

## MIT Open Access Articles

*Quantum Circuits Are Just a Phase*

The MIT Faculty has made this article openly available. **Please share** how this access benefits you. Your story matters.

**Citation:** Chris Heunen, Louis Lemonnier, Christopher McNally, and Alex Rice. 2026. Quantum Circuits Are Just a Phase. Proc. ACM Program. Lang. 10, POPL, Article 89 (January 2026), 28 pages.

**As Published:** <https://doi.org/10.1145/3776731>

**Publisher:** ACM

**Persistent URL:** <https://hdl.handle.net/1721.1/164694>

**Version:** Final published version: final published article, as it appeared in a journal, conference proceedings, or other formally published context

**Terms of use:** Creative Commons Attribution





# Quantum Circuits Are Just a Phase

CHRIS HEUNEN, University of Edinburgh, United Kingdom

LOUIS LEMONNIER, University of Edinburgh, United Kingdom

CHRISTOPHER MCNALLY, Massachusetts Institute of Technology, USA

ALEX RICE, University of Edinburgh, United Kingdom

Quantum programs today are written at a low level of abstraction—quantum circuits akin to assembly languages—and the unitary parts of even advanced quantum programming languages essentially function as circuit description languages. This state of affairs impedes scalability, clarity, and support for higher-level reasoning. More abstract and expressive quantum programming constructs are needed.

To this end, we introduce a simple syntax for generating unitaries from “just a phase”; we combine a (global) phase operation that captures phase shifts with a quantum analogue of the “if let” construct that captures subspace selection via pattern matching. This minimal language lifts the focus from gates to eigen-decomposition, conjugation, and controlled unitaries; common building blocks in quantum algorithm design.

We demonstrate several aspects of the expressive power of our language in several ways. Firstly, we establish that our representation is universal by deriving a universal quantum gate set. Secondly, we show that important quantum algorithms can be expressed naturally and concisely, including Grover’s search algorithm, Hamiltonian simulation, Quantum Fourier Transform, Quantum Signal Processing, and the Quantum Eigenvalue Transformation. Furthermore, we give clean denotational semantics grounded in categorical quantum mechanics. Finally, we implement a prototype compiler that efficiently translates terms of our language to quantum circuits, and prove that it is sound with respect to these semantics. Collectively, these contributions show that this construct offers a principled and practical step toward more abstract and structured quantum programming.

CCS Concepts: • **Theory of computation** → **Quantum computation theory**; **Formalisms**; **Categorical semantics**.

Additional Key Words and Phrases: Quantum circuits, phase, if let, independent coproduct

## ACM Reference Format:

Chris Heunen, Louis Lemonnier, Christopher McNally, and Alex Rice. 2026. Quantum Circuits Are Just a Phase. *Proc. ACM Program. Lang.* 10, POPL, Article 89 (January 2026), 28 pages. <https://doi.org/10.1145/3776731>

## 1 Introduction

Quantum computers can accommodate algorithms that solve certain classes of problems exponentially faster than the best known classical algorithms [46]. Spurred on by this promise, quantum hardware has developed to the point where it’s now a commercial reality. The current state of the art is still modest—qubit counts in the hundreds, coherence times in the microseconds, and gate error rates in the hundredths of a percent—but capabilities keep advancing at pace [16].

As quantum computing hardware keeps developing, the bottleneck to useful application is increasingly shifting to quantum software development. One reason for this lag is the low level of

---

Authors’ Contact Information: [Chris Heunen](#), University of Edinburgh, Edinburgh, United Kingdom, [chris.heunen@ed.ac.uk](mailto:chris.heunen@ed.ac.uk); [Louis Lemonnier](#), University of Edinburgh, Edinburgh, United Kingdom, [louis.lemonnier@ed.ac.uk](mailto:louis.lemonnier@ed.ac.uk); [Christopher McNally](#), Massachusetts Institute of Technology, Cambridge, USA, [mcnallyc@mit.edu](mailto:mcnallyc@mit.edu); [Alex Rice](#), University of Edinburgh, Edinburgh, United Kingdom, [alex.rice@ed.ac.uk](mailto:alex.rice@ed.ac.uk).



This work is licensed under a [Creative Commons Attribution 4.0 International License](https://creativecommons.org/licenses/by/4.0/).

© 2026 Copyright held by the owner/author(s).

ACM 2475-1421/2026/1-ART89

<https://doi.org/10.1145/3776731>

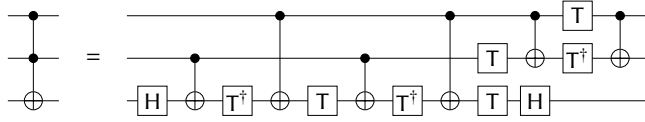


Fig. 1. The Toffoli gate expanded into a quantum circuit of native gates. The expansion obscures that the operation commutes with Z measurement.

abstraction at which quantum computers are currently programmed. Most quantum software today is written in terms of quantum circuits (see Figs. 1 and 2 for an example)—or worse, using hardware-specific execution instructions. These representations suffice for small-scale experimentation and applications, but face several challenges in the longer term:

- *Scalability.* Proven classical software engineering practice and principles show that developing and maintaining programs at larger scales needs functionality supporting modularity [53]. A related challenge is that interfacing with existing classical (high-performance) infrastructure similarly requires more structured representations.
- *Automatability.* Empirical evidence shows that the vast majority of quantum circuits implementing useful quantum algorithms is taken up by ‘bookkeeping’ [58]. The burden of having to write this boilerplate code can be shifted from the programmer to automated support systems when the representation has a high enough level of abstraction.
- *Understandability.* At the root of these challenges lies the problem that quantum circuits are too fine-grained for human programmers to understand quantum algorithms at a natural level. Intuition for quantum algorithms comes from quantum information theory, and ultimately linear algebra. Having to translate this in terms of specific quantum gate sets is merely an obscuring step. According to the empirically supported weak Sapir-Whorf hypothesis from cognitive linguistics, a language’s structure influences a speaker’s ability to perceive ideas, without strictly limiting or obstructing them [1]. Having a more abstract principled representation of quantum programs can therefore aid in the discovery of new quantum algorithms.

Additionally—but we will not address this challenge explicitly here—to enable optimising compiler passes, it is helpful to start at a higher level of abstraction, so that as much as possible of the programmer’s intent is retained [32]. At higher levels of abstraction, different optimisations become apparent. For example, the Toffoli gate is easily seen to commute with Z measurements, but this property is obscured when it is expressed in terms of hardware-native gates as in Fig. 1. To better support reasoning, optimisation, program synthesis and verification, principled quantum programming needs a representation with more expressive power and structural clarity. Working primarily with quantum circuits is *just a phase* in the coming of age of quantum programming.

Unfortunately, quantum programming requires different abstractions to classical programming; existing classical constructs do not transfer cleanly. Conditional if-then-else constructions need care within quantum computing [6, 9]. More generally, the quantum setting allows causal constructs fundamentally incompatible with classical control flow [13, 48]. Control structures such as for and while loops are limited because they cannot inspect the quantum variable controlling the loop without altering its value [4, 50]. The no-cloning theorem makes practical implementations of recursion schemes over quantum states very difficult [60, 62, 64]. Similarly, there are foundational challenges to higher-order structure [47, 52].

As a result, the abstractions available in current quantum programming languages are limited and broadly fit into two categories:

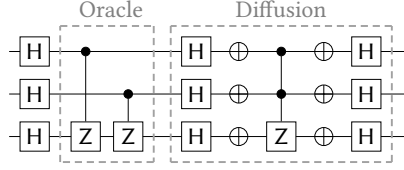


Fig. 2. A circuit for an instance of Grover’s algorithm searching for two marked bit strings 011 and 101 [22]. The intent of the programmer (and the meaning of the program) is obfuscated by the circuit representation.

- Many languages and libraries directly describe circuits [26, 58]. These languages often have various quantum gates directly as primitives, often adding orthogonal features such as classical control [17, 23, 54] or uncomputation [5, 31].
- Other languages take a much larger departure from the circuit model. Some are based on reversible computing [10, 27], including languages which utilise symmetric pattern matching [18, 39, 50] to define unitary operations. Alternative approaches [7] build on models of quantum computing such as the ZX-calculus [15]. Although these languages offer different abstractions over the circuit model, it is unclear how (or known to be hard [19]) to compile them down to circuits.

In this paper, we position ourselves between these two settings, offering an abstraction over quantum circuits while retaining a linear time compilation algorithm. We do this by introducing a new quantum programming construct, a quantum analogue of the “if let” statement, as used in Rust [37], which subsumes common operations such as conjugation and controlled blocks. When combined with an explicit treatment of global phase, often primitive quantum gates can be derived. It leverages the fact that many quantum algorithms, and in fact many linear algebra techniques, have at their heart a decomposition into eigenspaces and a manipulation of eigenvectors; the expression

if let  $p$  then  $e$

represents the former, while the latter is captured by the global phase operator

$\text{Ph}(\theta)$ .

The pattern  $p$  specifies a case split by selecting a subspace of a variable’s state space, with the “if let” expression applying its body to this subspace. Crucially, however, this subspace is not limited to align to classical values like  $|0\rangle$  or  $|1\rangle$ , but also quantum values like  $|+\rangle$  or higher-dimensional subspaces. Our syntax consists of *just a phase*, and is a simple but useful expression of the essence of eigendecompositions from linear algebra in quantum programming. We argue this in four ways.

First, this construct is expressive enough to serve as a foundational abstraction. For example, conventional gate-level operations, that are usually taken as primitive, instead emerge as derived constructs. Here is a standard computationally universal gate set in the combinator language introduced in Section 3:

$$\begin{array}{ll} Z := \text{if let } |1\rangle \text{ then } \text{Ph}(\pi) & X := \text{if let } |-\rangle \text{ then } \text{Ph}(\pi) \\ T := \text{if let } |1\rangle \text{ then } \text{Ph}(\pi/4) & Y := \text{if let } S \cdot |-\rangle \text{ then } \text{Ph}(\pi) \\ H := \text{if let } Y^{1/4} \cdot |1\rangle \text{ then } \text{Ph}(\pi) & CX := \text{if let } |1\rangle \otimes \text{id}_1 \text{ then } X \end{array}$$

As a second argument, we show that this language captures a broad class of quantum algorithms. For example, Grover’s search algorithm (whose circuit representation is given in Fig. 2) iterates two main subroutines. The most important one, the diffusion operator

$\text{Ph}(\pi) \otimes \text{id}_n$ ; if let  $|+\rangle \otimes \dots \otimes |+\rangle$  then  $\text{Ph}(\pi)$

is a one-liner. The programmer has to supply the oracle operator

$$\text{if let } |\omega_1\rangle \otimes \cdots \otimes |\omega_n\rangle \text{ then Ph}(\pi),$$

where  $\omega_j$  is the  $j^{\text{th}}$  bit in the binary expansion of the marked element, which also simplifies. In a similar way, we show that our representation can succinctly express important quantum algorithms including Quantum Fourier Transform, Hamiltonian simulation, Quantum Signal Processing, and Quantum Eigenvalue Transformation.

Third, we validate the practicality of our approach through a prototype compiler that translates our higher-level constructs into standard quantum circuits efficiently, which we present as an evaluation function converting any term to a canonical “circuit-like” form. This may also be regarded as an operational semantics for the language.

Fourth, we equip the language with a categorical denotational semantics which naturally relates to established models of quantum theory [10, 20]. More precisely, we build on rig dagger categories with independent coproducts. This semantics-first approach to language design fits in the larger programme of categorical quantum theory [28, 30]. We employ these semantics to prove soundness of our compilation algorithm.

*Implementation.* We include a prototype implementation of the combinator language [49]. This takes the form of a Rust executable and library, and can be built with cargo (tested with version 1.86.0). This implementation parses terms, performs typechecking, computes inverses and square roots, runs evaluation to a circuit, and finally outputs the matrix represented by the term. HTML documentation of this library is provided at [alexarice.github.io/phase-rs](https://alexarice.github.io/phase-rs).

*Related work.* Our work is set in the general framework of quantum control flow [56, 57], as opposed to classical control flow. In quantum control, the *branching* is decided by quantum data, leaving the result in a potential quantum superposition. There is a number of ongoing research projects around quantum control flow, and more specifically, on the integration of quantum control in programming languages paradigms.

- Symmetric pattern matching [11, 39, 50] is a proof-of-concept programming language for quantum control, in which control is only quantum. Similar to a  $\lambda$ -calculus, its only primitives are type connectives, completed with complex numbers for the quantum aspect of the language; these complex numbers allow for the expression of any unitary operator in the language. However, like the  $\lambda$ -calculus, it is an abstract language, and does not reasonably compile to quantum circuits or any quantum hardware. Symmetric pattern matching can be seen as an improvement of QML [3], also entirely based on quantum if statements, but less scalable.
- There exists a whole research program on proof languages that include quantum control [21] based on intuitionistic linear logic. While this approach is fully scalable and mathematically sound, it is geared towards logical intuition and understandability rather than quantum programming.
- Most quantum programming languages in the literature are circuit description languages, possibly with a quantum if statement [24, 62, 63], effectively acting as classical programming languages whose values are quantum circuits. In these languages, unitaries come as constants that can be called and used as black boxes. This is hard to scale or automate.
- Qunity [59] mixes symmetric pattern matching and circuit description languages to allow for some form of quantum control while keeping a syntax relatively close to quantum circuits. It, however, still contains most unitary operations as constants. The compiler has exponential blow-ups so scalability is a challenge.

- Silq [5, 31] has support for a quantum if but only for the type (qu)bit, and therefore lacks in scalability compared to what we are able to achieve. The language also does not come equipped with a compositional denotational semantics to support the validity of the operational semantics.
- The zeta calculus [7] is an abstract language—in the sense of the  $\lambda$ -calculus, which offers a compilation to the ZX-calculus, a graphical language for linear maps between finite-dimensional Hilbert spaces. However, the operations that the zeta calculus represent are not only unitary, since it allows to *copy* and to discard on the Z and X bases. There is no known way of compiling the zeta calculus (or the ZX calculus) to quantum circuits.
- Universal quantum if conditional [6]. On a more foundational aspect of quantum computing, it is known that there is no quantum operation that operates a generic “quantum if” on a black box oracle. It means in particular that quantum theory does not allow for an operator  $\lambda xy. \text{if } x \text{ then } y$ .

*Structure of this article.* After briefly reviewing the necessary background about quantum computing in Section 2, we introduce the syntax of our combinator language in Section 3, in addition to describing certain meta-operations on terms. Next, Section 4 details four examples of important (families of) quantum algorithms in our language. In Section 5, we discuss a prototype compiler from the language into quantum circuits. Denotational semantics are developed in Section 6, and are used to exhibit equalities that hold within the language, and prove our compilation algorithm is sound. We finally discuss different potential settings for our “if let” construction in Section 7, describing alternative nominal representations of the core combinator language. Section 8 concludes by discussing future developments.

*Acknowledgements.* The proofs of Theorems 20 and 23 came out of discussions with the authors of [45]. We extend our thanks to the people of the Quantum Programming group in the University of Edinburgh for their support and proofreading. This research was funded by the Engineering and Physical Sciences Research Council (EPSRC) under project EP/X025551/1 “Rubber DUQ: Flexible Dynamic Universal Quantum programming”. C.M. is supported by the U.S. Army Research Office Grant No. W911NFF-23-1-0045 (Extensible and Modular Advanced Qubits). The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the U.S. Government.

## 2 Quantum Computing

We start with the essential background on quantum computing. We will only be concerned with unitary quantum computing, and have no need to consider the measurement readout at the end of the computation in detail. For more details we refer to textbooks such as [46, 61].

*Qubits.* The unit of quantum information is the *qubit*. The state of a qubit is a unit vector  $\varphi = \begin{pmatrix} x \\ y \end{pmatrix}$  in the Hilbert space  $\mathbb{C}^2$ , that is, a pair of complex numbers  $x$  and  $y$  such that  $|x|^2 + |y|^2 = 1$ . This vector is often written in *ket notation*  $|\varphi\rangle$ . Two special vectors are the *computational basis* states  $|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$  and  $|1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$ ; a general state  $|\varphi\rangle$  is in a *superposition* of these two. Two such states that we will often use are  $|+\rangle = \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$  and  $|-\rangle = \frac{1}{\sqrt{2}}|0\rangle - \frac{1}{\sqrt{2}}|1\rangle$ .

*Entanglement.* In a system with multiple qubits, the state is a unit vector in the *tensor product*. For example, if the first qubit is in state  $|0\rangle$ , and the second qubit is in state  $|1\rangle$ , then the state of the compound system is the vector  $|0\rangle \otimes |1\rangle \in \mathbb{C}^2 \otimes \mathbb{C}^2$ , also written as  $|01\rangle$ . Not all states of a system with multiple qubits are of this form. For example, the state  $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$  is *entangled*: it cannot

be written in the form  $|\varphi\rangle \otimes |\psi\rangle$ . In general, the states of a system with  $n$  qubits can range over the unit vectors in  $\mathbb{C}^2 \otimes \dots \otimes \mathbb{C}^2 \simeq \mathbb{C}^{(2^n)}$ .

*Unitaries.* Qubits can undergo operations specified by unitary matrices. These are  $2^n$ -by- $2^n$  matrices  $U$  with complex entries satisfying  $U^\dagger U = 1$  (and hence also  $UU^\dagger = 1$ ), that is, the matrix is invertible and its inverse is its conjugate transpose. On single qubits, standard operations include:

$$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \quad X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \quad Y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix} \quad Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \quad T = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{bmatrix}$$

The matrix  $H$  is called the *Hadamard* transformation, and satisfies  $H|0\rangle = |+\rangle$  and  $H|1\rangle = |-\rangle$ .

*Control.* Another way to combine two  $n$ -qubit systems is by direct sum as in the left-hand side of

$$\mathbb{C}^{(2^n)} \oplus \mathbb{C}^{(2^n)} \simeq \mathbb{C}^{(2^{n+1})} \simeq \mathbb{C}^2 \otimes \mathbb{C}^{(2^n)}.$$

Notice how this ‘sum type’ can also be described as a ‘product type’ with a new qubit as in the right-hand side. This new qubit is called the *control* qubit, because it controls which operation is applied to the other, *target*, qubits, as follows. Any  $n$ -qubit unitary  $U$  can be extended to an  $(n+1)$ -qubit unitary  $CU$ , defined as  $I \oplus U$  or given by the block diagonal matrix

$$CU = \begin{bmatrix} I & 0 \\ 0 & U \end{bmatrix}$$

where  $I$  is the identity on  $\mathbb{C}^{(2^n)}$ . This controlled- $U$  will apply  $U$  only if the control qubit was in the state  $|1\rangle$ ; otherwise it will do nothing. For example, the controlled- $X$  gate  $CX$  is given by

$$CX = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

Since the control qubit can be in superposition, both branches are executed in superposition, which is a key difference from classical control flow, where only one branch is executed. Observe especially that this quantum control is predicated on the *computational basis*  $\{|0\rangle, |1\rangle\}$  of the control qubit  $\mathbb{C}^2$ . Many important quantum algorithms need to perform controlled operations on different subspaces than those aligned along the linear spans of  $|0\rangle$  and  $|1\rangle$ .

*Phases.* Two unitary matrices  $U$  and  $V$  are indistinguishable in their measurable effect on qubits when they are equal up to a *global phase*:  $U = e^{i\theta}V$  for some  $\theta \in [0, 2\pi)$ . In other words, we consider the identity matrix to model the same computation multiplying with the scalar  $e^{i\theta}$ . Nevertheless, *local phases*

$$P(\theta) = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\theta} \end{bmatrix}$$

are *not* identified with the identity, and in fact form the heart of many quantum algorithms, despite the fact that they can be regarded as controlled 0-qubit global phase operations. For example, notice that  $Z = P(\pi)$  and  $T = P(\pi/4)$ .

*Circuits.* Any quantum computation is described by a unitary matrix. These are typically built up from one- and two-qubit unitaries, also called *quantum gates*, that are combined using tensor products and matrix multiplication. It is customary to draw such a composite unitary as a *quantum circuit*: a graphical depiction where horizontal wires represent qubits acted upon by quantum gates while they flow from left to right. A controlled  $U$  gate is depicted as on the left below, and in

particular the controlled X gate is depicted as on the right below.



Gates can similarly be controlled on multiple qubits, such as the CCX matrix, or *Toffoli gate*, in Fig. 1. See Fig. 2 for another example of a quantum circuit.

*Eigendecomposition.* For a linear map  $U : \mathcal{H} \rightarrow \mathcal{H}$ , its *eigenvalues* are scalars  $\lambda$  such that the subspace  $\{v : Uv = \lambda v\}$  is non-zero. Such subspaces are referred to as *eigenspaces* and the vectors they contain are called *eigenvectors* of the map  $U$ . When  $U$  is unitary, there exists a basis  $\{|\lambda_i\rangle\}$  of eigenvectors for  $\mathcal{H}$ , each with eigenvalue  $\lambda_i$ , allowing us to obtain the *eigendecomposition*:

$$U = \sum_i \lambda_i |\lambda_i\rangle\langle\lambda_i|$$

The maps  $|\lambda_i\rangle\langle\lambda_i|$  are *projections*, maps  $p$  such that  $p^2 = p$ .

Further,  $U$  admits a *diagonalisation*  $U = Q\Lambda Q^\dagger$ , where  $\Lambda$  is the diagonal matrix with entries  $\lambda_i$ . In this sense, every unitary map can be realised as a sequence of phase rotations applied to its eigenspaces, motivating our representation of them utilising *just a phase*.

### 3 Syntax

We are now positioned to introduce the core language of this paper, a combinator-based language for describing unitary linear transformations, with two basic building blocks: a global phase unitary, and a quantum “if let” allowing a restricted form of pattern matching.

In this combinator-style language, terms represent unitary maps on a set number of qubits. The types for unitaries are therefore very simple, and are in direct correspondence with natural numbers: a term  $t$  of type  $qn \leftrightarrow qn$  will represent a unitary map  $\mathbb{C}^{2^n} \rightarrow \mathbb{C}^{2^n}$ , and this is the only possible type a unitary can have. This language has no variables, and hence its typing derivations do not require a context and are simply written:

$$\vdash t : qn \leftrightarrow qn$$

for a term  $t$  and  $n \in \mathbb{N}$ .

We introduce a (global) phase operation as the only primitive “gate”. It takes the form of a 0-qubit unitary (and should not be confused with the 1-qubit phase gate commonly referred to as S). For each angle<sup>1</sup>  $\theta \in \mathbb{R}$  we write:

$$\frac{}{\vdash \text{Ph}(\theta) : q0 \leftrightarrow q0}$$

To create larger programs, we must be able to compose unitaries together. This can be done sequentially or in parallel. We further require an explicit identity term. These have the following typing rules:

$$\frac{\vdash s : qn \leftrightarrow qn \quad \vdash t : qn \leftrightarrow qn}{\vdash s; t : qn \leftrightarrow qn} \quad \frac{\vdash s : qn \leftrightarrow qn \quad \vdash t : qm \leftrightarrow qm}{\vdash s \otimes t : q(n+m) \leftrightarrow q(n+m)} \quad \frac{}{\vdash \text{id}_n : qn \leftrightarrow qn}$$

At this point, this syntax can only represent unitaries that perform a global phase, which, as already noted in Section 2, have no computational effect in a quantum circuit. The ability to perform arbitrary quantum gates will be derived from our quantum “if let” construction. This construction allows a unitary to be performed on a subspace specified by a *pattern*. Patterns  $p$  are given types of

<sup>1</sup>In practice we must fix a (countable) group of angles in order for this syntax to be finitary.

the form  $qn < qm$  and correspond to isometries  $i: \mathbb{C}^{2^n} \rightarrow \mathbb{C}^{2^m}$ . The “if let” expression then has the following typing rule:

$$\frac{\vdash p : qn < qm \quad \vdash s : qn \leftrightarrow qn}{\vdash \text{if let } p \text{ then } s : qm \leftrightarrow qm}$$

One intuition for the action of the “if let” expression is the following: if  $s$  represents the unitary  $U$ , and  $p$  represents the isometry  $i$ , then the unitary represented by “if let  $p$  then  $s$ ” performs  $U$  on the range subspace of  $i$ , and the identity on its orthogonal complement. The “if let” construction can be viewed as a restricted form of symmetric pattern matching [50].

Patterns are given by a separate but related syntax to terms, for which the rules are given below:

$$\begin{array}{c} \frac{}{\vdash |0\rangle : q0 < q1} \quad \frac{}{\vdash |1\rangle : q0 < q1} \quad \frac{}{\vdash |+\rangle : q0 < q1} \quad \frac{}{\vdash |-\rangle : q0 < q1} \quad \frac{\vdash s : qn \leftrightarrow qn}{\vdash s : qn < qn} \\ \\ \frac{\vdash p : qn < qm \quad \vdash q : ql < qn}{\vdash p \cdot q : ql < qm} \quad \frac{\vdash p : qn < qm \quad \vdash q : qn' < qm'}{\vdash p \otimes q : q(n+n') < q(m+m')} \end{array}$$

The latter three rules observe that all unitary maps are also isometries, and that isometries are closed under composition and tensor products. Each pattern  $|x\rangle$  represents the isometry  $z \mapsto z|x\rangle : \mathbb{C} \rightarrow \mathbb{C}^2$ . We note that the composition for patterns is in function composition order, in contrast to the diagrammatic composition order for terms.

We highlight three important cases of this construction:

- If  $p : q0 < qn$ , then the term  $\text{if let } p \text{ then Ph}(\theta)$  represents the unitary which maps  $p(\alpha) + v$  to  $e^{i\theta}p(\alpha) + v$  (where  $\langle p(1), v \rangle = 0$ ), which has eigenvalues 1 and  $e^{i\theta}$ .
- Let  $s : qn \leftrightarrow qn$  represent the unitary  $U$  and consider the term:

$$\text{if let } |1\rangle \otimes \text{id}_n \text{ then } s$$

The unitary represented by this term sends any input of the form  $|1\rangle \otimes v$  to  $|1\rangle \otimes U(v)$ , and leaves any input of the form  $|0\rangle \otimes v$  unchanged. This term therefore represents the controlled  $U$  operation.

- Suppose  $s, t : qn \leftrightarrow qn$ , representing unitaries  $U$  and  $V$ . Then the unitary represented by

$$\text{if let } s \text{ then } t$$

is the unitary  $U \circ V \circ U^\dagger$ , which sends  $U(v)$  to  $U(V(v))$ .

**Example 1** (X gate). Our first example is the term:

$$\text{if let } |-\rangle \text{ then Ph}(\pi)$$

By the intuition above, this represents a unitary which maps  $|-\rangle$  to  $e^{i\pi}|-\rangle = -|-\rangle$ , and  $|+\rangle$  (which is orthogonal to  $|-\rangle$ ) to  $|+\rangle$ . Its action on other vectors is determined by linearity;  $|0\rangle = \frac{1}{\sqrt{2}}(|+\rangle + |-\rangle)$  is sent to  $\frac{1}{\sqrt{2}}(|+\rangle - |-\rangle) = |1\rangle$  and similarly  $|1\rangle$  is sent to  $|0\rangle$ , making this the quantum X gate.

The syntax presented here allows the simple definition of two important meta-level operations: inversion and exponentiation. The ability to obtain the inverse of a quantum program is not uncommon, yet we highlight the simplicity of the definition below.

**Definition 2** (Inversion). Given a term  $\vdash t : qn \leftrightarrow qn$ , we define its *inverse*  $\vdash t^\dagger : qn \leftrightarrow qn$  by structural induction on the syntax:

$$\text{Ph}(\theta)^\dagger = \text{Ph}(-\theta) \quad (\text{if let } p \text{ then } s)^\dagger = \text{if let } p \text{ then } s^\dagger \quad (s \otimes t)^\dagger = s^\dagger \otimes t^\dagger \quad (s; t)^\dagger = t^\dagger; s^\dagger$$

A simple induction shows the resulting term is well-typed.

$$\begin{array}{ll}
Z := \text{if let } |1\rangle \text{ then Ph}(\pi) & S := \sqrt{Z} = \text{if let } |1\rangle \text{ then Ph}(\pi/2) \\
X := \text{if let } |-\rangle \text{ then Ph}(\pi) & V := \sqrt{X} = \text{if let } |-\rangle \text{ then Ph}(\pi/2) \\
Y := \text{if let } |-\rangle \text{ then Ph}(\pi) & T := \sqrt{S} = \text{if let } |1\rangle \text{ then Ph}(\pi/4) \\
CZ := \text{if let } |1\rangle \otimes |1\rangle \text{ then Ph}(\pi) & CX := \text{if let } |1\rangle \otimes |-\rangle \text{ then Ph}(\pi) \\
H := \text{if let } Y^{1/4} \cdot |1\rangle \text{ then Ph}(\pi) & \\
= \text{if let}(\text{if let } S \cdot |-\rangle \text{ then Ph}(\pi/4)) \cdot |1\rangle \text{ then Ph}(\pi) & 
\end{array}$$

Fig. 3. Definitions of common quantum gates in the combinator syntax. A fully universal gate set can be generated with *just a phase*.

For terms which do not contain the composition constructor (in particular terms which consist of a single “if let” statement), we can perform the much more general operation of exponentiation. The ability to define exponentiation exemplifies the utility of our syntax.

**Definition 3** (Exponentials). Let  $\vdash t : qn \leftrightarrow qn$  be a “composition-free” term, i.e. a term containing no instances of “;”. For a real number  $\alpha$ , define the *exponentiation*  $\vdash t^\alpha : qn \leftrightarrow qn$  by structural induction:

$$\text{Ph}(\theta)^\alpha = \text{Ph}(\alpha\theta) \quad (\text{if let } p \text{ then } s)^\alpha = \text{if let } p \text{ then } s^\alpha \quad (s \otimes t)^\alpha = s^\alpha \otimes t^\alpha$$

Similar to above, a simple induction shows exponentiation is well-typed. We note that the case where  $\alpha = -1$  coincides with the inversion operation. We write  $\sqrt{t}$  for  $t^{1/2}$ .

We highlight the use of the exponentiation operation by applying it to the X gate.

**Example 4.** The quantum  $V = \sqrt{X}$  gate, which satisfies  $V \circ V = X$  is given by the matrix:

$$V = \frac{1}{2} \begin{bmatrix} 1+i & 1-i \\ 1-i & 1+i \end{bmatrix}$$

Deriving this matrix from only the definition of X (or indeed obtaining a quantum circuit for this gate) is non-trivial, yet its definition in our language is immediate from the definition of the X and exponentiation:

$$V = X^{0.5} = \text{if let } |-\rangle \text{ then Ph}(\pi/2)$$

This term represents the unitary given by the matrix above.

We are now able to recover the definitions of many common quantum gates, which are given in Fig. 3. We emphasise that these gates are approximately universal, and hence all unitaries (of dimension  $2^n$ ) can be represented using this language.

**Example 5.** The 5-qubit GHZ state ( $= 1/\sqrt{2}(|00000\rangle + |11111\rangle)$ ) can be prepared from the zero state as follows, where H and X are defined in Fig. 3:

$$H \otimes \text{id}_4; \text{if let } |1\rangle \otimes \text{id}_4 \text{ then } X \otimes X \otimes X \otimes X$$

This highlights one of the flaws of the combinator style syntax, to apply a Hadamard gate to the first qubit, we must explicitly tensor it with the remaining qubits. Further, the pattern  $|1\rangle \otimes \text{id}_4$  causes the body to be controlled by the first qubit, but also must explicitly be tensored with the remaining qubits.

The combinator syntax also enforces a total ordering on the qubits, which may or may not be desirable. If we had decided to create the GHZ state with successive CX gates, we would find that there is no trivial way to apply such a 2-qubit gate to the first and third qubits.

**Remark 6.** Instead of introducing  $|+\rangle$  and  $|-\rangle$  as primitive patterns, we could have instead introduced the Hadamard gate  $H$  as a primitive unitary, defining  $|+\rangle = H \cdot |0\rangle$  and  $|-\rangle = H \cdot |1\rangle$ . Presenting the language in this way may be beneficial for contexts where the Hadamard gate is an important or easy operation, as the definition of the Hadamard gate in Fig. 3 is more involved. The set up taken above, however, allows an arguably more minimal presentation by having the phase rotation be the only primitive unitary, and enables the exponentiation operation; it is unclear what the square root of a primitive Hadamard operation should be, yet its definition is immediate when presented as a single “if let” statement.

We end this section with one further illustrative example.

**Example 7.** Let  $XC = \text{if let } |-\rangle \otimes |1\rangle \text{ then Ph}(\pi)$ , a controlled not operation where the second qubit is the control qubit and the first is the target. We can then define concisely the swap gate as:

$$\text{Swap} := \text{if let CX then XC}$$

The unitary pattern  $CX$  acts on the body of the “if let” by conjugation, allowing us to recover the more usual definition  $CX; XC; CX$  of the swap. By a simple manipulation, we can also recover another definition of the swap gate:

$$\text{if let CX} \cdot (|-\rangle \otimes |1\rangle) \text{ then Ph}(\pi)$$

This presents an alternative way of understanding the action of this gate. By observing that  $CX|-\rangle = \frac{1}{\sqrt{2}}(|01\rangle - |10\rangle)$ , we notice that the term above multiplies this component of the input by  $-1$ , permuting the  $|01\rangle$  and  $|10\rangle$  components.

## 4 Algorithms

Many well-known quantum algorithms can be expressed naturally in terms of conditional phases. Below, we give implementations of several of the most prominent algorithms and prove their correctness. We begin with Grover’s search algorithm, whose *oracle* and *diffusion* operators are precisely conditional phases on the target subspace and the uniform subspace, respectively. Next, quantum simulation algorithms are often based on finite-difference time-domain Hamiltonian evolution. In these algorithms, we take a number of time-steps, in each of which the state advances by applying conditional phases corresponding to the eigenspaces of the Hamiltonian. After that, we show how the *quantum Fourier transform* (QFT) is naturally expressed as a sequence of conditional phases. Finally, we present implementations of *quantum signal processing* and the *quantum eigenvalue transform*, which implement functions of black-box unitaries by applying conditional phases.

### 4.1 Grover’s Algorithm

The celebrated quantum database search algorithm of Grover [25] has a simple formulation in this language. Assume we are searching a database  $X = \{0, 1, \dots, N-1\}$  of size  $N = 2^n$  for elements  $x$  on which a function  $f : X \rightarrow \{0, 1\}$  takes the value 1. We may assume that  $f(x) = 1$  at only a single element  $0 \leq \omega < N$ , as multiple marked elements can be handled by a sequence of such oracles. Recall that the algorithm consists of three steps [46]:

- (1) Preparation of a uniform superposition  $|s\rangle = \frac{1}{\sqrt{|X|}} \sum_{x \in X} |x\rangle$
- (2) Repeat  $\lceil \pi\sqrt{N}/4 \rceil$  times:
  - (a) Apply the *oracle operator*  $U_f = \sum_{x \in X} (-1)^{f(x)} |x\rangle\langle x| = 1 - 2|\omega\rangle\langle\omega|$ .
  - (b) Apply the *diffusion operator*  $U_s = 2|s\rangle\langle s| - I$ .
- (3) Measure the quantum state.

With high probability, the measurement result is  $|\omega\rangle$ .

The oracle operator  $U_f$  can be implemented by the following program:

if let  $|\omega_0\rangle \otimes \cdots \otimes |\omega_{n-1}\rangle$  then  $\text{Ph}(\pi)$

where  $\omega_j$  is the  $j^{\text{th}}$  bit in the binary expansion of  $\omega$ . Similarly, the diffusion operator  $U_s$  is given by the program  $\text{Ph}(\pi) \otimes \text{id}_n$ ; if let  $|+\rangle \otimes \cdots \otimes |+\rangle$  then  $\text{Ph}(\pi)$ . Compare these pieces of syntax with the circuits in Figure 2. Grover's algorithm is then simply an interleaving of these operators.

## 4.2 Quantum Simulation

We will now implement the Trotter simulation algorithm [40] by applying a sequence of conditional phases. This is perhaps not how computer scientists naturally think about quantum simulation, but physicist practitioners naturally think in terms of dynamics as composing programs 'spectrally' and applying differential phases. This is completely independent from the mechanism of the if-let construct, which we focus on. Once the Hamiltonian is decomposed into projections, we construct a program that realises conditional phases on the subspace picked out by each projection.

Let  $H \in \mathcal{B}(\mathcal{H})$  be a positive self-adjoint operator on a  $2^n$ -dimensional Hilbert space. Then there is a (possibly empty) decomposition  $H = \sum_{i=1}^K \lambda_i \Pi_i$ , where each  $\Pi_i$  is a projection. Write  $\tilde{H} = ((\lambda_i, \Pi_i))_{i=1}^K$  for the  $K$ -tuple of spectral components consisting of ordered pairs of eigenvalues and projectors. Now, the decomposition above is not unique, so neither is  $\tilde{H}$ . In fact, it need not be a *spectral* decomposition in the usual sense. We have two choices: impose uniqueness by requiring that the  $\Pi_i$  be orthonormal, that the range of  $\Pi_i$  coincides with the kernel of  $H - \lambda_i I$ , that  $\lambda_1 \leq \cdots \leq \lambda_K$ ; or, take  $\tilde{H}$  to be primary and  $H$  to be derived. Let us adopt the latter approach.

Now suppose we are given a set of patterns  $\{p_i\}_{1 \leq i \leq K}$ . By possibly padding out the range of the projection  $\Pi_i$  to power of two dimension, we may assume without loss of generality that  $\vdash p_i : qm_i < qn$ , that is  $\Pi_i = \iota_i \iota_i^\dagger$  where  $\iota$  is the isometry represented by  $p_i$ .

Then we define a program  $U_{\tilde{H}}(t)$  inductively, by

$$U_0(t) := \text{id}_n$$

$$U_{((\lambda_i, \Pi_i))_{i=1}^{k+1}}(t) := \text{if let } p_{k+1} \text{ then } \text{Ph}(-\lambda_{i+1}t) \otimes \text{id}_{m_i}; U_{((\lambda_i, \Pi_i))_{i=1}^k}(t).$$

If the  $\Pi_i$  are mutually orthogonal, then  $U_{\tilde{H}}(t)$  represents the unitary  $e^{-iHt}$ . This does not hold in general, for non-commuting projectors, but  $U_{\tilde{H}}(t/N)^N$  represents a unitary which converges to  $e^{-iHt}$  as  $N \rightarrow \infty$  by the well-known *Trotterisation* formula [55].

Let us now consider a simple concrete example of interacting spin-1/2 magnetic dipoles in an external magnetic field  $\mathbf{B}$  [51]. The Hamiltonian is

$$H = H_{\text{free}} + H_{\text{int}} \quad H_{\text{free}} = -\frac{\hbar}{2} \sum_{j=1}^2 \gamma_j \sigma_j \cdot \mathbf{B} \quad H_{\text{int}} = \frac{\mu_0 \gamma_1 \gamma_2 \hbar^2}{16\pi r^3} (\sigma_1 \cdot \sigma_2 - 3(\hat{\mathbf{r}} \cdot \sigma_1)(\hat{\mathbf{r}} \cdot \sigma_2))$$

where  $\sigma_j$  is the vector of Pauli operators on the  $j^{\text{th}}$  spin,  $\gamma_j$  is the gyromagnetic moment of the  $j^{\text{th}}$  spin,  $\mathbf{r}$  is the displacement between the two spins,  $r = \|\mathbf{r}\|$ , and  $\hat{\mathbf{r}} = \mathbf{r}/r$ . We can simplify this expression to  $H = \omega_1 \sigma^z \otimes I + \omega_2 I \otimes \sigma^z + J(\sigma^x \otimes \sigma^x + \sigma^y \otimes \sigma^y - 2\sigma^z \otimes \sigma^z)$ . Not having imposed uniqueness or even orthogonality on the decomposition  $\tilde{H}$ , we can work term-by-term. Writing

$$\Pi_{+z} = |0\rangle\langle 0| \quad \Pi_{-z} = |1\rangle\langle 1| \quad \Pi_{+x} = |+\rangle\langle +| \quad \Pi_{-x} = |-\rangle\langle -| \quad \Pi_{+y} = |i\rangle\langle i| \quad \Pi_{-y} = |-i\rangle\langle -i|,$$

we obtain

$$\begin{aligned} H = & \omega_1(\Pi_{+z} - \Pi_{-z}) \otimes (\Pi_{+z} + \Pi_{-z}) + \omega_2(\Pi_{+z} + \Pi_{-z}) \otimes (\Pi_{+z} - \Pi_{-z}) \\ & + J(\Pi_{+x} - \Pi_{-x}) \otimes (\Pi_{+x} - \Pi_{-x}) + J(\Pi_{+y} - \Pi_{-y}) \otimes (\Pi_{+y} - \Pi_{-y}) \\ & - 2J(\Pi_{+z} - \Pi_{-z}) \otimes (\Pi_{+z} - \Pi_{-z}). \end{aligned}$$

Finally, we can distribute the tensor products over the sums, obtaining the decomposition  $\tilde{H}$  into projectors. It remains to find the corresponding *patterns*. Recall that for each  $\Pi_i$  we must find a corresponding pattern  $p_i$ . The following patterns  $p_{\pm x}, p_{\pm y}, p_{\pm z}$  suffice:

$$p_{+z} = |0\rangle \quad p_{-z} = |1\rangle \quad p_{+x} = |+\rangle \quad p_{-x} = |-\rangle \quad p_{+y} = S \cdot |+\rangle \quad p_{-y} = S \cdot |-\rangle$$

These patterns allow us to compute a term representing  $e^{-iHt}$ , as required.

### 4.3 Quantum Fourier Transform

The quantum Fourier transform over  $\mathbb{Z}/2^n\mathbb{Z}$  is the  $n$ -qubit unitary operator  $F_{2^n}$  such that  $\langle y | F_{2^n} | x \rangle = 2^{-n/2} \omega^{xy}$ , for  $0 \leq x, y < 2^n$ , where  $\omega$  is a primitive  $2^n$ th root of unity [46]. If  $x_1 x_2 \dots x_n$  is the binary expansion of  $0 \leq x < 2^n$ , then

$$F_{2^n} |x_1\rangle \otimes \dots \otimes |x_n\rangle = 2^{-n/2} \left( \bigotimes_{j=1}^n \left( |0\rangle + e^{2\pi i x_j 2^{j-1-n}} |1\rangle \right) \right).$$

The textbook algorithm for implementing this operation consists of a sequence of controlled phases. Define the *dyadic rational phase gate*  $R_n :=$  if let  $|1\rangle$  then  $\text{Ph}(2\pi/2^n)$ . To define  $\vdash \text{QFT}_n : qn \leftrightarrow qn$ , the term representing the  $n$  qubit quantum Fourier transform, we can leverage that chains of controlled gates with the same control qubit can be replaced by a single control block. Using this we obtain the following recursive definition:

$$\begin{aligned} \text{QFT}_0 & := \text{id}_0 \\ \text{QFT}_{n+1} & := H \otimes \text{id}_n; \text{ if let } |1\rangle \otimes \text{id}_n \text{ then } R_2 \otimes \dots \otimes R_{n+1}; \text{id}_1 \otimes \text{QFT}_n \end{aligned}$$

In order to recover the original quantum Fourier transform  $F_{2^n}$ , the order of the output qubits must be inverted, which could be done with the addition of the appropriate Swap gates.

### 4.4 Quantum Signal Processing

*Quantum signal processing* (QSP) is a procedure that transforms a parametric single-qubit rotation to modify its sensitivity to the parameter [41, 43]. Following [44], we write

$$W(a) = e^{i\theta X} = \begin{bmatrix} a & i\sqrt{1-a^2} \\ i\sqrt{1-a^2} & a \end{bmatrix}$$

for the parametric unitary to which the user has black-box access (a rotation about the  $X$ -axis by  $\theta = -2 \cos^{-1} a$ ). QSP makes a number of calls to  $W(a)$ , as well as a number of phases, to implement a modified unitary,

$$\tilde{W}(a) = \begin{bmatrix} P(a) & iQ(a)\sqrt{1-a^2} \\ iQ^*(a)\sqrt{1-a^2} & P^*(a) \end{bmatrix}$$

This is useful in, among other applications, quantum control of large ensembles of quantum systems subject to inhomogeneous coherent control [8]. We regard  $a$  as the amplitude of a control field, which due to spatial gradients couples more strongly to some elements of an ensemble than to others. When the amplitude of the control field varies over an interval  $I = [a_0, a_1]$  within a sample,

we may choose  $P, Q$  such that  $\tilde{W}(I)$  is approximately constant. Dually, in sensing applications, we may wish to *enhance* sensitivity to the parameter  $a$ .

There turns out to be a construction of a composite control sequence that implements any  $\tilde{W}$  for any polynomials  $P$  (resp.  $Q$ ) of degree  $d$  (resp.  $d - 1$ ) and parity  $d \bmod 2$  (resp.  $(d - 1) \bmod 2$ ), subject to the constraints imposed by unitarity of  $\tilde{W}$ , using  $d$  calls to  $W$ . All such  $\tilde{W}$  can be realised as a product,

$$\tilde{W}(a) = W_{\vec{\phi}}(a) := S(\phi_0)W(a)S(\phi_1) \cdots S(\phi_{d-1})W(a)S(\phi_d),$$

for some tuple  $\vec{\phi} \in [0, 2\pi)^{d+1}$ , where  $S(\phi) = e^{i\phi_0 Z}$ .

As a single-qubit protocol, the translation to a program is easy. Define

$$\begin{aligned} R_z(\alpha) &:= \text{Ph}(-\alpha/2) \otimes \text{id}_1; \text{ if let } |0\rangle \text{ then Ph}(\alpha) \\ R_x(\alpha) &:= \text{Ph}(-\alpha/2) \otimes \text{id}_1; \text{ if let } |+\rangle \text{ then Ph}(\alpha). \end{aligned}$$

We then define a *quantum signal processing* program  $\text{QSP}(a; \vec{\phi})$  inductively by

$$\begin{aligned} \text{QSP}(a; (\phi_0)) &:= R_z(2\phi_0) \\ \text{QSP}(a; (\phi_0, \dots, \phi_k, \phi_{k+1})) &:= \text{QSP}(a; (\phi_0, \dots, \phi_k)); \\ &\quad R_x(-2 \cos^{-1}(a)); R_z(2\phi_{k+1}) \end{aligned}$$

(recalling that  $\vec{\phi}$  is by hypothesis of length  $\geq 1$ ).

#### 4.5 Quantum Eigenvalue Transform

QSP generalises rather dramatically to the *quantum eigenvalue transform* (QET) [42, 44]. Like QSP, the QET applies a polynomial transform to an operator, but in this instance an operator on a larger finite-dimensional quantum system, not merely a qubit. Still following the presentation and conventions of [44], we proceed to an implementation of the QET.

We are given a Hamiltonian  $H$  acting on a finite-dimensional Hilbert space  $\mathcal{H} = \mathbb{C}^{2^m}$ , and unitary  $U$  acting on  $\mathbb{C}^{2^n}$  such that  $U$  contains a copy of  $H$  in a block determined by a projector  $\Pi$ , which is also given to us. Then the QET produces a unitary  $\tilde{U}$ , such that (using the notation of [44]):

$$U = \begin{matrix} \Pi \\ \Pi \left[ \begin{array}{c} \mathcal{H} \cdot \\ \cdot \cdot \end{array} \right] \end{matrix} \quad \tilde{U} = \begin{matrix} \Pi \\ \Pi \left[ \begin{array}{c} P(\mathcal{H}) \cdot \\ \cdot \cdot \end{array} \right] \end{matrix}$$

where  $P(x)$  is some degree  $d$  polynomial function.

As an example, in the case where  $\Pi = |0\rangle\langle 0|$ , we may take:

$$U = Z \otimes H + X \otimes \sqrt{I - H^2} = \sum_{\lambda \in \sigma(H)} R(\lambda) \otimes |\lambda\rangle\langle \lambda|$$

where  $R(\lambda)$  is the operator  $\lambda Z + \sqrt{1 - \lambda^2} X$  qubit operator and  $\lambda$  ranges over the eigenvalues of  $H$  such that  $H = \sum_{\lambda} \lambda |\lambda\rangle\langle \lambda|$ .

The crux of QET is that there is a tuple  $\vec{\phi} = (\phi_1, \dots, \phi_d)$  such that

$$\tilde{U} := \begin{cases} \prod_{k=1}^{d/2} \Pi_{\phi_{2k-1}} U_{\phi_{2k-1}}^\dagger \Pi_{\phi_{2k}} U & d \text{ even} \\ \Pi_{\phi_1} U \prod_{k=1}^{(d-1)/2} \Pi_{\phi_{2k}} U_{\phi_{2k}}^\dagger \Pi_{\phi_{2k+1}} U & d \text{ odd} \end{cases}$$

where  $\Pi_\phi$  is the ‘‘projector controlled phase shift’’ and is defined to be:

$$\Pi_\phi = e^{i\phi(2\Pi - I)}$$

If we assume that the projector  $\Pi$  is provided to us via a pattern  $\vdash p_\Pi : qm < qn$  (such that  $p_\Pi$  represents an isometry  $\iota$  such that  $\Pi = \iota^\dagger$ ), we can implement this projector controlled phase shift as a single “if let”:

$$R_{p_\Pi}(\phi) := \text{Ph}(-\phi) \otimes \text{id}_n; \text{if let } p_\Pi \text{ then } \text{Ph}(2\phi) \otimes \text{id}_m$$

If we further assume we are given a term  $\vdash s_U : qn \leftrightarrow qn$  which represents the unitary  $U$ , then we define  $\text{QET}(s_U, p_\Pi, \vec{\phi})$  inductively by:

$$\begin{aligned} \text{QET}(s_U, p_\Pi; ()) &:= \text{id}_n \\ \text{QET}(s_U, p_\Pi; (\phi_1)) &:= s_U; R_{p_\Pi}(\phi_1) \\ \text{QET}(s_U, p_\Pi; (\phi_1, \dots, \phi_{k-1}, \phi_k)) &:= \text{QET}(s_U; (\phi_0, \dots, \phi_{k-2})); \\ &\quad s_U; R_{p_\Pi}(\phi_k); s_U^\dagger; R_{p_\Pi}(\phi_{k-1}) \end{aligned}$$

Note that there are two base cases: one each for  $d$  even and  $d$  odd.

## 5 Compilation

In this section we will describe a conversion from the combinator language in Section 3 to a more traditional quantum circuit representation of unitary programs. Our strategy for the compilation will be to provide a normalisation algorithm, putting terms in a canonical “circuit-like” form. From this form, a circuit representation can be trivially extracted. The circuit representation we compile to here will consist of a fixed number of qubits (which can be referred to by index), a Hadamard gate, and arbitrary multi-controlled phase gates (both zero- and one- controlled with arbitrary angle). The problem of further compiling multi-qubit gates to a finite gate set is well studied [33, 36] and outside the scope of this work.

We begin by defining this canonical form.

**Definition 8** (Normal term). Let a *simple* pattern be one in the form:

$$q_1 \otimes \dots \otimes q_n$$

where each  $q_i$  is either  $\text{id}_1$ ,  $|0\rangle$ ,  $|1\rangle$ ,  $|+\rangle$ , or  $|-\rangle$ . Define a *normal clause* to be a term of the form:

$$\text{if let } q \text{ then } \text{Ph}(\theta) \otimes \text{id}_k$$

where  $q$  is a simple pattern, and let a *normal term* be a sequential composition of such clauses.

Normal terms therefore take the following form:

$$\text{if let } q_{11} \otimes \dots \otimes q_{1n} \text{ then } \text{Ph}(\theta_1) \otimes \text{id}_{j_1}; \dots; \text{if let } q_{k1} \otimes \dots \otimes q_{kn} \text{ then } \text{Ph}(\theta_k) \otimes \text{id}_{j_k}$$

where each  $q_{ij}$  is either  $\text{id}_1$ ,  $|0\rangle$ ,  $|1\rangle$ ,  $|+\rangle$ , or  $|-\rangle$ . A circuit can then be directly extracted from such a form, with each “if let” statement being replaced by a multi-controlled phase, conjugated by Hadamard gates on qubits that are plus-controlled or minus-controlled.

To massage terms into such a form, the following cases must be tackled:

- Tensor products of gates must be reduced to sequential compositions of “whiskered” gates, where whiskering refers to taking a tensor product with the identity gate. As an example  $Z \otimes Z$  could be reduced to  $Z \otimes \text{id}_1; \text{id}_1 \otimes Z$ . The choice to avoid tensor products in our final representation is motivated by tensor products not being stable under control—the control of  $Z \otimes Z$  is not the tensor product of two CZ gates. In contrast, whiskering and sequential composition *are* stable under control.
- “If let” statements of sequential compositions should be reduced to compositions of “if let” statements.

- Patterns which are unitaries should be reduced to conjugation, for example the swap gate  
if let (if let  $| -1 \rangle$  then  $\text{Ph}(\pi)$ ) then if let  $| -1 \rangle$  then  $\text{Ph}(\pi)$

can be reduced to its more usual representation:

if let  $| -1 \rangle$  then  $\text{Ph}(\pi)$ ; if let  $| -1 \rangle$  then  $\text{Ph}(\pi)$ ; if let  $| -1 \rangle$  then  $\text{Ph}(\pi)$

- Nested “if let” statements can be combined into a single “if let” statement with a composed pattern. For example, below the left term can be reduced to the right:

if let  $p$  then if let  $q$  then  $s \rightsquigarrow$  if let  $p \cdot q$  then  $s$

Patterns are not necessarily in the form of a single unitary or a simple pattern, but the evaluation reduces any arbitrary pattern  $p$  to the form  $s \cdot q$ , where  $s$  is a unitary term and  $q$  is a simple pattern. This allows normalisation to proceed by conjugating with the unitary  $s$ .

We evaluate terms in an *evaluation context*  $(q, l, r) : k \rightarrow n$ , where  $\vdash q : qm < qn$  is a simple pattern, and  $l, r \in \mathbb{N}$  such that  $l + k + r = m$ . We motivate our evaluation context as follows: the simple pattern  $q$  allows us to track that the term we are currently evaluating should be applied to a certain subspace, and allows us to evaluate under an “if let” statement. The numbers  $l$  and  $r$  record how many identities the term being evaluated has been whiskered with on either side.

- For a context  $(q, l, r) : k \rightarrow n$  and unitary term  $s : qk \leftrightarrow qk$ , its evaluation  $\text{eval}_{q,l,r}^u(s)$  is a list  $[c_1, \dots, c_N]$  where each  $c_i : qn \leftrightarrow qn$  is a normal clause. For such a list, its *inverse*  $[c_1, \dots, c_N]^\dagger = [c_N^\dagger, \dots, c_1^\dagger]$ , where  $c_i^\dagger$  is the result of the inversion meta operation defined in Section 3. We write  $c_1 \# c_2$  for the concatenation of lists  $c_1$  and  $c_2$ .
- For a context  $(q, l, r) : k \rightarrow n$  and a pattern  $p : qj < qk$ , its evaluation  $\text{eval}_{q,l,r}^p(p)$  is a tuple  $([c_1, \dots, c_N], q')$  where  $q' : l + j + r \rightarrow n$  is a simple pattern, and each  $c_i : qn \leftrightarrow qn$  is a normal clause.

For intuition, if  $\text{eval}_{q,l,r}^u(s) = [c_1, \dots, c_N]$ , then  $c_1; \dots; c_N$  should be equivalent to

$$\text{if let } q \text{ then } \text{id}_l \otimes s \otimes \text{id}_r$$

and if  $\text{eval}_{q,l,r}^p(p) = ([c_1, \dots, c_N], q')$  then  $(c_1; \dots; c_N) \cdot q'$  should be equivalent to

$$q \cdot (\text{id}_l \otimes p \otimes \text{id}_r)$$

We make this intuition precise in Section 6.

**Definition 9** (Substitution). Given a simple pattern  $q : qm < qn$  let  $q[|x\rangle / |i\rangle]$  be the result of substituting the  $i^{\text{th}}$  id in  $q$  with  $|x\rangle$ . For example, if  $q = |0\rangle \otimes \text{id} \otimes \text{id}$  then  $q[|-\rangle / |0\rangle] = |0\rangle \otimes |-\rangle \otimes \text{id}$ .

The evaluation functions are now defined by induction using the rules in Fig. 4. The normal term of  $s$  can then be extracted by composing the final list of clauses (taking the normal term to be  $\text{id}_n$  in the empty case, where  $n$  is the number of qubits of the input term).

We end the section by proving type soundness, which claims that our evaluation function preserves typing judgements.

**Theorem 10** (Type soundness). *Let  $(q, l, r) : k \rightarrow n$  be an evaluation context. The following rules are derivable:*

$$\frac{\vdash s : qk \leftrightarrow qk \quad \text{eval}_{q,l,r}^u(s) = [c_1, \dots, c_N]}{\vdash c_1; \dots; c_N : qn \leftrightarrow qn} \quad \frac{\vdash p : qm < qk \quad \text{eval}_{q,l,r}^p(p) = ([c_1, \dots, c_N], q')}{\vdash c_1; \dots; c_N : qn \leftrightarrow qn \quad (q', l, r) : m \rightarrow n}$$

PROOF. The proof proceeds by simultaneously proving both rules, mutually inducting on terms and patterns.  $\square$

$$\begin{array}{c}
\frac{\text{eval}_{q,l,r}^u(\text{Ph}(\theta)) = [\text{if let } q \text{ then } \text{Ph}(\theta) \otimes \text{id}_{l+r}]}{\text{eval}_{q,l,r}^u(\text{Ph}(\theta)) = [\text{if let } q \text{ then } \text{Ph}(\theta) \otimes \text{id}_{l+r}]} \quad \frac{\text{eval}_{q,l,r}^u(s) = c \quad \text{eval}_{q,l,r}^u(t) = c'}{\text{eval}_{q,l,r}^u(s; t) = c \# c'} \\
\frac{\vdash s : qk_1 \leftrightarrow qk_1 \quad \vdash t : qk_2 \leftrightarrow qk_2 \quad \text{eval}_{q,l,r+k_2}^u(s) = c \quad \text{eval}_{q,l+k_1,r}^u(t) = c'}{\text{eval}_{q,l,r}^u(s \otimes t) = c \# c'} \\
\frac{}{\text{eval}_{q,l,r}^u(\text{id}_k) = []} \quad \frac{\text{eval}_{q,l,r}^p(p) = (c, q') \quad \text{eval}_{q',l,r}^u(s) = c'}{\text{eval}_{q,l,r}^u(\text{if let } p \text{ then } s) = c^\dagger \# c' \# c} \quad \frac{x \in \{0, 1, +, -\}}{\text{eval}_{q,l,r}^p(|x\rangle) = ([], q[|x\rangle] / l)} \\
\frac{\text{eval}_{q,l,r}^u(s) = c}{\text{eval}_{q,l,r}^p(s) = (c, q)} \quad \frac{\text{eval}_{q,l,r}^p(p_1) = (c, q') \quad \text{eval}_{q',l,r}^p(p_2) = (c', q'')}{\text{eval}_{q,l,r}^p(p_1 \cdot p_2) = (c' \# c, q'')} \\
\frac{\vdash p_1 : qj_1 < qk_1 \quad \vdash p_2 : qj_2 < qk_2 \quad \text{eval}_{q,l,r+k_2}^p(p_1) = (c, q') \quad \text{eval}_{q',l+j_1,r}^p(p_2) = (c', q'')}{\text{eval}_{q,l,r}^p(p_1 \otimes p_2) = (c' \# c, q'')}
\end{array}$$

Fig. 4. Rules for defining the evaluation functions. In each case we assume that  $(q, l, r) : k \rightarrow n$ .

## 6 Categorical Semantics

We now formalise the meaning of our combinator language by equipping it with categorical semantics. Using this we will prove that the evaluation algorithm in Section 5 is valid in all models.

### 6.1 Dagger Categories

Except for measurement, quantum programming is intrinsically reversible, following the rules of quantum mechanics. We capture this reversibility mathematically by providing, for each morphism  $f : X \rightarrow Y$ , an associated morphism  $f^\dagger : Y \rightarrow X$  in the converse direction. Formally, a dagger category is a category equipped with an identity-on-objects involutive contravariant functor  $(-)^\dagger : \mathbf{C}^{\text{op}} \rightarrow \mathbf{C}$ , meaning that  $X^\dagger = X$  and  $(fg)^\dagger = g^\dagger f^\dagger$ .

**Example 11.** Here are some examples of relevant dagger categories for reversible programming.

The category with sets as objects and partial injective functions as morphisms is a dagger category. We write  $\mathbf{Pinj}$  for this category. The dagger of a function  $f$  is its partial inverse, as in the following example morphism  $f : \{0, 1\} \rightarrow \{0, 1\}$ .

$$f(x) = \begin{cases} 1 & \text{if } x = 0 \\ \text{undefined} & \text{otherwise} \end{cases} \quad f^\dagger(x) = \begin{cases} 0 & \text{if } x = 1 \\ \text{undefined} & \text{otherwise} \end{cases}$$

This category represents what we refer to as *classical* reversibility.

The category  $\mathbf{Con}$  of Hilbert spaces and contractive linear maps—*i.e.* maps with norm at most 1—is a dagger category where the dagger is the adjoint of linear maps. This category embodies *pure* quantum operations: it contains both unitaries and isometries, and so can model pure quantum states as well as programs.

Note that the dagger is not necessarily an inverse, but should rather be seen as a *partial* inverse. This gives us the mathematical leverage to provide a semantics to “if let” (see next subsection). In a dagger category, a morphism  $f$  is a dagger monomorphism (resp. epimorphism) if  $f^\dagger f = \text{id}$  (resp. if  $f f^\dagger = \text{id}$ ), and it is a dagger isomorphism if it is both dagger monic and epic [30].

**Example 12.** In **Pinj**, the dagger monomorphisms are the totally defined injective functions, and the dagger isomorphisms are the bijections. In **Con**, the dagger monomorphisms are the isometries and the dagger isomorphisms are the unitaries.

Later in this section, we interpret the programs in our languages as dagger isomorphisms (e.g. unitaries in a quantum setting), and patterns as dagger monomorphisms (e.g. isometries).

**Definition 13** (Zero morphism). A category has *zero morphisms* if for all  $X, Y$ , there exists  $0_{X,Y}: X \rightarrow Y$  such that  $0_{X,Y}^\dagger = 0_{Y,X}$ , and  $0_{X',Y} \circ f = 0_{X,Y} = g \circ 0_{X,Y'}$  for  $f: X \rightarrow X', g: Y' \rightarrow Y$ .

**Example 14.** In **Pinj** the zero morphism  $0_{X,Y}: X \rightarrow Y$  is the function defined on no input. In **Con**, it is the constantly zero linear map.

A *dagger preserving* functor  $F: \mathbf{C} \rightarrow \mathbf{D}$  between two dagger categories is such that  $F(f^\dagger) = F(f)^\dagger$  for any morphism  $f$  in  $\mathbf{C}$ . If  $(C, \otimes, I)$  is symmetric monoidal and  $\otimes$  is a dagger functor, we say that that  $(C, \otimes, I)$  is dagger symmetric monoidal.

A *dagger rig category* is a dagger category equipped with symmetric monoidal structures  $(\otimes, I)$  and  $(\oplus, O)$ , such that  $\otimes$  and  $\oplus$  are dagger functors, with their coherence isomorphisms, and with additional natural dagger isomorphisms (satisfying coherence conditions [38]):

$$\begin{aligned} (X \oplus Y) \otimes Z &\xrightarrow{\sim} (X \otimes Z) \oplus (Y \otimes Z), & O \otimes X &\xrightarrow{\sim} O, \\ Z \otimes (X \oplus Y) &\xrightarrow{\sim} (Z \otimes X) \oplus (Z \otimes Y), & X \otimes O &\xrightarrow{\sim} O. \end{aligned}$$

**Example 15.** The categories **Pinj** and **Con** are dagger rig categories. The monoidal structures are the usual ones for **Pinj**: the product is the product of sets, its unit is a singleton, the sum is the disjoint union of sets, and its unit is the empty set. In **Con**, the product is the tensor product, and its unit is a one-dimensional Hilbert space. Its sum is the direct sum of spaces, and the unit is the zero-dimensional Hilbert space  $\{0\}$ .

The rig structure plays a central part in our semantics. We later make sense of the subspaces pointed out by patterns in the language with the  $\oplus$  tensor, through the notion of *independent coproducts* (see Theorem 17). In this view, for example,  $X_1$  and  $X_2$  are some subspaces of  $X_1 \oplus X_2$ . If we interpret qubits as  $I \oplus I$ , this means that the space generated by  $|0\rangle$  (resp.  $|1\rangle$ ) are subspaces—and they are not the only ones.

Now consider programs that act on the space  $(X_1 \oplus X_2) \otimes Y$ . The rig structure allows us to identify  $(X_1 \oplus X_2) \otimes Y$  with  $(X_1 \otimes Y) \oplus (X_2 \otimes Y)$ , which means that  $X_1 \otimes Y$  and  $X_2 \otimes Y$  are some subspaces of  $(X_1 \oplus X_2) \otimes Y$ . This identifies subspaces with patterns of the form  $p \otimes \text{id}_n$ .

We later prove that we can derive a dagger rig structure with independent coproducts (see Theorem 17) that are preserved by the tensor product (see Theorem 22).

## 6.2 Independent Coproducts

Our syntax in Section 3 relies on the “if let” conditional, which is an instance of case splitting. Case splitting in programming languages is usually interpreted with a *disjointness* structure [2]. For example, if the case splitting happens on a Boolean, the set of potential results is  $X_1 \sqcup X_2$ , where  $X_1$  happens when the Boolean is true, and  $X_2$  when it is false. In category theory, we capture this behaviour with coproducts, which are cospans:  $X_1 \xrightarrow{i_1} X_1 + X_2 \xleftarrow{i_2} X_2$  such that for all  $f_1: X_1 \rightarrow Y$  and  $f_2: X_2 \rightarrow Y$ , there is a unique mediating morphism  $m: X_1 + X_2 \rightarrow Y$  such that the

diagram

$$\begin{array}{ccccc}
 X_1 & \xrightarrow{i_1} & X_1 + X_2 & \xleftarrow{i_2} & X_2 \\
 & \searrow f_1 & \downarrow m & \swarrow f_2 & \\
 & & Y & & 
 \end{array} \tag{1}$$

commutes, without any special conditions on  $f_1$  and  $f_2$ . In our setting, we also need to ensure our program remains reversible and has a reversible semantics. In the particular case of case splitting, this reversibility condition requires that we decide deterministically whether the result of type  $Y$  through  $m$  came from the map  $f_1$  or the map  $f_2$ .

To do so, we introduce a notion of independence of morphisms, characterised by the condition  $f_1^\dagger f_2 = 0$  (see Definition 16), where  $f_1^\dagger$  is thought of as the (partial) inverse of  $f_1$ . This fits the notion of compatibility of morphisms in classical reversible settings [34, Definition 8] and the one for linear maps between Hilbert spaces [18, Section 2.3].

**Definition 16** (Independent cospan). In a dagger category with zero morphisms, an *independent cospan* is a pair of morphisms  $(f_1: X_1 \rightarrow Y, f_2: X_2 \rightarrow Y)$  such that  $f_1^\dagger f_2 = 0$ .

Additionally, we introduce a universal equivalent to independent cospans. In the same vein as coproducts, an *independent coproduct* is an independent cospan for which there exists a mediating morphism with any other independent cospan. An independent coproduct is also jointly epic, ensuring uniqueness of not only mediating morphisms, but of all morphisms that make the diagrams such as (1) commute. This uniqueness is essential to prove that our programs are indeed interpreted as dagger isomorphisms.

**Definition 17** (Independent coproduct). An *independent coproduct* in a dagger category with zero morphisms is a jointly epic independent cospan  $(\iota_1: X_1 \rightarrow X, \iota_2: X_2 \rightarrow X)$  such that  $\iota_1$  and  $\iota_2$  are dagger monomorphisms, and for all independent cospans  $(f_1: X_1 \rightarrow Y, f_2: X_2 \rightarrow Y)$ , there is a unique morphism  $u: X \rightarrow Y$ , making the following diagrams commute.

$$\begin{array}{ccc}
 X_1 & \xrightarrow{\iota_1} & X & \xleftarrow{\iota_2} & X_2 \\
 & \searrow f_1 & \downarrow u & \swarrow f_2 & \\
 & & Y & & 
 \end{array}
 \qquad
 \begin{array}{ccc}
 & X_1 & \\
 \iota_1 \swarrow & & \searrow f_1 \\
 X & \xrightarrow{\iota_2^\dagger} & X_2 & \xleftarrow{f_2^\dagger} & Y
 \end{array}$$

**Lemma 18.** Let  $p$  be a morphism. If both  $(p, c)$  and  $(p, d)$  are independent coproducts, then  $cc^\dagger = dd^\dagger$ .

It also implies that  $d^\dagger c$  is a dagger isomorphism. Therefore  $(p, c)$  and  $(p, d)$  are the same independent coproduct up to dagger isomorphism.

**Definition 19** (Independent coproducts). A category has *independent coproducts* if for all pairs of objects  $X_1, X_2$ , there exists an object  $X_1 \oplus X_2$  and chosen morphisms  $\iota_i: X_i \rightarrow X_1 \oplus X_2$  such that  $(\iota_1, \iota_2)$  is an independent coproduct. Given an independent cospan  $(f_1, f_2)$ , its mediating morphism with the chosen independent coproduct is written  $[f_1, f_2]$ .

**Theorem 20.** If  $C$  has independent coproducts and a zero object  $O$ , it is symmetric monoidal.

**Example 21.** Both **Plnj** and **Con** have independent coproducts, given by their usual notion of direct sum.

In our language, we also need to interpret the tensor product, which manifests as a monoidal product. If this monoidal product is compatible with independent coproducts (as described below), we obtain a dagger rig category.

**Definition 22** (Preservation of independent coproducts). We say that  $(C, \otimes, I)$  *preserves independent coproducts* if for all independent coproducts  $(t_1, t_2)$  and objects  $Y$ , then  $(t_1 \otimes \text{id}_Y, t_2 \otimes \text{id}_Y)$  is also an independent coproduct.

**Theorem 23.** *If  $(C, \otimes, I)$  is a dagger symmetric monoidal category with a monoidal zero object  $O$  (namely, equipped with dagger isomorphisms  $O \otimes X \cong O$ ), and independent coproducts preserved by the monoidal structure, then  $C$  is a dagger rig category.*

**Example 24.** Our two examples **Pinj** and **Con** have such a monoidal structure, and with independent coproducts, are rig categories under the conditions of Theorem 23. In fact, the category **Pinj** is the standard one to model classical reversible programming [12, 35]. It has many characteristics in common with **Con**, as highlighted by their axioms [29]. However, Axiom (5) for **Con** shows that ‘mixture occurs’: there is a map  $I \rightarrow I \oplus I$  that is orthogonal to neither injection; physicists call this ‘superposition’. In this, we present **Con** as suitable for quantum computing, as opposed to **Pinj**.

### 6.3 Semantics of “if let”

Let  $(C, \otimes, I)$  be a dagger symmetric monoidal category with a monoidal zero object and independent coproducts, equipped with a chosen, *distinguished* independent coproduct  $I \xrightarrow{r_1} I \oplus I \xleftarrow{r_2} I$  and a family of scalars  $\phi_\theta: I \rightarrow I$  for  $\theta \in \mathbb{R}$  satisfying  $\phi_\theta \circ \phi_{\theta'} = \phi_{\theta+\theta'}$ ,  $\phi_\theta^\dagger = \phi_{-\theta}$  and  $\phi_0 = \text{id}_I$ .

**Remark 25.** We choose here to have a phase group parameterised by real numbers, helping for a simpler presentation of both the syntax and its semantics. Note that the approach to phase can be more refined, and we could parameterise with any group.

We provide a semantics of our language in the category  $C$ . We first fix the semantics for qubit types as:  $\llbracket q0 \rrbracket = I$ , and  $\llbracket q(n+1) \rrbracket = \llbracket qn \rrbracket \otimes (I \oplus I)$ . Note that, due to the rig structure (see Theorem 23), there is an isomorphism  $\llbracket q(n+m) \rrbracket \cong \llbracket qn \rrbracket \otimes \llbracket qm \rrbracket$  for all  $n, m$ . In the rest of the section, we allow ourselves to write “ $\cong$ ” for coherence morphisms, to keep apparent only the key points of the semantics. Both well-formed terms  $qn \leftrightarrow qn$  and patterns  $qn < qm$  are interpreted as morphisms  $\llbracket qn \rrbracket \rightarrow \llbracket qn \rrbracket$  and  $\llbracket qn \rrbracket \rightarrow \llbracket qm \rrbracket$ , and we later prove that the interpretation of a term is a dagger isomorphism, and the one of a pattern is dagger monic (see Theorem 27).

**Remark 26.** The data types used in the syntax are only qubits, and the semantics of a qubit data is  $I \oplus I$ . While we keep this restriction, the content of this section is generalisable to any size of data (for example qutrits, modelled as  $I \oplus I \oplus I$ ). This would, however, require a different syntax.

The full semantics of the language is detailed in Figure 5. The cornerstone of the language is the “if let” expression, and the semantics of the other terms follow easily. Given a pattern  $\vdash p: qn < qm$ ,

assume that  $\llbracket \vdash p: qn < qm \rrbracket$  is part of an independent coproduct  $\llbracket qn \rrbracket \xrightarrow{\llbracket p \rrbracket} \llbracket qm \rrbracket \xleftarrow{\llbracket p \rrbracket^\perp} \bullet$ . The choice of  $\llbracket p \rrbracket^\perp$  does not matter, up to dagger isomorphism. With a unitary  $\vdash s: qn \leftrightarrow qn$ , we have an independent cospan  $(\llbracket p \rrbracket \llbracket s \rrbracket, \llbracket p \rrbracket^\perp)$  whose mediating morphism with  $(\llbracket p \rrbracket, \llbracket p \rrbracket^\perp)$  is written  $\llbracket \llbracket p \rrbracket \llbracket s \rrbracket, \llbracket p \rrbracket^\perp \rrbracket$  as shown below, which we let be the semantics of  $\vdash$  if let  $p$  then  $s: qm \leftrightarrow qm$ .

$$\begin{array}{ccc}
 \llbracket qn \rrbracket & \xrightarrow{\llbracket p \rrbracket} & \llbracket qm \rrbracket \xleftarrow{\llbracket p \rrbracket^\perp} \bullet \\
 \searrow \llbracket p \rrbracket \llbracket s \rrbracket & & \swarrow \llbracket p \rrbracket^\perp \\
 & \llbracket \llbracket p \rrbracket \llbracket s \rrbracket, \llbracket p \rrbracket^\perp \rrbracket & \\
 & \downarrow & \\
 & \llbracket qm \rrbracket &
 \end{array}
 \quad \bullet \quad
 \begin{array}{ccc}
 \llbracket qn \rrbracket & \xrightarrow{\llbracket p \rrbracket} & \llbracket qm \rrbracket \xleftarrow{\llbracket p \rrbracket^\perp} \bullet \\
 \downarrow \llbracket s \rrbracket & & \downarrow \text{id} \\
 \llbracket qn \rrbracket & \xrightarrow{\llbracket p \rrbracket} & \llbracket qm \rrbracket \xleftarrow{\llbracket p \rrbracket^\perp} \bullet \\
 & \llbracket \llbracket p \rrbracket \llbracket s \rrbracket, \llbracket p \rrbracket^\perp \rrbracket &
 \end{array}
 \quad (2)$$

In other words, the dagger monomorphism  $\llbracket p \rrbracket$  works as a subobject, on which  $\llbracket s \rrbracket$  is applied. The resulting  $\llbracket \llbracket p \rrbracket \llbracket s \rrbracket, \llbracket p \rrbracket^\perp \rrbracket$  is the morphism that acts as  $\llbracket s \rrbracket$  on the subobject identified as  $\llbracket p \rrbracket$ , and that acts as the identity on its orthogonal subobject  $\llbracket p \rrbracket^\perp$ .

$$\begin{aligned}
\llbracket \vdash \text{Ph}(\theta) : q0 \leftrightarrow q0 \rrbracket &= \phi_\theta \\
\llbracket \vdash \text{id}_n : qn \leftrightarrow qn \rrbracket &= \text{id}_{\llbracket qn \rrbracket} \\
\llbracket \vdash s; t : qn \leftrightarrow qn \rrbracket &= \llbracket \vdash t : qn \leftrightarrow qn \rrbracket \circ \llbracket \vdash s : qn \leftrightarrow qn \rrbracket \\
\llbracket \vdash s \otimes t : q(n+m) \leftrightarrow q(n+m) \rrbracket &= \cong \circ (\llbracket \vdash s : qn \leftrightarrow qn \rrbracket \otimes \llbracket \vdash t : qn \leftrightarrow qn \rrbracket) \circ \cong \\
\llbracket \vdash \text{if let } p \text{ then } s : qm \leftrightarrow qm \rrbracket &= [\llbracket p \rrbracket \llbracket s \rrbracket, \llbracket p \rrbracket^\perp] \\
\llbracket \vdash |0\rangle : q0 < q1 \rrbracket &= \iota_1 & \llbracket \vdash |1\rangle : q0 < q1 \rrbracket &= \iota_2 \\
\llbracket \vdash |+\rangle : q0 < q1 \rrbracket &= r_1 & \llbracket \vdash |-\rangle : q0 < q1 \rrbracket &= r_2 \\
\llbracket \vdash s : qn < qn \rrbracket &= \llbracket \vdash s : qn \leftrightarrow qn \rrbracket \\
\llbracket \vdash p \cdot q : ql < qm \rrbracket &= \llbracket \vdash p : qn < qm \rrbracket \circ \llbracket \vdash q : ql < qn \rrbracket \\
\llbracket \vdash p \otimes q : q(n+n') < q(m+m') \rrbracket &= \cong \circ (\llbracket \vdash p : qn < qm \rrbracket \otimes \llbracket \vdash q : qn' < qm' \rrbracket) \circ \cong
\end{aligned}$$

Fig. 5. Categorical semantics, defined by induction of the type derivation.

We prove, by induction on the typing derivations, that our semantics is well-defined. In particular, we show that programs  $s$  are interpreted as dagger isomorphisms (meaning that  $\llbracket s \rrbracket^\dagger \llbracket s \rrbracket = \text{id}$  and  $\llbracket s \rrbracket \llbracket s \rrbracket^\dagger = \text{id}$ ) and that patterns  $p$  are interpreted as dagger monomorphisms (meaning that  $\llbracket p \rrbracket^\dagger \llbracket p \rrbracket = \text{id}$ ). The well-definedness also heavily relies on the existence of an *orthogonal* morphism to  $\llbracket p \rrbracket$  for all  $p$ , in order to compute the “if let” statement as in (2).

**Theorem 27.** *For any well-typed pattern  $p$  and unitary  $s$ , we have that:*

- *there exists a morphism  $\llbracket p \rrbracket^\perp$  such that  $(\llbracket p \rrbracket, \llbracket p \rrbracket^\perp)$  is an independent coproduct;*
- *the morphism  $\llbracket p \rrbracket^\perp$  is unique up to dagger isomorphism;*
- *the morphism  $\llbracket p \rrbracket$  is a dagger monomorphism;*
- *the morphism  $\llbracket s \rrbracket$  is a dagger isomorphism.*

Note that given a well-typed pattern  $p$ , we can give a formula for  $\llbracket p \rrbracket^\perp$ , as follows:

$$\llbracket |0\rangle \rrbracket^\perp = \llbracket |1\rangle \rrbracket \quad \llbracket |+\rangle \rrbracket^\perp = \llbracket |-\rangle \rrbracket \quad \llbracket s \rrbracket^\perp = 0$$

$$\llbracket p \cdot q \rrbracket^\perp = [\llbracket p \rrbracket \llbracket q \rrbracket^\perp, \llbracket p \rrbracket^\perp] \circ \cong \quad \llbracket p \otimes q \rrbracket^\perp = \cong \circ [\llbracket p \rrbracket^\perp \otimes \llbracket q \rrbracket, \llbracket p \rrbracket \otimes \llbracket q \rrbracket^\perp, \llbracket p \rrbracket^\perp \otimes \llbracket q \rrbracket^\perp] \circ \cong$$

in which our use of  $\cong$  contains only coherence morphisms [38] that are identities in after the semi-strictification of rig categories (namely, all coherence morphisms except the multiplicative symmetry and the right distributor).

**Remark 28.** The category  $\mathbf{PInj}$  is a (degenerate) model of the language. It has all the structure required, with  $\phi_\theta = \text{id}_I$  for all  $\theta$ , and  $r_i = \iota_i$  for  $i \in \{1, 2\}$ . Naturally, the best model for our language is  $\mathbf{Con}$ , where patterns are interpreted as isometries, and programs as unitaries.

We showcase some equations that hold in any model of the language.

**Proposition 29.** *If  $p$  and  $q$  are well-typed patterns, and  $s$  and  $t$  are well-typed terms, then we have:*

- $\llbracket \text{if let } p \text{ then if let } q \text{ then } s \rrbracket = \llbracket \text{if let } p \cdot q \text{ then } s \rrbracket;$
- $\llbracket \text{if let } p \text{ then } (s; t) \rrbracket = \llbracket \text{if let } p \text{ then } s; \text{if let } p \text{ then } t \rrbracket;$
- $\llbracket \text{if let } t \text{ then } s \rrbracket = \llbracket t \rrbracket \llbracket s \rrbracket \llbracket t \rrbracket^\dagger.$

The categorical dagger agrees with the syntactic inverse (see Definition 2).

**Proposition 30.** *Given a well-typed term  $s$ , we have that  $\llbracket s \rrbracket^\dagger = \llbracket s^\dagger \rrbracket.$*

## 6.4 Soundness

We now show that any categorical model  $\mathcal{C}$  is sound with respect to the evaluation function for the compilation. We prove that the compilation procedure does not alter the semantics of the program.

Each normal clause  $c_i$  already has a well-defined semantics (see Theorem 27), and we define the semantics of a list of clauses  $[c_1, \dots, c_l]$  as the composition  $\llbracket c_l \rrbracket \circ \dots \circ \llbracket c_1 \rrbracket$ . In the following, we allow ourselves to loosely write  $c$  for a list of clauses, and  $\llbracket c \rrbracket$  for its semantics in a categorical model  $\mathcal{C}$ . Since  $q$  designates the subspace on which the program is applied, we should have that if  $\text{eval}_{q,l,r}^u(s) = c$ , then  $\llbracket c \rrbracket$  is the mediating morphism below:

$$\begin{array}{ccccc}
 \llbracket qm \rrbracket & \xrightarrow{\llbracket q \rrbracket} & \llbracket qn \rrbracket & \xleftarrow{\llbracket q \rrbracket^\perp} & \bullet \\
 \xi^\dagger \downarrow & & \downarrow \llbracket c \rrbracket & & \downarrow \text{id} \\
 \llbracket ql \rrbracket \otimes \llbracket qk \rrbracket \otimes \llbracket qr \rrbracket & & & & \\
 \text{id} \otimes \llbracket p \rrbracket \otimes \text{id} \downarrow & & & & \\
 \llbracket ql \rrbracket \otimes \llbracket qk \rrbracket \otimes \llbracket qr \rrbracket & & & & \\
 \xi \downarrow & & & & \\
 \llbracket qm \rrbracket & \xrightarrow{\llbracket q \rrbracket} & \llbracket qn \rrbracket & \xleftarrow{\llbracket q \rrbracket^\perp} & \bullet
 \end{array}$$

$\llbracket \text{id}_l \otimes s \otimes \text{id}_r \rrbracket$  (curved arrow from top-left to bottom-left)

where  $\xi: \llbracket ql \rrbracket \otimes \llbracket qk \rrbracket \otimes \llbracket qr \rrbracket \rightarrow \llbracket qm \rrbracket$  is the coherence isomorphism that rewrites one object into the other, since we know that  $l + k + r = m$ , but  $\xi$  basically acts as the identity. Intuitively, we get that the semantics of  $\text{eval}_{q,l,r}^u(s)$  is equal to the one of the term: if let  $q$  then  $(\text{id}_l \otimes s \otimes \text{id}_r)$ .

In the case of  $\text{eval}_{q,l,r}^p(p) = (c, q')$ , the intuition is that the semantics of the pattern  $c \cdot q'$  is equal to the one of  $q \cdot (\text{id}_l \otimes p \otimes \text{id}_r)$ ; and their orthogonal morphisms are equal as well.

**Lemma 31** (Induction hypothesis). *We have the following:*

- If  $\text{eval}_{q,l,r}^u(s) = c$ , then  $\llbracket c \rrbracket$  is the mediating morphism in the diagram above; in other words, the unique morphism such that:

$$\begin{cases} \llbracket c \rrbracket \llbracket q \rrbracket = \llbracket q \rrbracket \xi (\text{id} \otimes \llbracket s \rrbracket \otimes \text{id}) \xi^\dagger \\ \llbracket c \rrbracket \llbracket q \rrbracket^\perp = \llbracket q \rrbracket^\perp; \end{cases}$$

- if  $\text{eval}_{q,l,r}^p(p) = (c, q')$ , then we have that:  $\llbracket c \rrbracket \llbracket q' \rrbracket = \llbracket q \rrbracket \xi (\text{id} \otimes \llbracket p \rrbracket \otimes \text{id}) \xi'^\dagger$ .

Soundness is a direct corollary: we can conclude that the compilation scheme described by the evaluation function does not alter the program.

**Theorem 32** (Soundness). *Given a well-formed unitary term  $s$ , we have  $\llbracket s \rrbracket = \llbracket \text{eval}_{\text{id}^{\otimes n}, 0, 0}^u(s) \rrbracket$ .*

## 7 Beyond Combinators

In this final section, we give two nominal variations of the language, exploring how the “if let” construction could manifest in different contexts.

- A functional language: here we allow qubits to be bound to linear variables, and unitaries are treated as functions which consume input qubits, producing new output qubits. This allows a more concise syntax where a unitary can be applied to a subset of the available qubits without inserting explicit identity unitaries. A consequence of this format is that there arises a native implementation of the “swap” gate on two qubits, which in turn allows swaps to appear within “if let” blocks.

- An imperative language: in the imperative model, variables are viewed as representing the *location* of a qubit (i.e. their denotation is given by an inclusion into some ambient space) instead of viewing variables as storing the state of a qubit at some fixed point in time. In this model, variables need not be treated linearly, and unitaries are given by expressions rather than functions, and are viewed as mutating the variables they act on, rather than consuming them.

The following subsections briefly sketch these languages, exploring their differences to the core combinator language.

## 7.1 Functional

In the functional variant, we introduce variables which intuitively store the quantum state at intermediate stages of the program. Gates can then be explicitly applied to any specific qubits available, rather than requiring them to be tensored with appropriate identity operations to apply them to specific qubits.

With access to variables, the typing judgements must now be parameterised by a context. As unitaries are no longer the primitive constructions, we remove the unitary type  $qn \leftrightarrow qn$ , and instead have the following typing judgement for terms:

$$\Gamma \vdash s : qn$$

The context  $\Gamma$  is a list  $x_1 : qn_1, x_2 : qn_2, \dots$  where the  $x_i$  are variable names. We write  $\cdot$  for the empty context and  $\Gamma, \Delta$  for the concatenation of contexts  $\Gamma$  and  $\Delta$ . As is usual in the literature, we treat terms as equal up to  $\alpha$ -equivalence.

Instead of the composition operator present in the combinator language, we now have a “let” expression, allowing us to bind the (possibly multiple) outputs of a term. In general this will have the form:

$$\text{let } x_1 \otimes \dots \otimes x_n = s \text{ in } t$$

We refer to expressions of the form  $x_1 \otimes \dots \otimes x_n$  as *copatterns*, as they are also exactly the terms that can be matched to a pattern in an “if let” statement. To make this precise, we specify three classes of syntax:  $\text{Copattern} \subset \text{Unary} \subset \text{Pattern}$ . All three syntax classes are generated inductively, from the following grammars:

Copattern:  $c ::= x \mid c_1 \otimes c_2$

Unary:  $s, t ::= x \mid s \otimes t \mid \text{Ph}(\theta) \mid \text{let } c = s \text{ in } t \mid \text{if let } p = c \text{ then } s$

Pattern:  $p, q ::= x \mid p \otimes q \mid \text{Ph}(\theta) \mid \text{let } c = p \text{ in } q \mid \text{if let } p = c \text{ then } s \mid |0\rangle \mid |1\rangle \mid |+\rangle \mid |-\rangle$

where  $x$  represents a variable.

Within this syntax, we treat variables linearly, which corresponds to the deletion and copying of qubits being forbidden. This is reflected in the typing rules for terms, found below.

$$\frac{}{x : qn \vdash x : qn} \quad \frac{}{\cdot \vdash |0\rangle : q1} \quad \frac{}{\cdot \vdash |1\rangle : q1} \quad \frac{}{\cdot \vdash |+\rangle : q1} \quad \frac{}{\cdot \vdash |-\rangle : q1}$$

$$\frac{\Gamma \vdash s : qn \quad \Delta \vdash t : qm}{\Gamma, \Delta \vdash s \otimes t : q(n+m)} \quad \frac{}{\cdot \vdash \text{Ph}(\theta) : q0} \quad \frac{\Theta \vdash s : qm \quad \Delta \vdash c : qm \quad \Gamma, \Delta \vdash t : qn}{\Gamma, \Theta \vdash \text{let } c = s \text{ in } t : qn}$$

$$\frac{\Delta \vdash c : qm \quad \Theta \vdash p : qm \quad \Theta \vdash s : qn}{\Delta \vdash \text{if let } p = c \text{ then } s : qm}$$

With the above typing rules the term

$$\text{let } x' \otimes z' = (\text{if let } |1\rangle \otimes |-\rangle = x \otimes z \text{ then Ph}(\pi)) \text{ in } x' \otimes y \otimes z'$$

in the context  $x : q1, y : q1, z : q1$ , which applies a CX gate to the first and third qubits, is not well-typed, as it uses variables “in the wrong order”. To remedy this, the following exchange rule can be added.

$$\frac{\Gamma, x : qn, y : qm, \Delta \vdash s : ql}{\Gamma, y : qm, x : qn, \Delta \vdash s : ql}$$

Introducing this exchange rule has an unintended side effect; swaps (and therefore their controlled variants) are now natively representable within the language. For example:

$$x : q1, y : q1, z : q1 \vdash \text{if let } |1\rangle \otimes y' \otimes z' = x \otimes y \otimes z \text{ then } z' \otimes y' : q3$$

Performs a swap of  $y$  and  $z$ , controlled on  $x$ .

In the term above, we were required to “rebind”  $y$  and  $z$  to  $y'$  and  $z'$ , as this functional variant does not allow “variable capture”: the use of variables in the body of the “if let” which were defined outside the pattern. In principle, the typing rules could be modified to allow this, however doing so creates ambiguity in the semantics, as it can be unclear whether swaps occur inside the “if let” block (possibly being controlled) as opposed to happening before.

## 7.2 Semantics of the Functional Language

Let  $(\mathcal{C}, \otimes, I)$  be a dagger symmetric monoidal category with a zero object (such that  $O \otimes X \cong O$  for all  $X$ ) and independent coproducts, equipped with a chosen, *distinguished* independent coproduct  $I \xrightarrow{r_1} I \oplus I \xleftarrow{r_2} I$  and a family of scalars  $\phi_\theta : I \rightarrow I$  for  $\theta \in \mathbb{R}$  satisfying  $\phi_\theta \circ \phi_{\theta'} = \phi_{\theta+\theta'}$ ,  $\phi_\theta^\dagger = \phi_{-\theta}$  and  $\phi_0 = \text{id}_I$ .

The semantics of a judgement  $\Gamma \vdash p : qn$  is given as a morphism  $\llbracket \Gamma \vdash p : qn \rrbracket : \llbracket \Gamma \rrbracket \rightarrow \llbracket qn \rrbracket$  where  $\llbracket \Gamma \rrbracket = \llbracket qn_1 \rrbracket \otimes \cdots \otimes \llbracket qn_k \rrbracket$  for  $\Gamma = x_1 : qn_1, \dots, x_k : qn_k$ . Note that given the latter  $\Gamma$ , there is a canonical copattern  $c_\Gamma = x_1 \otimes \cdots \otimes x_k$ , such that  $\Gamma \vdash c_\Gamma : q(n_1 + \dots + n_k)$ .

We define the following semantics:

$$\begin{aligned} \llbracket x : qn \vdash x : qn \rrbracket &= \text{id}_{\llbracket qn \rrbracket} \\ \llbracket \cdot \vdash |0\rangle : q1 \rrbracket &= \iota_1 & \llbracket \cdot \vdash |1\rangle : q1 \rrbracket &= \iota_2 \\ \llbracket \cdot \vdash |+\rangle : q1 \rrbracket &= r_1 & \llbracket \cdot \vdash |-\rangle : q1 \rrbracket &= r_2 \\ \llbracket \cdot \vdash \text{Ph}(\theta) : q0 \rrbracket &= \phi_\theta \\ \llbracket \Gamma, \Delta \vdash s \otimes t : q(n+m) \rrbracket &= \llbracket \Gamma \vdash s : qn \rrbracket \otimes \llbracket \Delta \vdash t : qm \rrbracket \\ \llbracket \Gamma, \Theta \vdash \text{let } c = s \text{ in } t : qn \rrbracket &= \llbracket \Gamma, \Delta \vdash t : qn \rrbracket \circ \left( \text{id}_{\llbracket \Gamma \rrbracket} \otimes \left( \llbracket \Delta \vdash c : qm \rrbracket^\dagger \circ \llbracket \Theta \vdash s : qm \rrbracket \right) \right) \\ \llbracket \Delta \vdash \text{if let } p = c \text{ then } s : qm \rrbracket &= u \circ \llbracket \Delta \vdash c : qm \rrbracket \end{aligned}$$

where  $u$  is the mediating morphism in:

$$\begin{array}{ccccc} \llbracket \Theta \rrbracket & \xrightarrow{\llbracket p \rrbracket} & \llbracket qm \rrbracket & \xleftarrow{\llbracket p \rrbracket^\dagger} & \bullet \\ \llbracket s \rrbracket \downarrow & & \vdots & & \downarrow \text{id} \\ \llbracket qn \rrbracket & & \downarrow u & & \\ \llbracket c_\Theta \rrbracket \downarrow & & & & \\ \llbracket \Theta \rrbracket & \xrightarrow{\llbracket p \rrbracket} & \llbracket qm \rrbracket & \xleftarrow{\llbracket p \rrbracket^\dagger} & \bullet \end{array} \quad (3)$$

We expect to have the following results, with a similar proof as for Theorem 27.

- If we have  $\Gamma \vdash p : qn$  with  $p$  a pattern, then  $\llbracket \Gamma \vdash p : qn \rrbracket$  is an isometry.
- If we have  $\Gamma \vdash c : qn$  with  $c$  a unitary, then  $\llbracket \Gamma \vdash c : qn \rrbracket$  is a unitary.

### 7.3 Imperative

The final version of the syntax is an imperative syntax. By exploiting that unitaries always return the same number of qubits as they are input, we can instead view a unitary as a procedure that *mutates* its input qubits, and has no return value.

In this syntax, quantum computation is *effective*. Under this view, our qubit variables now act as qubit “locations” rather than qubit states, and can be freely copied or unused, allowing the linearity condition to be dropped. We note that in this syntax, copying a variable is more akin to creating an alias, rather than duplicating a qubit state (which cannot be done by the no-cloning theorem).

Removing the need for return values comes with various advantages:

- The overall syntax becomes more concise.
- It is no longer possible to natively represent swaps as in the functional variant, yet we do not have a rigid qubit order as in the combinator variant.
- It is much easier to give a typing rule to the “if let” clause which allows “variable capture”, the use of variables defined outside the pattern.

To demonstrate the last point, we consider the following program:

$$x : q1, y : q1 \vdash (\text{if let } |1\rangle = y \text{ then } X[x]); H[x]$$

We are free here to use the variable  $x$  in the body of the “if let” clause, despite it not appearing in the pattern  $|1\rangle$ . Contrary to the functional variation of the syntax, the variable  $x$  is not treated linearly, and so we are free to continue using  $x$  after the “if let” clause. We also note that this program can be viewed as using the variables  $x$  and  $y$  “in the wrong order”. The translation of this program into the functional syntax would require multiple uses of the exchange typing rule, yet in this syntax the program can be naturally interpreted without any swap gates.

A disadvantage of the imperative approach is its treatment of patterns. Patterns naturally have a more functional treatment, causing the resulting language to contain functional and imperative components. For example, in the pattern  $H \cdot |0\rangle$ , the term  $H$  acts as a function which outputs a pattern, whereas in the expression  $H[x]$ , the term  $H$  acts on a qubit  $x$  which it is then viewed as mutating, returning nothing. To remedy this, we add a sort of gates, which bind the free variables in a unitary, and allow them to be embedded into a pattern.

The syntax for this language has the following grammar, where  $\Gamma$  is a context and  $x$  is a variable:

$$\begin{aligned} \text{Copattern: } \quad c & ::= x \mid c_1 \otimes c_2 \\ \text{Pattern: } \quad p, q & ::= x \mid p \otimes q \mid g \cdot p \mid |0\rangle \mid |1\rangle \mid |+\rangle \mid |-\rangle \\ \text{Unitary: } \quad s, t & ::= \text{id} \mid s; t \mid \text{Ph}(\theta) \mid \text{if let } p = c \text{ then } s \\ \text{Gate: } \quad g & ::= \Gamma \mapsto s \end{aligned}$$

As in the functional syntax, Copattern  $\subset$  Pattern, but here the syntax for unitaries is separated.

We type this language with three judgements, one for pattern and copattern terms, one for unitary terms, and one for gates.

$$\Gamma \vdash p : qn \quad \Gamma \vdash s \quad \vdash g : qn \leftrightarrow qn$$

The typing rules for patterns and copatterns are:

$$\frac{}{x : qn \vdash x : qn} \quad \frac{\Gamma \vdash s : qn \quad \Delta \vdash t : qm}{\Gamma, \Delta \vdash s \otimes t : q(n+m)} \quad \frac{\vdash g : qn \leftrightarrow qn \quad \Gamma \vdash p : qn}{\Gamma \vdash g \cdot p : qn}$$

$$\frac{}{\cdot \vdash |0\rangle : q1} \quad \frac{}{\cdot \vdash |1\rangle : q1} \quad \frac{}{\cdot \vdash |+\rangle : q1} \quad \frac{}{\cdot \vdash |-\rangle : q1}$$

The typing rules for unitaries being:

$$\frac{}{\Gamma \vdash \text{id}} \quad \frac{\Gamma \vdash s \quad \Gamma \vdash t}{\Gamma \vdash s; t} \quad \frac{}{\Gamma \vdash \text{Ph}(\theta)} \quad \frac{\Delta \vdash c : qm \quad \Theta \vdash p : qm \quad \Gamma, \Theta \vdash s}{\Gamma, \Delta \vdash \text{if let } p = c \text{ then } s}$$

Lastly, the unique typing rule for gates is the following, where  $\#\Gamma$  is the number obtained by summing the indices of each type in  $\Gamma$ :

$$\frac{\Gamma \vdash s \quad \#\Gamma = n}{\vdash \Gamma \mapsto s : qn \leftrightarrow qn}$$

In addition, we allow typing rules which permute the context, much like in the functional version of the syntax, though note that due to the lack of return values in this syntax, there is no way to create a “native” controlled swap, (i.e. without inserting a swap gate using quantum operations).

We further note that the typing rule here for the “if let” expression allows variable capture, without any of the complications that arised in the functional syntax.

## 8 Future Work

We conclude by discussing some future directions for the development of languages using the constructs introduced in this work. An immediate direction is to extend the compilation and denotational semantics to the functional and imperative variants of the language presented in Section 7.

In this paper we have focussed on using the “if let” construction as a device for writing quantum programs, yet it could also find use as a representation of quantum programs within an optimising compiler. At this higher level of abstraction than quantum circuits, more global optimisations become apparent. Section 5 already contains one instance of this, simplifying a conjugation appearing in a controlled block.

Additionally, a study of more comprehensive equational systems on our language could extend the semantic equivalences between terms presented in Section 6. This could take the form of a complete representation of equality between quantum programs (as has been done for quantum circuits [14]), or aim to characterise an equality relation which may not be complete (with respect to the semantics in unitary matrices) but could exhibit an efficient normalisation function, with potential applications for optimisation.

Finally, the language presented here describes programs which are purely quantum, with no classical components, omitting even allocation and measurement. A more fully-featured quantum programming language based on the “if let” construction requires the addition of these operations. In contrast to naively extending the language, these operations can reuse parts of the pattern infrastructure, with measurement bases specified by patterns, and patterns specified by subprograms with allocation but without measurement.

## Acknowledgements

This research was funded by the Engineering and Physical Sciences Research Council (EPSRC) under project EP/X025551/1 “Rubber DUQ: Flexible Dynamic Universal Quantum programming”. Christopher McNally acknowledges the support of the CQE-LPS Doc Bedard Fellowship. This research was funded in part by the U.S. Army Research Office under Award No. W911NF-23-1-0045. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the U.S. Government.

## References

- [1] L. M. Ahearn. 2012. *Living language: an introduction to linguistic anthropology*. Wiley. doi:10.1002/9781444340563
- [2] T. Altenkirch, P. Dybjer, M. Hofmann, and P. Scott. 2001. Normalization by evaluation for typed lambda calculus with coproducts. In *Proceedings 16th Annual IEEE Symposium on Logic in Computer Science*. 303–310. doi:10.1109/LICS.2001.932506
- [3] T. Altenkirch and J. Grattage. 2005. A Functional Quantum Programming Language. In *Logic in Computer Science*. IEEE, 249–258. doi:10.1109/LICS.2005.1
- [4] P. Andres-Martinez and C. Heunen. 2022. Weakly measured while loops: peeking at quantum states. *Quantum Science and Technology* 7 (2022), 025007. doi:10.1088/2058-9565/ac47f1
- [5] B. Bichsel, M. Baader, T. Gehr, and M. Vechev. 2020. Silq: A high-level quantum language with safe uncomputation and intuitive semantics. In *Programming Language Design and Implementation*. ACM, 286–300. doi:10.1145/3385412.3386007
- [6] A. Bisio, M. Dall’Arno, and P. Perinotti. 2016. Quantum conditional operations. *Physical Review A* 94 (2016), 022340. doi:10.1103/PhysRevA.94.022340
- [7] N. Botö and F. Forslund. 2023. The zeta calculus. (2023). doi:10.48550/arXiv.2303.17399 arXiv:2303.17399.
- [8] K. R. Brown, A. W. Harrow, and I. L. Chuang. 2004. Arbitrarily accurate composite pulse sequences. *Physical Review A—Atomic, Molecular, and Optical Physics* 70, 5 (2004), 052318. doi:10.1103/PhysRevA.70.052318
- [9] C. Bădescu and P. Panangaden. 2015. Quantum alternation: prospects and problems. In *Quantum Physics and Logic (Electronic Proceedings in Theoretical Computer Science, Vol. 195)*. 33–42. doi:10.4204/EPTCS.195.3
- [10] J. Crette, C. Heunen, R. Kaarsgaard, and A. Sabry. 2024. With a few square roots, quantum computing is as easy as Pi. In *ACM Principles of Programming Languages*, Vol. 8. 546–574. doi:10.1145/3632861
- [11] K. Chardonnet. 2023. *Towards a Curry-Howard Correspondence for Quantum Computation*. Ph. D. Dissertation. Université Paris-Saclay. <https://theses.hal.science/tel-03959403>
- [12] K. Chardonnet, L. Lemonnier, and B. Valiron. 2024. Semantics for a Turing-Complete Reversible Programming Language with Inductive Types. In *Formal Structures for Computation and Deduction (FSCD 2024) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 299)*. 19:1–19:19. doi:10.4230/LIPIcs.FSCD.2024.19
- [13] G. Chiribella, G. M. D’Ariano, P. Perinotti, and B. Valiron. 2013. Quantum computations without definite causal structure. *Physical Review A* 88 (2013), 022318. doi:10.1103/PhysRevA.88.022318
- [14] A. Clément, N. Heurtel, S. Mansfield, S. Perdrix, and B. Valiron. 2023. A Complete Equational Theory for Quantum Circuits. In *2023 38th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. 1–13. doi:10.1109/LICS56636.2023.10175801
- [15] B. Coecke and R. Duncan. 2008. Interacting Quantum Observables. In *International Colloquium on Automata, Languages and Programming (ICALP) (Lecture Notes in Computer Science, Vol. 5126)*. Springer, 298–310. doi:10.1007/978-3-540-70583-3\_25
- [16] McKinsey & Company. 2024. Quantum Technology Monitor. Digital. <https://www.mckinsey.com/~/media/mckinsey/business%20functions/mckinsey%20digital/our%20insights/steady%20progress%20in%20approaching%20the%20quantum%20advantage/quantum-technology-monitor-april-2024.pdf>
- [17] A. Cross, A. Javadi-Abhari, T. Alexander, N. De Beaudrap, L. S. Bishop, S. Heide, C. A. Ryan, P. Sivarajah, J. Smolin, and J. M. Gambetta. 2022. OpenQASM 3: A broader and deeper quantum assembly language. *ACM Transactions on Quantum Computing* 3, 3 (2022), 1–50. doi:10.1145/3505636
- [18] K. Dave, L. Lemonnier, R. Péchoux, and V. Zamdzhiev. 2025. Combining quantum and classical control: syntax, semantics and adequacy. In *Foundations of Software Science and Computation Structures*. Springer, 155–175. doi:10.1007/978-3-031-90897-2\_8
- [19] N. de Beaudrap, A. Kissinger, and J. van de Wetering. 2022. Circuit Extraction for ZX-Diagrams Can Be #P-Hard. In *International Colloquium on Automata, Languages, and Programming (ICALP) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 229)*. 119:1–119:19. doi:10.4230/LIPIcs.ICALP.2022.119

- [20] M. Di Meglio. 2023. Pre-Hilbert \*-categories: the Hilbert-space analogue of abelian categories. (2023). doi:10.48550/arXiv.2312.02883 arXiv:2312.02883.
- [21] A. Diaz-Caro. 2025. Towards a Computational Quantum Logic. In *Crossroads of Computability and Logic: Insights, Inspirations, and Innovations*. Springer, 34–46. doi:10.1007/978-3-031-95908-0\_3
- [22] C. Figgatt, D. Maslov, K. A. Landsman, N. M. Linke, S. Debnath, and C. Monroe. 2017. Complete 3-qubit Grover search on a programmable quantum computer. *Nature Communications* 8 (2017), 1918. doi:10.1038/s41467-017-01904-7
- [23] P. Fu, K. Kishida, N. J. Ross, and P. Selinger. 2023. Proto-Quipper with Dynamic Lifting. *Proc. ACM Program. Lang.* 7, POPL, Article 11 (Jan. 2023), 26 pages. doi:10.1145/3571204
- [24] P. Fu, K. Kishida, N. J. Ross, and P. Selinger. 2024. Proto-Quipper with Reversing and Control. arXiv:2410.22261 [cs] doi:10.48550/arXiv.2410.22261
- [25] L. K. Grover. 1996. A fast quantum mechanical algorithm for database search. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing* (Philadelphia, Pennsylvania, USA) (STOC '96). 212–219. doi:10.1145/237814.237866
- [26] B. Heim, M. Soeken, S. Marshall, C. Granade, M. Roetteler, A. Geller, M. Troyer, and K. Svore. 2020. Quantum programming languages. *Nature Reviews Physics* 2 (2020), 709–722. doi:10.1038/s42254-020-00245-7
- [27] C. Heunen and R. Kaarsgaard. 2022. Quantum information effects. *Proc. ACM Program. Lang.* 6, POPL, Article 2 (Jan. 2022), 27 pages. doi:10.1145/3498663
- [28] C. Heunen and A. Kornell. 2022. Axioms for the category of Hilbert spaces. *Proceedings of the National Academy of Sciences* 119, 9 (2022), e2117024119. doi:10.1073/pnas.2117024119
- [29] C. Heunen, A. Kornell, and N. van der Schaaf. 2024. Axioms for the category of Hilbert spaces and linear contractions. *Bulletin of the London Mathematical Society* 56, 4 (2024), 1532–1549. doi:10.1112/blms.13010
- [30] C. Heunen and J. Vicary. 2019. *Categories for Quantum Theory: an introduction*. Oxford University Press. doi:10.1093/oso/9780198739623.001.0001
- [31] K. Hirata and C. Heunen. 2025. Qurts: Automatic Quantum Uncomputation by Affine Types with Lifetime. In *Proceedings of the ACM on Programming Languages*, Vol. 9. 155–182. doi:10.1145/3704842
- [32] H. H. Hoos. 2012. Programming by optimization. *Commun. ACM* 55, 2 (2012), 70–80. doi:10.1145/2076450.2076469
- [33] K. Huang and J. Palsberg. 2024. Compiling Conditional Quantum Gates without Using Helper Qubits. *Proc. ACM Program. Lang.* 8, PLDI, Article 206 (June 2024), 22 pages. doi:10.1145/3656436
- [34] R. Kaarsgaard, H. B. Axelsen, and R. Glück. 2017. Join inverse categories and reversible recursion. *J. Log. Algebraic Methods Program.* 87 (2017), 33–50. doi:10.1016/j.jlamp.2016.08.003
- [35] R. Kaarsgaard and M. Rennela. 2021. Join Inverse Rig Categories for Reversible Functional Programming, and Beyond. In *Mathematical Foundations of Programming Semantics (MFPS) (Electronic Proceedings in Theoretical Computer Science, Vol. 351)*. 152–167. doi:10.4204/EPTCS.351.10
- [36] T. Khattar and C. Gidney. 2025. Rise of conditionally clean ancillae for efficient quantum circuit constructions. *Quantum* 9 (May 2025), 1752. doi:10.22331/q-2025-05-21-1752
- [37] S. Klabnik and C. Nichols. 2023. *The Rust programming language*. No Starch Press.
- [38] M. L. Laplaza. 1972. Coherence for distributivity. In *Coherence in Categories*. Springer Berlin Heidelberg, Berlin, Heidelberg, 29–65. doi:10.1007/BFb0059555
- [39] L. Lemonnier. 2024. *The Semantics of Effects: Centrality, Quantum Control and Reversible Recursion*. Ph.D. Dissertation. Université Paris-Saclay. https://theses.hal.science/tel-04625771
- [40] S. Lloyd. 1996. Universal quantum simulators. *Science* 273, 5278 (1996), 1073–1078. doi:10.1126/science.1229163
- [41] G. H. Low and I. L. Chuang. 2017. Optimal Hamiltonian simulation by quantum signal processing. *Physical review letters* 118, 1 (2017), 010501. doi:10.1103/PhysRevLett.118.010501
- [42] G. H. Low and Y. Su. 2024. Quantum eigenvalue processing. In *Foundations of Computer Science (FOCS)*. IEEE, 1051–1062. doi:10.1109/FOCS61266.2024.00070
- [43] G. H. Low, T. J. Yoder, and I. L. Chuang. 2016. Methodology of resonant equiangular composite quantum gates. *Physical Review X* 6, 4 (2016), 041067. doi:10.1103/PhysRevX.6.041067
- [44] J. M. Martyn, Z. M. Rossi, A. K. Tan, and I. L. Chuang. 2021. Grand Unification of Quantum Algorithms. *PRX Quantum* 2 (2021), 040203. Issue 4. doi:10.1103/PRXQuantum.2.040203
- [45] M. Di Meglio, C. Heunen, J.-S. P. Lemay, P. Perrone, and D. Stein. 2025. Dagger categories of relations: the equivalence of dilatory dagger categories and epi-regular independence categories. (2025). doi:10.48550/arXiv.2508.01146 arXiv:2508.01146.
- [46] M. A. Nielsen and I. L. Chuang. 2010. *Quantum Computation and Quantum Information* (10th anniversary edition ed.). Cambridge university press. doi:10.1017/CBO9780511976667
- [47] M. Pagani, P. Selinger, and B. Valiron. 2014. Applying quantitative semantics to higher-order quantum computing. In *Principles of Programming Languages*. ACM, 647–658. doi:10.1145/2535838.2535879

- [48] L. M. Procopio, A. Moqanaki, M. Araújo, F. Costa, I. A. Calafell, E. G. Dowd, D. R. Hamel, L. A. Rozema, C. Brukner, and P. Walther. 2015. Experimental superposition of orders of quantum gates. *Nature Communications* 6, 1 (2015), 7913. doi:10.1038/ncomms8913
- [49] A. Rice. 2025. Phase-rs - rust implementation of quantum phase language. Zenodo. doi:10.5281/zenodo.17467011
- [50] A. Sabry, B. Valiron, and J. K. Vizzotto. 2018. From Symmetric Pattern-Matching to Quantum Control. In *Foundations of Software Science and Computation Structures (Lecture Notes in Computer Science, Vol. 10803)*. Springer, 348–364. doi:10.1007/978-3-319-89366-2\_19
- [51] J. J. Sakurai and J. Napolitano. 2020. *Modern quantum mechanics*. Cambridge University Press.
- [52] P. Selinger. 2004. Towards a semantics for higher-order quantum computation. In *Quantum Physics and Logic*. 127–143.
- [53] K. J. Sullivan, W. G. Griswold, Y. Cai, and B. Hallen. 2001. The structure and value of modularity in software design. *ACM SIGSOFT Software Engineering* 26, 5 (2001), 99–108. doi:10.1145/503271.503224
- [54] K. Svore, A. Geller, M. Troyer, J. Azariah, C. Granade, B. Heim, V. Kliuchnikov, M. Mykhailova, A. Paz, and M. Roetteler. 2018. Q#: Enabling Scalable Quantum Computing and Development with a High-level DSL. In *Proceedings of the Real World Domain Specific Languages Workshop*. Article 7, 10 pages. doi:10.1145/3183895.3183901
- [55] H. F. Trotter. 1959. On the product of semi-groups of operators. *Proc. Amer. Math. Soc.* 10, 4 (1959), 545–551. doi:10.2307/2033649
- [56] B. Valiron. 2022. Semantics of quantum programming languages: Classical control, quantum control. *Journal of Logical and Algebraic Methods in Programming* 128 (2022), 100790. doi:10.1016/j.jlamp.2022.100790
- [57] B. Valiron. 2024. *On Quantum Programming Languages*. HDR. Université Paris Saclay. <https://theses.hal.science/tel-04740855>
- [58] B. Valiron, N. J. Ross, P. Selinger, D. Scott Alexander, and J. M. Smith. 2015. Programming the quantum future. *Commun. ACM* 58, 8 (2015), 52–61. doi:10.1145/2699415
- [59] F. Voichick, L. Li, R. Rand, and M. Hicks. 2023. Qunity: A Unified Language for Quantum and Classical Computing. *Proc. ACM Program. Lang.* 7, POPL, Article 32 (Jan. 2023), 31 pages. doi:10.1145/3571225
- [60] W. Wootters and W. Zurek. 1982. A single quantum cannot be cloned. *Nature* 299, 5886 (1982), 802–803. doi:10.1038/299802a0
- [61] N. S. Yanofsky and M. Mannucci. 2008. *Quantum Computing for Computer Scientists*. Cambridge University Press.
- [62] M. Ying. 2016. *Foundations of quantum programming*. Morgan Kaufmann. doi:10.1016/C2014-0-02660-3
- [63] M. Ying and Z. Zhang. 2023. Quantum recursive programming with quantum case statements. (2023). doi:10.48550/arXiv.2311.01725 arXiv:2311.01725.
- [64] Z. Zhang and M. Ying. 2024. Quantum Register Machine: Efficient Implementation of Quantum Recursive Programs. In *Proceedings of the ACM on Programming Languages*, Vol. 9. 822–847. doi:10.1145/3729283

Received 2025-07-10; accepted 2025-11-06