.2009.07.045.
36. Dal Lago, U.; Masini, A.; Zorzi, M. Confluence Results for a Quantum Lambda Calculus with Measurements. *Electr. Notes Theor. Comput. Sci.* **2011**, *270*, 251–261. doi:10.1016/j.entcs.2011.01.035.
37. Pagani, M.; Selinger, P.; Valiron, B. Applying quantitative semantics to higher-order quantum computing. In Proceedings of the POPL '14, San Diego, CA, USA, 22–24 January 2014; pp. 647–658. doi:10.1145/2535838.2535879.
38. Dal Lago, U.; Zorzi, M. Wave-Style Token Machines and Quantum Lambda Calculi. In Proceedings of the Third International Workshop on Linearity—LINEARITY 2014, Vienna, Austria, 13 July 2014; Electronic Proceedings in Theoretical Computer Science 176; Open Publishing Association: Waterloo, Australia, 2014; pp. 64–78. doi:10.4204/EPTCS.176.6.
39. Grattage, J. QML: A Functional Quantum Programming Language. Ph.D. Thesis, University of Nottingham, Nottingham, UK, 2006.
40. O'Hearn, P.W. *Algol-like Languages*; Progress in Theoretical Computer Science; Birkhäuser: Basel, Switzerland.
41. Paolini, L.; Piccolo, M.; Zorzi, M. QPCF: Higher-Order Languages and Quantum Circuits. *J. Automat. Reason.* **2019**. doi:10.1007/s10817-019-09518-y.
42. Green, A.S.; Lumsdaine, P.L.; Ross, N.J.; Selinger, P.; Valiron, B. An Introduction to Quantum Programming in Quipper. In Proceedings of the 5th International Conference on Reversible Computation, Victoria, BC, Canada, 4–5 July 2013; Springer: Berlin/Heidelberg, Germany, 2013; pp. 110–124. doi:10.1007/978-3-642-38986-3_10.
43. Selinger, P.; Valiron, B. On a Fully Abstract Model for a Quantum Linear Functional Language: (Extended Abstract). *Electr. Notes Theor. Comput. Sci.* **2008**, *210*, 123–137. doi:10.1016/j.entcs.2008.04.022.
44. Makie, I.; Gay, S., Eds. *Semantic Techniques in Quantum Computation*; Cambridge University Press: Cambridge, UK, 2009. doi:10.1017/CBO9781139193313.

45. Hasuo, I.; Hoshino, N. Semantics of higher-order quantum computation via geometry of interaction. In Proceedings of the LICS'11, IEEE Computer Society, Washington, DC, USA, 21–24 June 2011; pp. 237–246.
46. Abramsky, S.; Haghverdi, E.; Scott, P.J. Geometry of Interaction and Linear Combinatory Algebras. *Math. Struct. Comput. Sci.* **2002**, *12*, 625–665. doi:10.1017/S0960129502003730.
47. Altenkirch, T.; Grattage, J.; Vizzotto, J.K.; Sabry, A. An Algebra of Pure Quantum Programming. *Electron. Notes Theoret. Comput. Sci.* **2007**, *170*, 23–47.
48. Grattage, J. An Overview of QML With a Concrete Implementation in Haskell. *Electr. Notes Theor. Comput. Sci.* **2011**, *270*, 165–174. doi:10.1016/j.entcs.2011.01.015.
49. Ross, N.J. Algebraic and Logical Methods in Quantum Computation. Ph.D. Thesis, Department of Mathematics and Statistics, Dalhousie University, Halifax, NS, Camada, 2015.
50. Mahmoud, M.Y.; Felty, A.P. Formalization of Metatheory of the Quipper Quantum Programming Language in a Linear Logic. *J. Autom. Reason.* **2019**, *63*, 967–1002. doi:10.1007/s10817-019-09527-x.
51. Lindenhovius, B.; Mislove, M.W.; Zamdzhiev, V. Enriching a Linear/Non-linear Lambda Calculus: A Programming Language for String Diagrams. *arXiv* **2018**, arXiv:1804.09822.
52. Rennela, M.; Staton, S. Classical Control and Quantum Circuits in Enriched Category Theory. *Electron. Notes Theoret. Comput. Sci.* **2018**, *336*, 257–279. doi:10.1016/j.entcs.2018.03.027.
53. Rand, R.; Paykin, J.; Lee, D.H.; Zdancewic, S. Reqwire: Reasoning about reversible quantum circuits. In Proceedings of the 15th International Conference on Quantum Physics and Logic—QPL 2018, Dalhousie University, Halifax, NS, Canada, 3–7 June 2018; Volume 287, pp. 299–312. doi:10.4204/EPTCS.287.17.
54. Simon, D.R. On the Power of Quantum Computation. *SIAM J. Comput.* **1994**, *26*, 116–123. doi:10.1137/S0097539796298637.
55. Arora, S.; Barak, B. *Computational Complexity: A Modern Approach*, 1st ed.; Cambridge University Press: New York, NY, USA, 2009. doi:10.1017/CBO9780511804090.
56. Kaye, P.; Laflamme, R.; Mosca, M. *An Introduction to Quantum Computing*; Oxford University Press: Oxford, UK, 2007; pp. xii+274.
57. Omer, B. Structured Quantum Programming. Ph.D. Thesis, Vienna University of Technology, Wien, Austria, 2003.
58. Bettelli, S.; Serafini, L.; Calarco, T. Toward an architecture for quantum programming. *arXiv* **2001**, arXiv:cs/0103009.
59. Nielsen, M. Universal quantum computation using only projective measurement, quantum memory, and preparation of the 0 state. *Phys. Lett.* **2003**, *308*, 96–100.
60. Clairambault, P.; De Visme, M.; Winskel, G. Game Semantics for Quantum Programming. *Proc. ACM Program. Lang.* **2019**, *3*, 32:1–32:29. doi:10.1145/3290345.
61. Preskill, J. Quantum Computing in the NISQ era and beyond. *Quantum* **2018**, *79*. doi:10.22331/q-2018-08-06-79.
62. Baltag, A.; Smets, S. Quantum logic as a dynamic logic. *Synthese* **2011**, *179*, 285–306. doi:10.1007/s11229-010-9783-6.
63. Ying, M.; Ying, S.; Wu, X. Invariants of quantum programs: Characterisations and generation. In Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, 18–20 January 2017; Castagna, G., Gordon, A.D., Eds.; ACM: New York, NY, USA, 2017; pp. 818–832. doi:10.1145/3009837.
64. Amy, M. Towards large-scale functional verification of universal quantum circuits. In Proceedings of the 15th International Conference on Quantum Physics and Logic, QPL 2018, Dalhousie University, Halifax, NS, Canada, 3–7 June 2018; Volume 287, pp. 1–21. doi:10.4204/EPTCS.287.1.