Compiling Quantum Lambda-Terms Into Circuits via the Geometry of Interaction

Kostia Chardonnet[1], Ugo Dal Lago[2], Naohiko Hoshino[3] and Paolo Pistone[4]

[1]*Université de Lorraine, CNRS, Inria, LORIA, France*
[2]*University of Bologna, Italy*
[3]*Sojo University, Japan*
[4]*Université Claude Bernard Lyon 1, France*

Abstract

We present an algorithm turning any term of a linear quantum λ-calculus into a quantum circuit. The essential ingredient behind the proposed algorithm is Girard's geometry of interaction, which, differently from its well-known uses from the literature, is here leveraged to perform as much of the *classical* computation as possible, at the same time producing a circuit that, when evaluated, performs all the *quantum* operations the evaluation of the underlying λ-term requires. We identify *higher-order control flow* as the primary obstacle towards efficient solutions to the problem at hand. Notably, geometry of interaction proves sufficiently flexible to enable efficient compilation in many cases, while still supporting a total compilation procedure. Finally, we characterize through a type system a broad class of λ-terms for which compilation can be performed in polynomial time.

Keywords

Lambda-calculus, geometry of interaction, quantum computing

1. Introduction

Quantum computing holds immense potential to revolutionize computer science by solving complex problems faster than classical computing [1, 2], leveraging qubits via *superposition* and *entanglement*. Multiple architectures are explored, including gate-based models, quantum annealing [3], and topological quantum computing [4]. On the programming side, the QRAM model [5] is prominent: a classical machine interacts with a quantum device by allocating qubits, applying unitary operations, and measuring results. Programs thus mix classical control flow with quantum data access, yielding probabilistic outcomes due to the presence of measurements.

Several quantum programming languages instantiate this idea, from assembly-like systems [6, 7], to imperative languages [8], to functional calculi [9, 10, 11]. They combine classical computation with external quantum interaction, allowing arbitrary classical processing between quantum calls, and expose measurement results as first-class Booleans for control flow. However, modern hardware is not naturally QRAM-based: it requires whole circuits to be submitted, which are then optimized via transpilation. Hence the mainstream quantum programming languages are circuit description languages such as `Qiskit` [12], `Cirq` [13], or `Quipper` [14, 15], where circuits are built as data structures. This enables offline optimization but removes intermediate measurement visibility, except for partial workarounds like dynamic lifting [16].

This gap raises the question of whether QRAM programs can be efficiently compiled into circuits while maximizing classical computation beforehand and handling measurements and conditionals effectively and correctly, especially when higher-order control flow is involved, which complicates direct circuit extraction and may cause exponential blow-ups.

ICTCS 2026: Italian Conference on Theoretical Computer Science, September 7–9, 2026, Udine, Italy
kostia.chardonnet@protonmail.com (K. Chardonnet); ugo.dallago@unibo.it (U. Dal Lago); nhoshino@cis.sojo-u.ac.jp (N. Hoshino); paolo.pistone@ens-lyon.fr (P. Pistone)
https://kostiachardonnet.github.io (K. Chardonnet); https://udallago.github.io (U. Dal Lago); https://sites.google.com/view/naohikohoshino/home (N. Hoshino); https://perso.ens-lyon.fr/paolo.pistone/ (P. Pistone)

Contributions. We present a compilation procedure for the linear fragment of a quantum λ-calculus [9, 17] into a QASM-like circuit model [6], eliminating higher-order operations and classical control while extracting equivalent first-order circuits. The key ingredient is Girard's *Geometry of Interaction* (GoI) [18, 19], which is well-known to induce abstract machines [20, 21, 22] and circuit synthesis methods [23]. Here, geometry of interaction is used in a novel way: given a derivation $\Gamma \vdash M : A$, tokens traverse type occurrences, producing a circuit connecting inputs and outputs by following data-flow paths. For example, $x : \mathit{qubit}, y : \mathit{qubit} \vdash M : (\mathit{qubit} \multimap \mathit{qubit}) \multimap \mathit{qubit}$ yields a circuit with three input and two output qubits, even if M contains higher-order constructs.

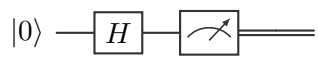
Our compilation procedure consists of two phases: GoI first translates typing derivations into circuits *with* classical control flow; a second step then *removes* control flow, yielding standard quantum circuits. While the second phase takes linear-time, the first is computationally harder, due to higher-order conditionals, whose branches may themselves have higher-order types. Naïve asynchronous handling is sound but can lead to exponential blow-up, whereas synchronous GoI-based execution yields compact circuits which may deadlock. Our algorithm combines the two approaches: it applies the synchronous strategy whenever possible, falling back to the asynchronous one only when necessary. Without asynchronous fallback, size remains linear and compilation can be performed in polynomial time.

Besides the definition of the compilation procedure based on an abstract machine, we prove its soundness via a simulation with respect to an alternative GoI semantics of quantum λ-calculi [24], connected through completely positive maps semantics [25] to our machine. Finally, we characterize a class of source terms that avoid deadlocks via a type system ensuring synchronous compilation is always applicable. This system is both sound and complete, guaranteeing polynomial-time compilation for well-typed programs in this fragment.

2. A Bird's Eye View on the Problem

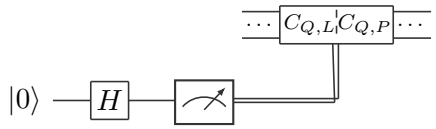
In this section we provide an overview on the problem addressed, focusing on the challenges it poses from a computational viewpoint. Our goal is to translate a term M written in a linear quantum λ-calculus [9] onto a quantum circuit C_M [26], namely, a finite sequence of operations for creating, modifying and measuring qubits, possibly carrying around the result of previous measurements for the sake of conditioning the applications of subsequent quantum operations. Observe that M can manipulate higher-order functions, while C_M has a genuinely first-order nature.

A first obvious idea would be to try to compile M via rewriting or abstract machines [27, 28, 22], adapting such approaches so that evaluating a quantum instruction occurring in M has the effect of modifying an incrementally-built circuit, rather than explicitly acting on the quantum data. For example, suppose that M is the term $\texttt{let}\ x = S\ \texttt{in}\ Q$, where S is $\texttt{if}\ N\ \texttt{then}\ L\ \texttt{else}\ P$. Suppose further that the branches L and P have type $\texttt{qbit} \multimap \texttt{qbit}$, and that the guard N is a term of Boolean type whose value is produced through some form of quantum computation, e.g., N is $\texttt{meas}(\texttt{H}\ (\texttt{new}\ \texttt{ff}))$, while L and P are $\lambda x.x$ and $\lambda x.\texttt{H}(x)$, respectively. (Here, H stands for the so-called *Hadamard* gate, while new and meas initializes and measure a qubit, respectively.) Now, suppose we want to compile M into an equivalent quantum circuit C_M. When evaluating N, such an abstract machine would (correctly) not perform any quantum operations, but rather produce a circuit like the following one:



As the Boolean produced by N depends on the result of a measurement, its value would not be known at circuit-*building* time, but only at circuit-*execution* time. Therefore, *both* branches of $S = \texttt{if}\ N\ \texttt{then}\ L\ \texttt{else}\ P$ should be compiled. Yet, the two branches have *functional* type, and compiling the continuation Q only once would be non-trivial, because x occurs free in it, and it has function type. The situation would be even more complicated if L and P had even higher order type. Since we want to keep our abstract machine compliant with the underlying reduction semantics, it is thus natural to proceed by compiling $Q[x \leftarrow L]$ and $Q[x \leftarrow P]$ into two circuits $C_{Q,L}$ and $C_{Q,P}$. The overall circuit

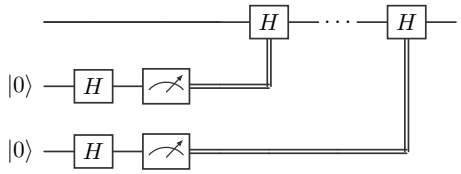
constructed would then be as the circuit below, where the box at the top is a conditional gate with branches $C_{Q,L}$ and $C_{Q,P}$:



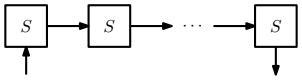
In this way, however, we eventually incur into an exponential blowup. Indeed, suppose we generalize the above example to a family of terms $M_n = R_n^n$, where

$$R_0^m \coloneqq \lambda x.x_1(x_2(\ldots(x_m x))). \qquad R_{n+1}^m \coloneqq \texttt{let}\, x_n = S\, \texttt{in}\, R_n^m;$$

It is clear that, by proceeding as above while compiling M_n, we would obtain a circuit of size *exponential* in n, consisting of n levels of nested conditionals, although the size of M_n is linear in n. Still, compiling M_n is not *intrinsically* difficult, since a much simpler circuit that captures the quantum operations performed by M_n can be constructed namely the following:



The compilation technique presented in this paper produces, on input M_n, *precisely* the circuit described above. The key idea is to view compilation as a form of *data-flow analysis*, supported by the geometry of interaction. This proceeds by letting tokens follow occurrences of base types in the underlining type derivation. When applied to the term M_n, this gives rise to a data-flow graph similar to the following one

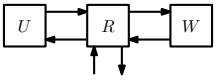


There are here n nodes, each corresponding to an occurrence of the conditional S. Edges here model data dependencies. Please observe that there are no circularities here. The compilation of M_n involves $n+1$ tokens. The first n tokens each correspond to the n occurrences of N, are local to one of the S nodes, and have a simple and short-lived behavior. The last token, instead, traverses the n occurrences of S. The handling of conditionals is here done as follows: when *all* the incoming tokens are available, the two branches are compiled in a recursive way, resulting in a conditional at the circuit level. This is precisely the already mentioned *synchronous* rule: by tracing token paths *hierarchically*, starting from the conditional branches placed at highest depth, and then moving upwards, the synchronous rule turns such *higher-order control flow* onto plain, *circuit-level* conditionals, yielding in the end compact circuits like the one above. Remarkably, no exponential blowup occurs.

Is this the end of the story? Unfortunately, no: as anticipated, this synchronous method is not *always* applicable. Consider, for instance, the term:

$$R \coloneqq \texttt{if}\ N\ \texttt{then}\ (\lambda f.\lambda g.\lambda x.f(gx))\ \texttt{else}\ (\lambda f.\lambda g.\lambda x.g(fx)).$$

The two branches of the conditional are here of *second-order* type $(\texttt{qbit} \multimap \texttt{qbit}) \multimap (\texttt{qbit} \multimap \texttt{qbit}) \multimap (\texttt{qbit} \multimap \texttt{qbit})$. Let U and W be quantum gates. If one were to apply the aforementioned compilation scheme to the term $R\,U\,W$, one would obtain a data dependency graph like the one below:



A form of *deadlock* arises, due to the presence of a circular dependency between the conditional R and the two gates U and W. Observe that deciding whether compiling U *before* W or viceversa depends on how the conditional in R evaluates. Moreover, compiling the circuits underlying the two branches of R requires knowing, *in advance*, to which gates and in which order the variables f and g will be applied, and thus to have already compiled both U and W.

It should be noted that this limitation is not as much due to our chosen architecture, as it is, in fact, *unavoidable*: it can be shown that no quantum circuit containing only a single occurrence of both U and W is equivalent to the term above. To be a bit more precise, there is no quantum circuit C with three holes $[-]$, $\{-\}$ and $\lfloor - \rfloor$ (where each of $[-]$, $\{-\}$ and $\lfloor - \rfloor$ occurs exactly once) such that for any Boolean value x and quantum gates U and W, the circuits $C[x]\{U\}\lfloor W \rfloor$ and $\texttt{if}\ x\ \texttt{then}\ W; U\ \texttt{else}\ U; W$ are equivalent. (For more detail and proof of impossibility, see [29]). We give an elementary proof of this result for quantum gates U and W acting on two qubits in the Appendix, available as Supplementary Material.

How can we overcome this *impasse*? Is it possible to compile the family of terms M_n efficiently, while at the same time handling satisfactorily deadlock phenomena such as those occurring in the term R? The answer is affirmative, and consists in complementing the synchronous rule mentioned above with another, *asynchronous*, rule that essentially follows the natural (but inefficient) idea of breaking deadlocks through duplication. The resulting machine is on the one hand capable of compiling any term of the source λ-calculus, and, on the other hand, efficient in those cases in which circular dependencies like those just highlighted do *not* occur.

3. The Main Ingredients of the Introduced Token Machine

The core contribution of the paper is the **QCSIAM** abstract machine. It extends Girard's classical *Geometry of Interaction* (GoI) to quantum λ-calculi, mapping functional programs directly into quantum circuits. Rather than giving a formal definition of the programming language under consideration, of the **QCSIAM** abstract machine, and of the quantum circuits it produces, in this section we provide an informal overview of these three components. The purpose is to give the reader an intuitive understanding of the framework, while more details are available in [30].

3.1. The Programs

As anticipated in the previous section, the programs we consider are written in a linear fragment [10] of the well-known quantum λ-calculus introduced by Selinger and Valiron [17, 9], one of the most influential higher-order quantum programming languages. The linear type discipline reflects the no-cloning principle of quantum mechanics by treating qubits as linear resources that cannot be duplicated or discarded arbitrarily. The language provides the standard primitives required for quantum computation. Qubits can be initialized through a `new` construct, transformed by applying unitary operations, and eventually measured through the `meas` operator. Measurements yield classical Boolean values, which can subsequently be manipulated using ordinary conditional expressions. The type system also includes tensor product types, constructed by means of the operator $\otimes$, and linear function types, constructed through the operator $\multimap$. The operational semantics of the language can be naturally formulated in terms of *quantum closures*, which combine classical program terms with the associated quantum state.

3.2. The Circuits

The target of our compilation procedure consists of standard quantum circuits. Such circuits are made of quantum gates connected by wires carrying either quantum bits or classical bits, depending on the kind of information being manipulated. Quantum wires transport qubits, whereas classical wires carry the outcomes of measurements and are used to control the subsequent computation. For technical convenience, we assume that the circuit language also provides a conditional construct, allowing

the circuit itself to branch according to classical values. This assumption considerably simplifies the presentation of the machine and its correctness proof. As shown separately, however, this construct is not fundamentally required, since conditional circuits can eventually be compiled away into standard quantum circuits without loss of expressive power.

3.3. The Machine

The **QCSIAM** machine is inspired by the multitoken machines developed within the Geometry of Interaction framework. Unlike traditional GoI machines, whose execution directly evaluates the underlying λ-term, **QCSIAM** separates the classical and quantum aspects of the computation. The machine executes only the classical control flow, while quantum operations are delayed and progressively accumulated into a quantum circuit representing the quantum component of the computation. The GoI-inspired execution model makes this separation particularly natural. Whenever all the tokens corresponding to the inputs of a quantum gate become simultaneously available, the gate is not immediately applied to a quantum state. Instead, it is appended to the circuit currently being constructed, together with the appropriate wiring information. As a consequence, the machine incrementally generates a circuit while simultaneously evaluating the classical part of the program. This mechanism is straightforward and efficient whenever the interpreted program, although possibly higher-order, does not involve conditional constructs, or when conditionals do not interact with higher-order functions. As discussed in the previous section, the combination of higher-order control flow and conditionals introduces substantial complications. **QCSIAM** addresses these situations through two complementary transition rules:

- **Asynchronous rule.** An asynchronous transition is always applicable. It immediately maps branching in the source program into branching in the generated circuit, without waiting for all the tokens involved in the corresponding computation to become available. While this strategy guarantees progress, once a circuit branch is created it remains open until the end of the computation. Consequently, repeated branching may lead to an exponential blow-up in the size of the generated circuit.
- **Synchronous rule.** The alternative is a synchronous transition, which waits until all the relevant tokens have reached the branching point before proceeding. This additional synchronization allows circuit branches to be merged once they reconverge, so that the continuation of the computation can be shared among different branches. The resulting circuits are therefore significantly more compact. However, this rule is not universally applicable: as explained in the previous section, higher-order control flow may induce cyclic dependencies in the program data flow, preventing the required synchronization from taking place.

Declaration on Generative AI

The authors have not employed any Generative AI tools.

References

[1] L. K. Grover, A fast quantum mechanical algorithm for database search, in: Proc. of STOC 1996, ACM, 1996, p. 212–219. doi:10.1145/237814.237866.

[2] P. W. Shor, Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer, SIAM review 41 (1999) 303–332. doi:10.1137/S0097539795293172.

[3] M. W. Johnson, M. H. S. Amin, S. Gildert, T. Lanting, F. Hamze, N. Dickson, R. Harris, A. J. Berkley, J. Johansson, P. Bunyk, E. M. Chapple, C. Enderud, J. P. Hilton, K. Karimi, E. Ladizinsky, N. Ladizinsky, T. Oh, I. Perminov, C. Rich, M. C. Thom, E. Tolkacheva, C. J. S. Truncik, S. Uchaikin, J. Wang, B. Wilson, G. Rose, Quantum annealing with manufactured spins, Nature 473 (2011) 194–198. doi:10.1038/nature10012.

[4] A. Kitaev, Fault-tolerant quantum computation by anyons, Annals of Physics 303 (2003) 2–30. doi:10.1016/S0003-4916(02)00018-0.

[5] E. Knill, Conventions for quantum pseudocode, 1996. doi:10.2172/366453.

[6] A. W. Cross, L. S. Bishop, J. A. Smolin, J. M. Gambetta, Open quantum assembly language, 2017. URL: https://arxiv.org/abs/1707.03429. arXiv:1707.03429.

[7] A. Cross, A. Javadi-Abhari, T. Alexander, N. De Beaudrap, L. S. Bishop, S. Heidel, C. A. Ryan, P. Sivarajah, J. Smolin, J. M. Gambetta, B. R. Johnson, Openqasm 3: A broader and deeper quantum assembly language, ACM Transactions on Quantum Computing 3 (2022) 1–50. doi:10.1145/3505636.

[8] Y. Feng, M. Ying, Quantum hoare logic with classical variables, ACM Transactions on Quantum Computing 2 (2021). doi:10.1145/3456877.

[9] P. Selinger, B. Valiron, Quantum lambda calculus, in: Semantic Techniques in Quantum Computation, Cambridge University Press, 2009, pp. 135–172.

[10] P. Selinger, B. Valiron, On a fully abstract model for a quantum linear functional language, Electronic Notes in Theoretical Computer Science 210 (2008) 123–137. doi:10.1016/j.entcs.2008.04.022.

[11] U. Dal Lago, A. Masini, M. Zorzi, On a measurement-free quantum lambda calculus with classical control, Math. Struct. Comput. Sci. 19 (2009) 297–335. doi:10.1017/S096012950800741X.

[12] A. Javadi-Abhari, M. Treinish, K. Krsulich, C. J. Wood, J. Lishman, J. Gacon, S. Martiel, P. D. Nation, L. S. Bishop, A. W. Cross, B. R. Johnson, J. M. Gambetta, Quantum computing with Qiskit, 2024. doi:10.48550/arXiv.2405.08810. arXiv:2405.08810.

[13] G. A. Q. Team, Cirq Programming Language, https://quantumai.google/cirq, 2018.

[14] A. S. Green, P. L. Lumsdaine, N. J. Ross, P. Selinger, B. Valiron, Quipper: a scalable quantum programming language, in: Proc. of PLDI 2013, ACM, 2013, pp. 333–342. doi:10.1145/2491956.2462177.

[15] A. S. Green, P. L. Lumsdaine, N. J. Ross, P. Selinger, B. Valiron, An introduction to quantum programming in quipper, in: Reversible Computation, Springer Berlin Heidelberg, 2013, pp. 110–124. doi:10.1007/978-3-642-38986-3_10.

[16] P. Fu, K. Kishida, N. J. Ross, P. Selinger, Proto-quipper with dynamic lifting, Proc. ACM Program. Lang. 7 (2023). doi:10.1145/3571204.

[17] P. Selinger, B. Valiron, A lambda calculus for quantum computation with classical control, in: Proc. of TLCA 2005, Springer Berlin Heidelberg, 2005, pp. 354–368. doi:10.1007/11417170_26.

[18] J.-Y. Girard, Towards a geometry of interaction, Contemporary Mathematics 92 (1989) 6.

[19] J.-Y. Girard, Geometry of interaction 1: Interpretation of system F, in: Logic Colloquium '88, volume 127 of *Studies in Logic and the Foundations of Mathematics*, Elsevier, 1989, pp. 221–260. doi:10.1016/S0049-237X(08)70271-4.

[20] I. Mackie, The geometry of interaction machine, in: Proc. of POPL 1995, ACM, 1995, pp. 198–208. doi:10.1145/199448.199483.

[21] V. Danos, L. Regnier, Reversible, irreversible and optimal lambda-machines, Theoretical Computer Science 227 (1999) 79–97. doi:https://doi.org/10.1016/S0304-3975(99)00049-3.

[22] B. Accattoli, U. Dal Lago, G. Vanoni, The machinery of interaction, in: Proc. of PPDP 2020, ACM, 2020. doi:10.1145/3414080.3414108.

[23] D. R. Ghica, Geometry of synthesis: a structured approach to VLSI design, in: Proc. of POPL 2007, ACM, 2007, pp. 363–375. doi:10.1145/1190216.1190269.

[24] U. Dal Lago, C. Faggian, B. Valiron, A. Yoshimizu, The geometry of parallelism: classical, probabilistic, and quantum effects, in: Proc. of POPL 2017, ACM, 2017, pp. 833–845. doi:10.1145/3009837.3009859.

[25] P. Selinger, Towards a quantum programming language, Mathematical. Structures in Comp. Sci. 14 (2004) 527–586. doi:10.1017/S0960129504004256.

[26] M. A. Nielsen, I. L. Chuang, Quantum Computation and Quantum Information: 10th Anniversary Edition, Cambridge University Press, 2010. doi:10.1017/cbo9780511976667.

[27] P. J. Landin, The mechanical evaluation of expressions, The Computer Journal 6 (1964) 308–320.

doi:10.1093/comjnl/6.4.308.

[28] J.-L. Krivine, A call-by-name lambda-calculus machine, Higher-Order and Symbolic Computation 20 (2007) 199–207. doi:10.1007/s10990-007-9018-9.

[29] G. Chiribella, G. M. D'Ariano, P. Perinotti, B. Valiron, Quantum computations without definite causal structure, Phys. Rev. A 88 (2013) 022318. doi:10.1103/PhysRevA.88.022318.

[30] K. Chardonnet, U. Dal Lago, N. Hoshino, P. Pistone, Compiling quantum lambda-terms into circuits via the geometry of interaction, CoRR abs/2602.17482 (2026). doi:10.48550/ARXIV.2602.17482.