Check for updates

Quantum programming languages

Bettina Heim[1,2] ✉, *Mathias Soeken*[1], *Sarah Marshall*[1], *Chris Granade*[1], *Martin Roetteler*[1], *Alan Geller*[1], *Matthias Troyer*[1] *and Krysta Svore*[1]

Abstract | Quantum programming languages are essential to translate ideas into instructions that can be executed by a quantum computer. Not only are they crucial to the programming of quantum computers at scale but also they can facilitate the discovery and development of quantum algorithms even before hardware exists that is capable of executing them. Quantum programming languages are used for controlling existing physical devices, for estimating the execution costs of quantum algorithms on future devices, for teaching quantum computing concepts, or for verifying quantum algorithms and their implementations. They are used by newcomers and seasoned practitioners, researchers and developers working on the next ground-breaking discovery or applying known concepts to real-world problems. This variety in purpose and target audiences is reflected in the design and ecosystem of the existing quantum programming languages, depending on which factors a language prioritizes. In this Review, we highlight important aspects of quantum programming and how it differs from conventional programming. We overview a selection of several state-of-the-art quantum programming languages, highlight their salient features, and provide code samples for each of the languages and Docker files to facilitate installation of the software packages.

As the effort to produce robust, enterprise-scale quantum computing hardware advances[1,2], and as new quantum algorithms are being developed[3–5], the quantum computing community is faced with challenging tasks. To realize the potential offered by quantum computing, it is necessary to program and control quantum devices such that they execute algorithms to solve practical problems[6,7]. Making programming efficient and seamless to the developer requires automating tasks such as circuit layout and gate synthesis[8,9] that are currently often done manually on a case-by-case basis. Classical and quantum computations are often interspersed in quantum applications[10–12] and the inherently asynchronous nature of quantum subroutines can make the task of coordinating between the two a challenging endeavour. Providing the required information for automation and execution requires the development of suitable programming languages and compilation procedures. Tackling real-world problems of interest requires combining different quantum subroutines and algorithms into an application that solves the problem end-to-end. The transition from developing a toolset of quantum algorithms and subroutines to enabling the construction of applications is predicated on the availability of suitable programming languages and frameworks.

For the design of quantum applications, the choice of programming language has implications far beyond what tools are available for expressing a quantum program. More than a matter of convenience, language design choices can have a substantial impact on the costs of developing, running and maintaining an application. To illustrate the consequences and evolution of language design choices over time, consider, for example, the introduction of the `null` pointer in classical languages. It has been criticized for entailing the need for null checks throughout the codebase, thus increasing engineering costs[13]. The ever-ongoing effort to improve the usability and productivity of popular classical languages has hence prompted the adoption of features to revise some of detrimental consequences in languages as disparate as C#, Python, Rust and JavaScript.

Constructs that are available within a language determines how we 'think' and 'reason' about the implemented algorithms. Therefore, the choice of programming language affects how we conceptualize a problem[14], ideally promoting expressing programs in a way that facilitates an efficient execution, rather than encouraging unnecessarily convoluted implementations that are hard to optimize. This is especially important in quantum programming, where even the largest of devices in the medium term will be heavily constrained in terms of logical qubit count and coherence times, and may place heavy demands on timescales for classical feedback. At the same time, some of the conceptual algorithmic techniques, such as amplitude amplification or phase kickback, are ideally supported by the language, so that the

[1]Microsoft Quantum, Redmond, WA, USA.
[2]Theoretische Physik ETH Zürich, Zurich, Switzerland.
✉e-mail: Bettina.Heim@microsoft.com

https://doi.org/10.1038/s42254-020-00245-7

Key points

- Quantum computing fundamentally differs from other means of computing, requiring the development of domain-specific programming languages and compilation techniques.
- Quantum programming languages today cater to a variety of purposes and target audiences ranging from newcomers to seasoned practitioners, and which factors are prioritized substantially impacts the design of the language, the ecosystem and the community around it.
- Programming languages and software tools facilitate the discovery, advancement and development of quantum applications by enabling the verification, resources estimation, program analysis and visualization of quantum applications; they are essential for understanding and analysing both large-scale applications and algorithms for near-term hardware and suitable circuit compilation methods.
- The requirements for software tools for near-term devices and applications differ substantially from those geared towards scalable, fault-tolerant quantum computing.
- The field of quantum programming languages and compilers is still nascent, and current research and implementation efforts are focused on low-level circuit optimizations, concentrating on the purely quantum pieces rather than on optimizations that act across the entire program structure.
- Emulating and accelerating a similar progression to that followed by conventional computing over the past 50 years requires a collaborative effort across disciplines to take full advantage of the acquired knowledge.

programmer can think in terms of the techniques rather than the underlying elementary quantum operations. This in turn can substantially expand the range of applications for quantum computing. The design of quantum programming languages thus must not only draw from the wealth of experience we have in designing classical languages but also address the unique challenges posed in the quantum domain.

In this Review, we overview some of the quantum programming languages that are currently available, exploring different purposes, use cases and the assets of each language. Before talking about different approaches and about the capabilities of each language and supporting tools and ecosystems, it is helpful to expand on the fundamental question regarding how should we reason about quantum algorithms. We thus first proceed to discuss some commonly used patterns in quantum programs, and illustrate some concepts and challenges that are unique to quantum computing.

Basic elements of quantum computing

At the lowest level, quantum algorithms are built from a handful of primitive quantum instructions, just like classical algorithms at the lowest level are made of primitives such as AND, NOT and OR gates. In this section, we briefly introduce the most basic quantum primitives, before elaborating on how this historic point of view relates to expressing quantum programs at a similar abstraction level as we are used to in classical software engineering.

From the perspective of quantum algorithms research, how we think and reason about quantum algorithms is rooted in the mathematical concepts that are at the foundation of quantum computing. An idealized quantum system consisting of n logical qubits used for computation is described by a complex state vector in $\mathbb{C}^{2^n}$. For instance, the state of a single-qubit system can be written as $|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle$, for some complex numbers α and β, and where $|0\rangle := \begin{pmatrix}1\\0\end{pmatrix}$ and $|1\rangle := \begin{pmatrix}0\\1\end{pmatrix}$ are vectors forming the computational basis for a single qubit. The states of a multiqubit system consisting of unentangled subsystems can be described in terms of the tensor products of valid states for individual subsystems. For instance, a two-qubit state can be constructed as $|01\rangle = |0\rangle \otimes |1\rangle = (0100)^\dagger$, where † denotes the conjugate transpose.

The set of valid states of the entire computation space is then found by the closure under linear combinations, subject to the consistency requirement that any valid quantum state must be normalized as $|\langle\psi|\psi\rangle|^2 = 1$. The two-qubit state $(|00\rangle + |11\rangle)/\sqrt{2}$ is therefore a valid state, despite the fact that there does not exist any pair of single-qubit states that can be assigned to each individual qubit. That certain states are impossible to express as a tensor product of the states of individual subsystems implies a particular form of correlation between these subsystems known as entanglement.

Within the above model, a computation corresponds to a sequence of mathematical transformations applied to the state vector. These transformations are described by $2^n \times 2^n$ complex unitary matrices or measurements. A unitary matrix is a matrix whose inverse is given by its conjugate transpose, also referred to as its adjoint. For example, a single-qubit state can be transformed from $|0\rangle$ to $|1\rangle$ and back via the X operator, represented by the unitary matrix

$$X := \begin{pmatrix}0 & 1\\ 1 & 0\end{pmatrix}. \tag{1}$$

That transformation applied to a particular qubit within a multiqubit system is obtained by taking the tensor product of X with the identity operator $\mathbf{1}$ on the remaining qubits. The effect of applying X to the second qubit within a three-qubit system, for example, is represented by the unitary operator $\mathbf{1} \otimes X \otimes \mathbf{1}$. We adopt the convention of specifying transformations by their actions on the transformed subsystem and will leave the extension to the entire system implicit. The X operator is one of three Pauli operators that together with the identity form a basis for all unitary transformations on a single qubit. The matrix representations for the other two operators Y and Z are

$$Y := \begin{pmatrix}0 & -i\\ i & 0\end{pmatrix}, \quad Z := \begin{pmatrix}1 & 0\\ 0 & -1\end{pmatrix}. \tag{2}$$

Any single-qubit unitary transformation can be expressed as a linear combination of these operators. To transition from this mathematical perspective to a formulation that is better aligned with how computer programs are commonly expressed, it is convenient to introduce the single-qubit operators H and T given by

$$H := \frac{1}{\sqrt{2}}\begin{pmatrix}1 & 1\\ 1 & -1\end{pmatrix}, \quad \text{and} \quad T := \begin{pmatrix}1 & 0\\ 0 & e^{i\pi/4}\end{pmatrix}. \tag{3}$$

The Hadamard operator H is the transformation between the eigenbases of X and Z, whereas T rotates a qubit around the Z axis by an angle of π/8. Its square

$S := T^2$ is the transformation between the X and Y eigenbases. To obtain a finite set of discrete instructions that are universal for arbitrary multiqubit unitary transformations only one additional instruction is needed, namely the CNOT operator acting on two qubits. This operator is represented by the unitary matrix

$$\mathrm{CNOT} := \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}, \tag{4}$$

and is often referred to as the controlled-NOT, or CX, gate. Conditioned on the state of the first qubit, the transformation X is applied to the second qubit. Such a conditional transformation in a sense can be seen as the 'quantum version' of a conditional branching, where both branches can execute simultaneously if the condition given by the state of the first qubit being $|1\rangle$ is a superposition of being satisfied and not satisfied[15]. The concept of this coherent branching can be extended to performing arbitrary multiqubit transformations conditioned on the state of multiple qubits. Just like any unitary transformation, such controlled transformations can be broken down into a sequence of simpler one- and two-qubit transformations[16,17].

T, H and CNOT form a universal gate set, that is, any target unitary transformation can be approximated to arbitrary precision over this gate set. The corresponding gate set is often called the Clifford + T gate set. Examples for which such approximations are needed include rotations around the Pauli axes and are common in many quantum algorithms. The corresponding operator is $R_P(\theta) = \mathrm{e}^{-i\theta P/2}$, where $P \in \{X, Y, Z\}$ denotes one of the Pauli matrices and $\theta \in [0, 2\pi)$ is a rotation angle. For further reading about approximations over the Clifford + T and other gate sets, we refer to REFS[9,18,19] and the references cited therein. Besides efficient algorithms for approximation of gates, the Clifford + T gate set has also several benefits when it comes to circuit optimization and rewriting (see REF.[20]).

When implemented fault tolerantly (meaning implemented on top of an error correcting code), gates can have widely different cost in terms of the resources required. In typical fault-tolerant architectures, T gates have to be created by a special distillation process[21]. This process can create large overheads, depending on the underlying noise level and other architectural parameters. In the Clifford + T gate set, the cost of a T gate that is several orders of magnitude higher than the cost of H, Z, X, S and CNOT, which are all elements of the so-called Clifford group. This motivates to just count the total number of T gates required by a quantum algorithm implementation and the total circuit T depth when trying to parallelize as many Ts into stages as possible.

Although large parts of a quantum algorithm can be described by unitary transformations, ultimately classical information needs to be extracted for use by non-quantum hardware. Such an extraction is achieved by projecting the state onto the eigenspaces of a unitary operator, which yields the corresponding eigenvalue as the measurement result. In general, only a limited set of measurements are available as hardware instructions, and any other projection is achieved by suitable transformations beforehand. For example, to project a single-qubit state onto the eigenstates of the X operator, one should apply the basis transformation H, perform the measurement with respect to the computational basis and reverse the basis switch by applying H again. Similarly, to project onto the eigenstates of the Y operator, the basis transformation is given by $(SH)^\dagger = HS^\dagger$, with the reversion SH. Despite being non-unitary, projective measurements can be used to manipulate quantum entanglement[22] and are an integral part of a wide class of quantum algorithms. A simple CNOT gate, for example, can be performed by leveraging entanglement, measurements and single-qubit operations applied conditionally on the measurement outcomes[23].

A quantum program can be seen as a sequence of primitive instructions, such as the elements of the Clifford + T gate set. The sequence is herein generated by a classical algorithm, where the generating algorithm contains classical control flow that potentially depends probabilistically on the execution of preceding transformations. Such dependencies arise if the program continuation is conditioned on the outcome of measurement results, and are widely used in particular in the form of repeat-until-success patterns[24–26] and in iterative phase-estimation-based algorithms[27–29] used in applications such as Hamiltonian simulation[30].

Quantum programming concepts

There are several aspects that are common to both conventional and quantum programs, and correspondingly several areas in language design and compiler heuristics where the quantum computing community can profit immensely from the experience gained from conventional computing over the past decades. There are also unique concepts and possibilities that have no classical analogue and are germane to quantum computing. We review some of these domain-specific concepts below.

Commuting operations. When determining dependencies between quantum instructions, one should ideally account for the fact that even if two instructions act on the same block of quantum memory, the order in which they are applied does not matter as long as they commute. Even if two quantum instructions do not commute in the mathematical sense, it is possible that the error accumulated due to the non-zero commutator merely constitutes a trade-off rather than a decrease in the overall quality of the computation. Although determining bounds on accumulated errors during the computation is certainly within the domain of quantum algorithms research, it stands to reason that for practical purposes, heuristics for determining the required precision with which each program piece needs to be executed might perform reasonably well[31]. One might therefore wonder whether carefully chosen language constructs to express these properties could aid in developing optimizations that exploit such quantum-specific phenomenon.

Clean and borrowed qubits. A particular use of quantum memory is referred to as 'borrowed qubits': qubits that are already in use and may be entangled with other parts of the quantum computer, but are simultaneously used for intermediate computations[16,32]. A quantum subroutine that makes use of such qubits must guarantee that, after its completion, the borrowed qubits are in the same state as before its execution. Even for 'clean qubits', which refers to qubits that are allocated and returned in a known state, the process of releasing allocated quantum memory is not as simple as overwriting it with new content. Although it is physically possible to reset individual qubits by measuring their state, quantum algorithms frequently harness quantum entanglement for computations. In an entangled state, observing part of the state has an immediate impact on the state of the remaining system. Due to this particular quantum correlation, releasing allocated memory generally requires disentangling the corresponding qubits from the remaining system. As the entangling computation corresponds to a unitary transformation, applying its adjoint has the desired effect.

Adjoint operations. A common pattern in quantum programming is hence the need to apply such an adjoint transformation to reverse the effects of a computation. For optimization purposes, it is useful to preserve the association between a transformation and its adjoint, that is, the transformation that reverses its effects. That information can be used, for example, to determine an execution order that reduces computation costs by omitting transformations that are immediately followed by their adjoint as both their effects cancel. Although an adjoint can be defined for any unitary transformation, this is usually not the case for any transformation involving measurements. Even though the effects of a measurement involving several qubits cannot easily be undone, it is possible to combine different measurements in a way that the overall transformation of the state is unitary[33]. It is barely possible to detect such a pattern automatically short of performing a complete symbolic evaluation of the program, and suitable abstractions and language tools are necessary for exploitation in this case[34]. As at a physical level measurements are one of the most expensive instructions, reducing the number of measurements by detecting and optimizing such patterns can save on precious coherence time budgets.

Controlled operations. Another frequently occurring and potentially computationally intensive pattern is the use of controlled unitary transformations. Executing a controlled transformation means that the transformation is applied conditional on the control qubits being in a certain state. They are, so to speak, the quantum version of an if statement, where both branches can be executed simultaneously if the control qubits are in a superposition. For larger transformations, the controlled version is usually composed incrementally, based on the controlled versions of the contained subtransformations. Reordering of parts may allow for building a more favourable composition. Preserving the information about the occurrences of such patterns as abstractions in a programming language makes code more understandable and maintainable, allows for automation and may facilitate optimizations on future large-scale processors. Similar considerations may hold for other constructs, such as applying the same transformation multiple times.

Quantum algorithms by design are built by combining and adapting elementary concepts such as the ones mentioned above. Language design for quantum programs should therefore address the common building blocks of interest for automation and optimization purposes. One might argue that it is sufficient to consider intrinsic hardware instructions as common building blocks, and that answering the question of how to best combine those should remain within the field of quantum algorithms research. This perspective is particularly tempting considering that to optimize approximation errors, the same mathematical transformation of a quantum state may need to be translated into different hardware instruction depending on its context. However, this case-by-case approach for each application and targeted hardware seems like a rather daunting task even if it is to be expected that the range of quantum applications will probably be more narrow than in conventional computing.

One may therefore wonder whether reasoning about more abstract concepts would allow for a substantial advantage in scalability both in terms of program size and development effort. Looking at ubiquitous quantum subroutines, amplitude amplification[35], phase estimation[36] and the quantum Fourier transformation[37] come to mind. The same functionality can often be executed in different ways that are particularly performant with regards to memory requirements, runtime or robustness to errors, for example. Moving from the primitive instructions defined in the previous section to higher-level concepts in terms of which a quantum program is formulated, permits to dynamically adapt the executed instructions depending on the context in which a subroutine is invoked.

The above observations lead to a more general principle: designing a quantum programming language that captures sufficient information to optimize at different levels of abstraction can open the gate to harnessing a wealth of knowledge accumulated on quantum algorithms well before large-scale hardware becomes available. In combination with suitable tools to evaluate various performance metrics, this facilitates and fast tracks the development of compiler heuristics that will extend the range of applications that can be run using a given amount of resources.

Quantum programming

Quantum computing is still in its infancy. Quantum hardware today has relatively few qubits compared with the number of logical qubits needed to execute even the simulation of a small molecule that can't easily be done on classical hardware. Current noise levels would make such a computation infeasible in the sense that the executed computation would probably be incorrect or imprecise. The quantum computing community is tiny compared with the broad computing and programming community. There are a small number of algorithms[38] that are relatively well understood, and an even smaller number of practical applications.

In many ways, today's environment is similar to the state of classical computing in the early 1950s: each system is different, and capabilities and resources are highly limited. Handcrafting a thoroughly optimized implementation for a particular problem instance and hardware target can yield better results than automatically optimized user code containing more abstractions. The latter requires that the language and compiler expose a high degree of control over how code is translated into hardware instruction. Conversely, code that is written specifically for a particular hardware target is less reusable. Enabling code reuse may be less of a priority early on, considering that the current coherence times limit the size of programs. Languages in the current early era thus are often at the same level as the machine and assembler code that was the norm in classical computing before the introduction of higher-level languages in the mid-1950s.

In the future, we expect quantum systems to grow in size and functionality. In the coming decades, systems will contain millions of physical qubits and will use error correction and related techniques to present thousands of logical qubits that are sufficiently error free to make very large computations feasible. As the hardware grows, so too will the community and the algorithm corpus. To be useful in the future of large-scale quantum computing, quantum programming languages will need to evolve just as classical programming languages did from the 1950s to the present day.

In this section, we illustrate different use cases of today's quantum programming languages and how they are typically supported by tools such as simulators and resource estimators as well as ecosystems such as libraries, documentation and learning materials. In the next section, we then discuss different quantum programming languages and explain how they relate to these principles.

Use cases. Given the steady evolution of quantum hardware, one of the use cases of quantum programming languages is to express programs intended for execution on a variety of different quantum computing devices, ideally independent of the underlying technology, such as superconducting qubits[39,40], quantum dots[41], ion traps[42] and topological qubits[43]. As a result of this variety of platforms and goals, there is similarly a difference in the applications targeted by the different hardware efforts. For instance, some efforts are focused more on short-term applications, whereas other efforts are focused on building fully scalable solutions.

Near-term applications, often termed noisy intermediate-scale quantum computing[44] (NISQ), can typically not afford quantum error correction. As a result, optimizations favour single-qubit operations over two-qubit operations and do not distinguish between Clifford and non-Clifford operations. As an example, a T gate has the same cost as a Z gate in stark contrast to error-corrected devices that distil T gates to increase their accuracy[21], causing them to be much more expensive. Further, the resources in near-term quantum computers are severely limited, that is, only a few qubits are available, and the noise due to the lack of error correction limits the number of gates that can be executed before measurement results are essentially random. Every gate and every qubit counts, and quantum programs targeting this level are therefore expressed in terms of explicit gate-level operations, as a higher level of abstraction may lead to a unaffordable cost overhead.

In contrast to the NISQ regime, scalable applications assume fault-tolerant quantum computers with a large number of qubits. Owing to the requirement of magic-state distillation[21] for non-Clifford operations, the set of basic operations is typically limited to just a few Clifford gates and one or two non-Clifford gates. The large number of qubits makes it infeasible to address the quantum computer at the gate level and therefore enable higher levels of abstraction. These are in particular useful, as quantum programs targeting fault-tolerant quantum computers are expressed in terms of logical qubits, which requires a further (automated) compilation step to the actual physical operations and interaction with physical qubits on the quantum computer.

Classical feedback while the quantum state remains coherent can be considered a critical requirement for a scalable quantum computing platform, but is challenging to implement on experimental platforms. We therefore currently see a dichotomy between languages such as PyQuil/Quil[45] and Q#[46], which both include classical feedback as fundamental primitive, and languages such as the Open Quantum Assembly Language (OpenQASM)[47,48] and Cirq[49], which include limited or no classical feedback.

Tools. Having tools that facilitate the discovery and advancement of quantum algorithms even before hardware exists that is capable of executing them is paramount for the success of quantum computing. Beyond enabling execution of quantum programs, quantum programming languages frequently serve as tools for algorithm development, verification and debugging. For that purpose, being able to analyse or simulate a quantum program on classical hardware is often far more convenient and informative than the actual execution on quantum hardware would be. Programming languages primarily geared towards enabling the discovery and analysis of new quantum algorithms hence cater to a very different set of requirements than languages that are primarily intended for running known algorithms on quantum hardware.

Simulation addresses the task of emulating a physical quantum computer on a classical computer. This is a challenging task, particularly when the full state of the quantum computation needs to be captured, which requires memory resources that are exponential in the number of qubits. If one is interested in simulating a computation without accounting for hardware-related errors during execution, the simulation can readily be performed on a standard computer, such as a laptop, for small computations involving up to around 30 qubits. Current supercomputers may be able to simulate up to around 50 qubits[50], or even more if the computation doesn't require that the full state vector is tracked throughout the execution. However, things look much more dire if the goal is to simulate the execution under general noise, described by the Kraus operators, which generally requires tracking not only the state vector but also the full density matrix. Wavefunction Monte Carlo techniques[51] provide a less memory-intensive alternative, at the cost of increasing the computational effort by tracking ensemble trajectories. Simulations are a valuable tool for validating whether the algorithm and implementation works as expected. Depending on the application and quantum algorithm, some simulation tasks can be much simpler. A quantum operation that describes a permutation of the state vector, that is, in reversible operations that describe quantum arithmetic, can be simulated using a Toffoli simulator requiring memory resources that are linear in the number of qubits. Similarly, a CNOT–Hadamard–phase simulator, sometimes also known as a stabilizer simulator, may benefit quantum error correction studies[52].

When targeting a classical simulator that records the full state vector, it may be useful to allow to programmatically access amplitudes for algorithm research purposes. Programs making use of any such capabilities cannot be executed on quantum hardware. Short of allowing user programs full access to post-selection resources, a simulation framework can offer other capabilities to user programs that can be more effectively stripped out when targeting hardware. For instance, ProjectQ[53] allows user programs to specify emulation logic, allowing for larger examples such as Shor's algorithm[37] to run efficiently even with modest classical hardware. Similarly, Q# allows user programs to specify unit testing assertions in terms of strong classical simulation resources; these assertions can then be safely removed during execution on targets that do not provide strong classical simulation.

Resource estimation is the task of costing a quantum operation in terms of the number of primitive gate operations. This task is similar to simulation, but requires much fewer memory resources. A simulator for resource estimation simply needs to update corresponding counters when executing the quantum program. Further, as the semantics of the program is irrelevant for resource estimation, program transformations may substantially speed up this process.

Recent analyses have shown the importance of quantum programming languages to cost estimation[54], as much more accurate numbers can be obtained by explicitly counting the instructions invoked by a particular quantum program than by relying on first-principles bounds on or asymptotic analysis of costs. Moreover, cost estimates derived from runtime observation of quantum programs can also incorporate information from an entire runtime stack, including error correction and layout.

Verification addresses the problem of proving the correctness of a quantum program using static analysis, that is, without executing the program, such as with a simulator. One can either verify whether the program behaves correctly according to a specification, or whether two different programs for the same algorithm are semantically equivalent. The advantage of verification over simulation is that it may require fewer memory resources, by performing analytically rather than explicitly representing the computation in terms of the full state vector, and that it can be applied to parameterized quantum programs, for example, in the number of qubits, and therefore is not limited by it. Quantum programming languages for verification require a well-formalized semantics that can be understood by a theorem prover. Most quantum programming languages targeting verification are based on an interactive theorem prover, in which case, the developer is required to prove the correctness in terms of a sequence of proof steps that can be verified by the theorem prover. Automatic theorem provers to verify the correctness of a quantum program exist; however, these are often not scalable due to the high underlying computational complexity of the verification problem. We won't discuss quantum languages for verification in the remainder of this paper, and refer the reader for more on quantum program verification to the literature (see REFS[55–57]).

Ecosystems. A programming language is but one element of a broader ecosystem that enables developers to construct and express concepts and algorithms programmatically. Programming in any language or context can be a challenging task. Much of the skill, art and trade of modern classical programming are obtained from using tools that facilitate development. A developer may rely on features such as autocomplete or signature help to avoid frequent references to documentation, or may rely on pre-existing libraries to avoid re-implementing common logic. To further ease the developer's role and task, appropriate tooling and integration is required for a seamless execution and debugging experience. Therefore, another important aspect of quantum programming is the ecosystem, which includes libraries, development environments, learning materials, and online access to physical and simulated quantum computers through the cloud. Most of these aim to ease the introduction to both quantum computing and quantum programming, as one of the primary difficulties in getting involved with quantum computing research or development can be the working understanding of quantum computing concepts.

Libraries enable high levels of abstraction by providing generic implementations of often-used quantum algorithms such as a Grover search or quantum phase estimation[58], or application specific libraries, such as for quantum chemistry or quantum machine learning. In addition, they allow the creation of larger applications through composition. A thriving community developing such libraries provides a support network that fosters new ideas that combine the knowledge of experts across different areas. Open-source code can also serve as a source for learning material by means of reference implementations for common functionality.

Other learning material includes static documentation[49,59,60] or interactive coding tutorials, for example, by means of Jupyter Notebooks[61] or Katas. Jupyter Notebooks are interesting for learning quantum programming languages, as they enable the visualization of concepts through graphics and plots, which are essential to understand quantum algorithms and their execution results. Another advantage of Jupyter Notebooks is that they can be executed in a browser and therefore relieve

Kraus operators
Operators defined as part of a theorem characterizing the action of completely positive maps.

Toffoli simulator
A simulator capable of simulating the execution of Toffoli gates.

Grover search
Quantum algorithm for searching an unstructured database.

Quantum phase estimation
Quantum algorithm for estimating the eigenphases of an operator.

the user from the requirement to install additional software. Code Katas are an alternative, in which the learner is provided with a sequence of small exercises, each adding just a little more complexity. In the context of quantum programming, it can teach quantum computing concepts by expressing them in a quantum programming language, thereby learning both quantum computing and how to express it in a program. Katas are implemented in a way such that they can be used in a self-assessment setting, where the learner gets immediate feedback on whether the exercise was solved successfully or not. In the latter case, hints may be provided to master the exercise.

Integrated development environments are an important tool for programmers. Standard features include syntax highlighting and code completion, which can simplify the editing of code. Furthermore, warning and error messages are not only useful to debug code but also help in learning a new programming language. Finally, more advanced concepts, such as code actions, can provide hints to the developer and introduce unknown capabilities of the programming language.

Access to physical quantum computers is provided through the cloud via a multitude of providers (see REFS[62–68] to give just a few examples). Developers can access these through online editors, representational state transfer (REST) application programming interfaces (APIs), or APIs integrated in the programming language. The limited number of available quantum computing devices and their early stage of development reinforces the merit of tooling to analyse and verify a quantum program before execution on quantum hardware.

Language deep dives

In this section, we discuss some of the quantum programming languages, along with the tools and ecosystems around them, that are currently in use and/or have served as a foundation for current and future work. We do not aspire to be exhaustive, but instead cover a representative set of languages with distinct priorities and focusing on different paradigms. Our selection serves as illustration for the implications and distinguishing features that follow from prioritizing certain use cases when designing quantum programming languages. These choices include decisions such as prioritizing execution on currently available hardware versus future fault-tolerant devices, enabling extensive program verification or striving to facilitate the creation and use of performant simulation tools on classical computers.

We choose to elaborate on these topic by examples, including the languages Q#, Qiskit, Cirq, Quipper and Scaffold. Our selection is not a judgement on the importance or popularity of individual languages; some widely used languages that are referenced below are not discussed in detail. Instead, we discuss a selection of languages that we hope captures unique facets of interest to the reader. The selected languages demonstrate the difference in design and ecosystem depending on the intended purpose and target audience.

Without diminishing the multifaceted capabilities of individual languages, we chose these five language for the following reasons: Q# is focused on supporting large-scale quantum applications, whereas Cirq is primarily geared towards NISQ devices. OpenQASM has served as an intermediate representation that is supported by a variety of hardware and software providers. Quipper has served as the foundation in the field of quantum programming language research and inspired the development of formal verification techniques, whereas Scaffold with its low-level virtual machine (LLVM)-based compiler ScaffCC has spawned discussions about quantum programming within the classical compiler research community.

A detailed discussion of a broader spectrum of quantum programming languages and open-source software frameworks such as Forest/PyQuil[45,69], ProjectQ[70,71], QWIRE[72,73], staq[74,75], Strawberry Fields[76,77], t|ket⟩[78,79], XACC[80,81] or QuTiP[82–84] was out of scope of this Review. However, many of the strength of the languages that we highlight in this Review also apply to these languages and software frameworks.

A quick overview over some of the discussed features is assembled in TABLE 1. It should be noted that all languages, compilers and tools associated in this selection are open source. For further reading about quantum programming languages, we point the reader to a review article[85], and conference and workshop reports ~~series such as REF. 86~~.

As a simple, yet illustrative, example we show how quantum teleportation (a protocol in which one sends the state of a qubit to another qubit by using pre-shared entanglement and classical communication as resources[87,88]) is expressed in each language. Although teleportation is too simple to fully capture the strengths and weaknesses of each software framework, it nonetheless serves as a first impression to introduce the language. Along with this Review comes the code that implements the sample in each language, along with a collection of Docker[89] files to build the frameworks for each of the languages allowing the reader to build the required software packages from scratch in a robust and reproducible way.

Q#. Q# is a hardware-agnostic quantum programming language developed with the goal to enable execution of large-scale applications on future quantum hardware[46]. Hence, Q# is focused on providing high-level abstractions that facilitate reasoning about the intended functionality rather than following the imperative style encouraged by assembly-like languages. In contrast to, say, Python-based languages, Q# is strongly and statically typed. All types follow value semantics, including arrays[90]. It supports the constructs commonly provided for immutable data types, such as the means to construct a new array based on an existing one[59]. This restriction has the benefit that all side effects of a computation have to be of quantum nature, meaning they can only impact the quantum state; there are no language constructs in Q# that can modify classical values that are accessible outside the operation or function in which they are declared. The only way for side effects to impact the program flow is hence to have a branching based on the result of a measurement outcome. In combination

Side effects
A side effect in programming is an effect that modifies the program state outside the local environment.

Table 1 | **Overview of the languages surveyed in this Review**

Feature	Q#	Qiskit	Cirq	Quipper	Scaffold
Invocation	Standalone, usable from Python, C#, F#	Embedded into Python	Embedded into Python	Embedded into Haskell[a]	Standalone
Classical feedback	Yes	Yes[b]	No	Yes	Yes[c]
Adjoint generation	Yes	Yes	Yes	Yes	No
Resource estimation	Gate counts, number of qubits, depth and width, call graph profiling	Gate counts, number of qubits, depth and width	Gate counts, number of qubits	Gate counts, number of qubits, depth and width	Gate counts, number of qubits, depth[d]
Libraries	Standard, chemistry, numerics, ML	Standard, chemistry, optimization, finance, QCVV, ML	Standard, chemistry, ML	Standard, numerics	Standard[e]
Learning materials	Docs, tutorials, Katas	Docs, tutorials, textbook	Docs, tutorials	Docs[f], tutorials	Tutorials[g]

[a]Standalone versions such as Proto-Quipper-S and Proto-Quipper-M are proposed or under development. [b]Some restrictions apply regarding allowed types and language constructs in OpenQASM branching statements. [c]However, see relevant GitHub issue[122] regarding code generation for classical feedback. [d]Resources estimation includes different flavours of error correction (see REF.[123]). [e]See REF.[121] for the current selection of implemented algorithms. [f]Online API documentation available in REF.[124]. [g]Tutorials and manual in REFS[116,118]. ML, machine learning; QCVV, quantum characterization, verification and validation.

with the fact that measurement outcomes are represented by a dedicated type, this allows the restriction of how local computations can impact the program flow via the type system if needed, for example, to accommodate current hardware limitations.

A salient feature of Q# is that it supports expressing arbitrary classical control flow[91]. This is in contrast to other quantum programming languages where this capability is often provided by a classical host language. The representation within the quantum programming language itself permits developers to reason about the program structure at the application level. This allows the integration of, for instance, computing precision requirements for rotation synthesis[18], or scheduling and layout on the quantum chip for future large-scale applications.

Q# distinguishes between operations and functions. Both are first-class values and can be freely assigned or passed as arguments[46]. Functions are purely classical and deterministic in nature. As a consequence, functions can be fully evaluated as soon as their input is known. Operations, however, may contain arbitrarily interleaved classical and quantum computations, including allocations and deallocations of quantum memory.

Unlike other quantum programming languages geared towards supporting formal verification, qubits are treated like any other data type in Q#. Some languages, such as Proto-Quipper-M[92], use a linear-type system to enforce the no-cloning property of quantum states. By contrast, Q# treats qubits as virtual entities of quantum memory. It thus takes a purely operational perspective, and has no notion of a quantum state within the language itself[93].

In addition to abstractions such as type parameterization and user-defined types that primarily serve the purpose of user convenience and code robustness, Q# defines constructs that facilitate representing and leveraging certain quantum-specific patterns for optimization. Examples for such constructs are borrowing of qubits[16], conjugations representing patterns of the form $UVU^{\dagger}$, where U and V are some unitary transformations (see REF.[59]), and functors[59]. Functors can be seen as higher-order bijective meta-functions that associate quantum transformations that have a certain relation to each other. The adjoint functor, for example, maps a unitary quantum transformation to its inverse. Which set of functors an operation supports is reflected in its type. The captured relations can be exploited for optimization purposes.

Q# is compiled in a standalone manner, making whole program analysis more tractable. Q# offers interoperability with Python and .NET languages such as C# and F#. Its nature as a standalone programming language makes it easier to define a natural representation for quantum programs, as it is not constrained by choices made in the host language. This comes at the cost of not being able to leverage the rich set of existing tools for popular classical languages such as Python. Q# comes with its own set of tools, including, among others, support for Jupyter Notebooks[59,61], and an implementation of the Language Server Protocol[94] for providing semantic information to editors. Microsoft provides two integrated development environment extensions for Q#: Visual Studio Code, which is supported on macOS, Linux and Windows, and Visual Studio for Windows.

A rich set of samples, libraries, tutorials and Katas exist around Q# (see REFS[95–97]). In addition to domain-specific libraries on chemistry, machine learning and quantum arithmetic, the standard libraries offer an arsenal of tools. Each contained callable and type is extensively documented in the code. That information is displayed to the user via integrated development environment tools. The corresponding API documentation is generated for each release, and complements the documentation on the language, tools and quantum computing concepts. Katas and other teaching materials[95], designed to learn about both quantum computing and Q#, facilitate entering the field of quantum computing.

The associated libraries, the Q# compiler and all other components of the quantum development kit are open source. Several NuGet packages are currently distributed, including a package containing tools for simulation and resources estimation. Microsoft has partnered with hardware providers to offer a service for executing Q# code on quantum hardware as part of the cloud-based Azure Quantum service[63]. This comes at the caveat that only a limited subset of Q# programs can be executed on quantum hardware, as current devices do not provide sufficient control flow capabilities to execute everything that can be expressed in Q#. At the time of writing, Honeywell, IonQ and QCI are announced hardware partners[63].

In the Q# teleportation code (see also Listing 1 in Supplementary Information), the state preparation in operation `PrepareBellPair (left : Qubit, right : Qubit) : Unit is Adj + Ctl` returns `@Unit@`, as quantum operations are modelled as side effects. Its adjoint operation as invoked as `Adjoint PrepareBellPair (msg, helper);` is automatically generated by the compiler. The teleportation is executed on a full state simulator as part of a unit test defined via the `@Test@` attribute. Our example is written as unit test. Executing the command dotnet test in the application folder will run the test.

OpenQASM and Qiskit. OpenQASM[48] is a gate-based intermediate representation for quantum programs. It expresses quantum programs as lists of instructions, often intended to be consumed by a quantum processor without further compilation. OpenQASM allows for abstractions in the form of quantum gates, which can be composed in a hierarchical manner based on a set of intrinsic primitives that are assumed to be available on the targeted processor; an example being a Toffoli gate composed of CNOT gates, T gates and H gates. OpenQASM also supports single-qubit measurement and basic classical control operations.

Qiskit[98,99] provides a Python-based programming environment that allows one to generate and manipulate OpenQASM programs. It provides powerful abstraction capabilities, such as the ability to synthesize gate decompositions for arbitrary isometries and certain unitary transformations. The ecosystem of Qiskit comprises four software frameworks:

- Terra: provides fundamental data structures for quantum computing.
- Aer: provides various simulation backends for executing circuits compiled in Qiskit Terra and tools for noise modelling.
- Aqua: provides generalized and customizable quantum algorithms, including domain application support for chemistry, finance, machine learning and optimization.
- Ignis: a framework to understand and mitigate noise in quantum programs using quantum characterization, verification and validation protocols.

Qiskit can be used to access physical quantum computers through cloud services. An online service, called IBM Quantum Experience[62] allows to write quantum programs either through a visual interactive quantum circuit editor or using the Python-based libraries within Jupyter Notebooks. The IBM Quantum Experience also includes the OpenPulse framework for pulse-level control, which allows users to construct their own schedules of pulses and execute them on IBM's quantum computer hardware.

At the time of writing, Qiskit can be used on IBM's quantum computers in addition to devices offered by Alpine Quantum Technologies[66,100] and Honeywell[101]. Layout and routing stages during compilation permit targeting also limited-connectivity architectures without manually tailoring programs to hardware restrictions. The compilation infrastructure furthermore includes a large number of general purpose optimization passes for quantum circuit optimization.

Besides its support for execution on quantum processors, Qiskit comes with extensive simulation capabilities, including state vector and density matrix simulators that can be executed on both CPUs and GPUs. It thus provides support for simulating the effects of noise defined by any custom model including arbitrary Kraus operators. Qiskit furthermore contains an efficient Clifford stabilizer state simulator and a tensor-network state vector simulator that uses a matrix product state representation for the state.

The online documentation[60] gives a good overview over the full spectrum of capabilities included in Qiskit, includes tutorials and is generated for each release. In terms of learning resources, an interactive textbook[102] based on the new Jupyter Book platform[103] teaches beginners quantum computing by means of interactive code examples. These examples can easily be run via direct links to notebooks on the IBM Quantum Experience.

The OpenQASM quantum teleportation code (see also Listing 2 in Supplementary Information) can be compiled using Qiskit Terra, and then dispatched to a simulator or to hardware using Qiskit Aer, as illustrated here (see also Listing 3 in Supplementary Information). Alternatively, the circuit can be constructed completely using the Python API, as shown here (see also Listing 4 in Supplementary Information).

Cirq. Cirq is a quantum programming library for Python with a strong focus on supporting near-term quantum hardware. Cirq's primary goal is to enable the development of quantum programs that are capable of running on quantum computers available now or in the near future without the means of error correction (NISQ hardware), and subject to certain device topologies. Correspondingly, Cirq exposes fine-grained control to the end users. It provides mechanisms for fine-tuning exactly how a quantum program executes on the targeted quantum hardware, and tools for simulating hardware constraints, such as limitations due to noise or the physical layout of the qubits[49].

The types of qubit available to the programmer further demonstrates Cirq's focus on NISQ hardware. The physical layout of qubits in a quantum computer can be modelled by a GridQubit for hardware with a 2D lattice, or a LineQubit for hardware with a 1D lattice[49]. Cirq also provides qubit types that do not impose any physical layout, either for developing quantum programs that are intended only to be simulated, or to use as part of the definition of a custom layout. Once a qubit type is chosen, device constraints can be specified programmatically and Cirq will validate that a particular circuit adheres to all of the constraints[49]. For example, a common hardware constraint is that two-qubit gates may only operate on adjacent qubits.

In contrast to other languages where qubits may be allocated dynamically, layout is performed manually in Cirq. Qubits can only be allocated by providing their position (for example, row and column for a GridQubit) or another globally unique identifier (for example, a string for a NamedQubit). This means the programmers must decide which physical qubits to use for each part of an algorithm, but as a result, they have the most control over how a NISQ computer's limited number of qubits are being used.

Similarly, the programmer has several options when it comes to scheduling each quantum operation. A circuit in Cirq is divided into 'moments', which are discrete units of time in which all operations in the same moment execute simultaneously[49]. Only a single operation can affect a particular qubit at any given moment. When operations are added to the circuit, they can be added as part of a new moment (increasing the total length of time of the program), or instead can 'slide' back to an earlier moment if the affected qubits are not already being used at that time. This has trade-offs similar to those that come from the manual allocation of qubits: it requires the programmer to make more decisions, but gives the most flexibility in how the hardware is being used at every point in time while the program is running.

Circuits in Cirq are defined declaratively as a sequence of moments, where each moment contains a set of gates to apply. As Cirq is embedded in Python, it is easy to manipulate circuits, as they behave similarly to other Python sequences. For example, higher-level quantum operations can be created by defining Python functions that return a sequence of gates that can be appended to a circuit. It is also possible to iterate over, transform or filter the moments in a circuit. Furthermore, the operations in each moment can be inspected or transformed as well.

Cirq is embedded in Python. The control flow constructs provided by Python, such as `if` and `while` test statements, can be used to construct a circuit before executing it. There is no direct support for control flow based on measurement results. When targeting simulators, it is possible to emulate control flow by retrieving the quantum state at the end of a simulation, performing arbitrary classical logic on any measurement results obtained, and initializing a new simulation to start in the previous state. When targeting quantum devices however, a similar scheme is not possible to run quantum algorithms that use non-deterministic circuits — circuits in which different operations are applied depending on measurements. This includes, for example, the class of repeat-until-success algorithms[24–26,104] and iterative phase estimation and potentially other primitives that require non-trivial control flow. This limitation of Cirq is reasonable to have, considering how it mirrors the capabilities of NISQ hardware.

It is also possible to use Cirq to model the effect of noise on a circuit by creating noisy quantum channels defined by Kraus operators[49]. Common channels are included with Cirq, such as channels that introduce bit flip or phase flip errors with probability p.

Libraries for Cirq include OpenFermion-Cirq and TensorFlow Quantum[105]. Both are adaptations of existing libraries to provide interoperability with Cirq. OpenFermion-Cirq is based on OpenFermion, a quantum chemistry library[106]. TensorFlow Quantum[105] is based on TensorFlow[107], a machine learning library, adapted for use with quantum machine learning.

Detailed documentation for Cirq is available online[49], including an installation guide and a tutorial to help get new users familiar with Cirq. The documentation is generated for each release. In-depth sections are also available for more detailed descriptions of Cirq's features, including a complete API reference. As Cirq is a Python framework, it can also leverage the large variety of existing tools available for developing Python applications.

It is possible to run Cirq programs on Google's Quantum Cloud Service, but this requires access, which according to a note in the source code is granted by invitation only[108]. Google plans to grant public access to cloud-based quantum simulators and actual quantum hardware in the future[64]; currently Cirq supports running programs on a local simulator. In terms of third-party offerings, Alpine Quantum Technologies, in collaboration with the University of Innsbruck, supports both Cirq and Qiskit on their ion-trap quantum computers in Innsbruck[66].

Cirq comes with device models for many of Google's quantum processors[49], including Bristlecone and Sycamore. The models can be used to determine whether a circuit is suitable to run on a particular device, validating circuit characteristics such as gate set, dimensions of qudits (qubit, qutrit and so on) and locality of multiqubit operations. Device support can also be extended to custom devices with their own set of constraints. In addition to simply validating whether a circuit conforms to a particular gate set, Cirq can transform circuits into the gate sets of Google devices, but transforming circuits into arbitrary gate sets is not currently supported[49].

We provide a Cirq quantum teleportation code (see also Listing 5 in Supplementary Information). Cirq does not support applying quantum gates conditioned on a classical control bit. Rather than a branching based on classical control bit, the example instead makes use of a controlled Z gate (`CZ`) and a controlled X gate (`CNOT`), thus replacing the classical control bit with a qubit. When run, the program prints both a graphical representation of the circuit and its representation as a unitary matrix, and the measurement results from simulating the circuit. It can be executed directly by the Python interpreter.

Quipper. Quipper[109] is a functional quantum programming language, embedded into the Haskell programming language. It addresses the problem of describing quantum computations at a practical scale, and demonstrated by describing quantum circuit representations with up to trillions of quantum gates. Quipper uses the computation model that uses a classical computer to control a quantum device, and it is not dependent on any particular quantum hardware. In addition to being used to estimate the costs of quantum computations[110], the language provides a foundation for quantum programming language research. Quipper has served as inspiration for leveraging existing tools for formal verification for quantum programs[111,112], and subsequent work[92,113,114] building on the ideas of Quipper has a distinct focus on how to facilitate reasoning about the semantics of quantum programs.

Quipper is a circuit description language, that is, the language can be used in a structured way to construct circuits by applying gates to qubits. The circuits themselves are data that can be passed to functions in the host language Haskell, for example, to perform circuit optimization, resource estimation or error correction. As in many other embedded programming languages, the mismatch between the type system of the host language and the type system of the quantum programming language allows the developer to write programs that are not well defined and lead to runtime errors. In particular, Haskell is not able to enforce linearity, such that type system cannot guarantee that multiple operations do not simultaneously act on the same qubit. Standalone prototypical implementations of Quipper-like languages, such as Proto-Quipper-S[113], Proto-Quipper-M[92] and Proto-Quipper-D[114] have emerged with the goal to enforce quantum-specific properties such as the no-cloning theorem of quantum information. They are based on a linear-type system and take other quantum programming related concepts explicitly into account. For example, Proto-Quipper-M distinguishes between parameters, which are data known at circuit generation time (for example, the bitwidth of an arithmetic operation) and states, which are data known at circuit execution time (for example, a quantum state or measurement result). Whereas Quipper allows to describe families of quantum circuits, Proto-Quipper-M is more general, in that it describes families of morphisms in any symmetric monoidal category[92]. The latest addition, Proto-Quipper-D, introduces linear-dependent types that can be used to express program invariants and constraints, and allows for type-safe uncomputation of garbage qubits[114].

Quipper represents a quantum program as the accumulation of quantum side effects in a circuit monad, called `Circ`. The side effects accumulated in a `Circ` can be run by the use of a function in the `QuipperLib.Simulation` module. Quipper supports high-level circuit combinators, for example, for circuit reversal and iteration, and has the capability to compute gate counts for a particular circuit[115]. Part of the resource estimation framework is the concept of 'boxing' of subroutines, which allows the user to obtain gate counts and other metrics in a scalable way. The monadic framework allows Quipper to leverage patterns of the form $UVU^{\dagger}$ and their controlled application, similar to, for example Q#.

In the Quipper quantum teleportation code (see also Listing 7 in Supplementary Information), we demonstrate how the `teleport` function from Haskell code (see also Listing 6 in Supplementary Information) that instantiates the target machine for the quantum teleportation code can be used with `Quipper.print_simple ASCII`, `QuipperLib.Simulation.run_generic random_number_generator` and other such functions to export or run Quipper programs against simulation resources.

Monad
In programming, a monad is a structure that represents computations.

Scaffold. Scaffold is designed for expressing quantum algorithms in a high-level format that can be compiled into low-level implementations whose properties can be studied[116]. Analysing these properties can help one understand what hardware capabilities are needed to feasibly execute different kinds of quantum algorithms.

Scaffold is a standalone language. It is designed to be similar to existing classical programming languages, particularly C: Scaffold adopts C's imperative programming model and many of its familiar features such as functions (known as modules in Scaffold), `if` statements, loops, structures and preprocessor directives[116]. In addition, Scaffold programs can automatically convert classical functions into reversible logic, implemented using quantum gates, so that they can be embedded as an oracle in a larger quantum algorithm[116,117].

ScaffCC, the Scaffold compiler, can analyse programs by estimating quantum resource usage and critical path length[117], and is designed to scale to trillions of gates. ScaffCC obtains more accurate resource estimates by applying classical and quantum optimizations and configurable quantum gate decompositions to programs. To estimate the critical path length, ScaffCC creates a schedule of the order and timing of quantum operations, and optimizes that schedule using properties of the quantum circuit, such as gate dependencies.

ScaffCC is open source and based on LLVM[117], and integrates with other quantum computing tools. Scaffold supports RevKit for Quantum Computation (RKQC)[118], an adaptation of the reversible circuit design toolkit RevKit[119] for use with quantum circuits. RKQC is used to compile oracles embedded in Scaffold programs. In addition to creating resource estimation reports, ScaffCC supports generating output in three different variants of QASM: 'hierarchical' QASM, 'flattened' QASM and OpenQASM, where hierarchical QASM preserves Scaffold modules, whereas flattened QASM and OpenQASM do not. Lastly, ScaffCC can also generate input files for QX Simulator[118,120], a third-party quantum simulator.

Some learning resources are available for Scaffold. The Scaffold language report describes the initial language's features with code examples[116]. The ScaffCC GitHub repository[121] includes a user guide for installing and using the compiler, and several examples of common quantum algorithms such as quantum Fourier transformation, Shor's algorithm or a variational quantum eigensolver algorithm implemented in Scaffold.

We provide a Scaffold implementation of quantum teleportation code. It can be compiled into QASM by invoking the ScaffCC compiler. It should be noted that although the example compiles without errors or warnings, the generated QASM code is missing the classical control flow (namely, the `if` statements needed for the correction of the teleported qubit). We refer the reader to the corresponding issues on the ScaffCC GitHub repository[122] for further details.

Outlook

Quantum computing promises possibilities, however, several major technical challenges need to be overcome first. The necessary breakthroughs require a collaborative effort across disciplines and institutions. Beyond taking full advantage of the classical computational resources available today to facilitate these developments, the community must leverage the insights into how computing itself has evolved over the course of more than half a century. Computing has grown in ways that would have been impossible to predict back in the early days. In striving to emulate and accelerate a similar progression for quantum computing over the next couple of decades, researchers have the tremendous advantage of having the hindsight into what were the essential success factors that made that growth possible.

The goal of quantum computing is to enable tasks that can't be achieved with other computing means. Accomplishing this entails certain requirements not just on the scale of quantum hardware and applications but also on the scale of the field itself. For example, realizing the full potential of quantum computing requires the pursuit of different ideas and approaches to discover a path that will ultimately lead to success. It also requires contributions to disparate elements from people with diverse backgrounds, perspectives and areas of expertise. A tight collaboration between people working on hardware, firmware, control software, compilers and languages is therefore inevitable. Making quantum computing succeed hence demands more than building a stack for executing quantum programs. The community needs the tools and frameworks to enable individuals to contribute, communicate and share their knowledge effectively.

The purpose of programming languages is to enable such a communication by representing an idea or concept in a way that allows the reader to visually distinguish and identify the key components and how they fit together. Quantum programming languages serve as a facilitator for clearly expressing quantum algorithms and executing them, but also to explore and develop applications and the hardware to support them in the first place. Suitable tools can help to analyse, understand and mitigate noise in quantum programs using quantum characterization, verification and validation protocols, and to develop automatic calibration and tuning protocols for quantum devices. We have seen how quantum programming languages play a critical role for application development by enabling the verification, simulation and visualization of quantum algorithms and programs. The ecosystem and community around them promotes the development of compilation techniques tailored to the unique nature and potential of quantum-specific concepts, and a variety of teaching materials introduce a new generation of researchers to the fundamentals of quantum computing.

The field of quantum programming languages and compilers is still nascent. Quantum compilers are not yet as sophisticated as modern highly optimizing classical compilers. Current research and implementation efforts are focused on low-level circuit optimizations, concentrating on the purely quantum pieces rather than on optimizations that act across the entire program structure. This is in contrast to classical compilers where a lot of optimizations are concerned with higher-level abstractions and control flow constructs such as various kinds of loops. In quantum computing, it is currently often the case that a more general implementation of an algorithm performs worse than a custom implementation for a specific problem. The specific version can rely on characteristics and properties of the one problem that are not valid in general. This same dichotomy exists in classical high-performance computing and has led to decades of compiler optimization research and engineering, and work in language design to give more information to the compiler.

Whereas some well-known classical language patterns have been adopted by several quantum languages, more work needs to be done to understand how the quantum-specific aspects of a general algorithm can be expressed so that they can be optimized well enough to compete with a custom implementation. Today, specialized solutions for each problem instance are often hand-coded and expressed as a sequence of gates that are natively implemented by a targeted processor. More advanced compilation techniques and automatic optimizations will permit to write general solutions to entire classes of problems by building layers of abstraction.

Finally, we would like to offer a few thoughts about the long-term perspectives of quantum programming and the new horizons of this exciting field.

- The growth in hardware scale will make programming individual qubits infeasible. Languages will need to provide developers with abstractions and constructs that allow groups of qubits of variable sizes to be manipulated. In some cases, the sizes may not be known until runtime, so the full end-to-end system needs to be able to represent and manipulate collections of unknown sizes. Manipulating qubit collections will require language constructs such as loops and functional map and fold, familiar from classical programming languages.
- The growth in community scale will drive the need for sharable, reusable code libraries. This will radically improve productivity by allowing developers to build on components they or others have built in the past. Perhaps even more important, as the classical developer community has found, sharable code is incredibly effective as a communications and teaching medium. A well-written and testable implementation is the best specification of an algorithm.
- The growth in the number of algorithms will contribute to the need for libraries. It will also put pressure on languages to provide more general and more useful forms of composition than the purely imperative model. It is reasonable to expect that quantum programming languages will follow the evolution of classical languages to richer and richer composition models, similar to how different paradigms such as object-oriented programming, functional programming and logic programming, as well as the various hybrids, are common today in classical languages.

The field of quantum applications research is ever evolving, and it is not obvious which future applications are likely to ultimately benefit society. Only as more insight is gained will it become more evident what optimizations need to be considered the most impactful ones. To advance the field of quantum programming and computation, all areas discussed in this paper should be explored, and no single software stack is likely to excel in all of them.

Code availability

Examples of code can be found at https://github.com/msr-quarc/quantum-languages-review/tree/master/src.

Published online: 16 November 2020

1. Arute, F. et al. Quantum supremacy using a programmable superconducting processor. *Nature* **574**, 505–510 (2019).
2. Kandala, A. et al. Error mitigation extends the computational reach of a noisy quantum processor. *Nature* **567**, 491–495 (2019).
3. Montanaro, A. Quantum algorithms: an overview. *npj Quantum Inf.* **2**, 15023 (2016).
4. Roetteler, M. & Svore, K. M. Quantum computing: codebreaking and beyond. *IEEE Secur. Priv.* **16**, 22–36 (2018).
5. Montanaro, A. Quantum speedup of branch-and-bound algorithms. *Phys. Rev. Res.* **2**, 013056 (2020).
6. Chong, F. T., Franklin, D. & Martonosi, M. Programming languages and compiler design for realistic quantum hardware. *Nature* **549**, 180–187 (2017).
7. Ross, J. The dawn of quantum programming. *Quantum Views* **2**, 4 (2018).
8. Nam, Y., Ross, N. J., Su, Y., Childs, A. M. & Maslov, D. Automated optimization of large quantum circuits with continuous parameters. *npj Quantum Inf.* **4**, 23 (2018).
9. Kliuchnikov, V., Bocharov, A., Roetteler, M. & Yard, J. A framework for approximating qubit unitaries. Preprint at https://arxiv.org/abs/1510.03888 (2015).
10. Farhi, E., Goldstone, J. & Gutmann, S. A quantum approximate optimization algorithm. Preprint at https://arxiv.org/abs/1411.4028 (2014).
11. Peruzzo, A. et al. A variational eigenvalue solver on a photonic quantum processor. *Nat. Commun.* **5**, 4213 (2014).
12. Moll, N. et al. Quantum optimization using variational algorithms on near-term quantum devices. *Quantum Sci. Technol.* **3**, 030503 (2018).
13. Hoare, T. Null references: the billion dollar mistake. *InfoQ* https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare/ (2009).
14. Iverson, K. E. Notation as a tool of thought. *Commun. ACM* **23**, 444–465 (1980).
15. Ying, M. *Foundations of Quantum Programming* (Morgan Kaufmann, 2016).
16. Barenco, A. et al. Elementary gates for quantum computation. *Phys. Rev. A* **52**, 3457–3467 (1995).
17. Bergholm, V., Vartiainen, J. J., Möttönen, M. & Salomaa, M. M. Quantum circuits with uniformly controlled one-qubit gates. *Phys. Rev. A* **71**, 052330 (2005).
18. Kliuchnikov, V., Maslov, D. & Mosca, M. Fast and efficient exact synthesis of single qubit unitaries generated by Clifford and *T* gates. *Quantum Inf. Comput.* **13**, 607–630 (2013).
19. Ross, N. J. & Selinger, P. Optimal ancilla-free Clifford. *T* approximation of *z*-rotations. *Quantum Inf. Comput.* **16**, 901–953 (2016).
20. Amy, M., Maslov, D., Mosca, M. & Roetteler, M. A meet-in-the-middle algorithm for fast synthesis of depth-optimal quantum circuits. *IEEE Trans. CAD Integr. Circuits Syst.* **32**, 818–830 (2013).
21. Bravyi, S. & Kitaev, A. Universal quantum computation with ideal Clifford gates and noisy ancillas. *Phys. Rev. A* **71**, 022316 (2005).
22. Jozsa, R. An introduction to measurement based quantum computation. Preprint at https://arxiv.org/abs/0508124 (2005).
23. Gottesman, D. & Chuang, I. L. Demonstrating the viability of universal quantum computation using teleportation and single-qubit operations. *Nature* **402**, 390–393 (1999).
24. Bocharov, A., Roetteler, M. & Svore, K. M. Efficient synthesis of universal repeat-until-success quantum circuits. *Phys. Rev. Lett.* **114**, 080502 (2015).
25. Bocharov, A., Roetteler, M. & Svore, K. M. Efficient synthesis of probabilistic quantum circuits with fallback. *Phys. Rev. A* **91**, 052317 (2015).
26. Wiebe, N. & Roetteler, M. Quantum arithmetic and numerical analysis using repeat-until-success circuits. Preprint at https://arxiv.org/abs/1406.2040 (2014).
27. Granade, C., Ferrie, C., Wiebe, N. & Cory, D. Robust online hamiltonian learning. *New J. Phys.* **14**, 103013 (2012).
28. Paesani, S. et al. Experimental Bayesian quantum phase estimation on a silicon photonic chip. *Phys. Rev. Lett.* **118**, 100503 (2017).
29. Wiebe, N. & Granade, C. Efficient Bayesian phase estimation. *Phys. Rev. Lett.* **117**, 010503 (2016).
30. Kivlichan, I. D., Granade, C. E. & Wiebe, N. Phase estimation with randomized Hamiltonians. Preprint at https://arxiv.org/abs/1907.10070 (2019).
31. Meuli, G., Soeken, M., Roetteler, M. & Häner, T. Automatic accuracy management of quantum programs via (near-) symbolic resource estimation. Preprint at https://arxiv.org/abs/2003.08408 (2020).
32. Low, G. H., Kliuchnikov, V. & Schaeffer, L. Trading T-gates for dirty qubits in state preparation and unitary synthesis. Preprint at https://arxiv.org/abs/1812.00954 (2018).
33. Gidney, C. Halving the cost of quantum addition. *Quantum* **2**, 74 (2018).
34. Meuli, G., Soeken, M., Roetteler, M., Bjorner, N. & De Micheli, G. Reversible pebbling game for quantum memory management. In *DATE* 288–291 (IEEE, 2019); https://doi.org/10.23919/date.2019.8715092
35. Brassard, G., Høyer, P., Mosca, M. & Tapp, A. Quantum amplitude amplification and estimation. *Quantum Comput. Inf.* **305**, 53–74 (2002).
36. Kitaev, A. Quantum measurements and the abelian stabilizer problem. Preprint at https://arxiv.org/abs/quant-ph/9511026 (1995).
37. Shor, P. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM J. Comput.* **26**, 1484–1509 (1997).
38. *Quantum Algorithm Zoo*; http://quantumalgorithmzoo.org/
39. Houck, A. A., Koch, J., Devoret, M. H., Girvin, S. M. & Schoelkopf, R. J. Life after charge noise: recent results with transmon qubits. *Quantum Inf. Process.* **8**, 105–115 (2009).
40. Barends, R. et al. Coherent Josephson qubit suitable for scalable quantum integrated circuits. *Phys. Rev. Lett.* **111**, 080502 (2013).
41. Imamoglu, A. et al. Quantum information processing using quantum dot spins and cavity QED. *Phys. Rev. Lett.* **83**, 4204–4207 (1999).
42. Cirac, J. I. & Zoller, P. Quantum computations with cold trapped ions. *Phys. Rev. Lett.* **74**, 4091–4094 (1995).
43. Nayak, C., Simon, S. H., Stern, A., Freedman, M. & DasSarma, S. Non-Abelian anyons and topological quantum computation. *Rev. Mod. Phys.* **80**, 1083–1159 (2008).
44. Preskill, J. Quantum computing in the NISQ era and beyond. *Quantum* **2**, 79 (2018).
45. Smith, R. S., Curtis, M. J. & Zeng, W. J. A practical quantum instruction set architecture. Preprint at https://arxiv.org/abs/1608.03355 (2016).
46. Svore, K. M. et al. Q#: enabling scalable quantum computing and development with a high-level domain-specific language. In *Proc. Real World Domain Specific Languages Workshop*, 7 (ACM, 2018).
47. Svore, K. M., Aho, A. V., Cross, A. W., Chuang, I. & Markov, I. L. A layered software architecture for quantum computing design tools. *Computer* **39**, 74–83 (2006).
48. Cross, A. W., Bishop, L. S., Smolin, J. A. & Gambetta, J. M. Open quantum assembly language. Preprint at https://arxiv.org/abs/1707.03429 (2017).
49. Cirq Documentation; https://cirq.readthedocs.io/en/stable/
50. Häner, T. & Steiger, D. S. 0.5 petabyte simulation of a 45-qubit quantum circuit. In *Proc. International Conference for High Performance Computing, Networking, Storage and Analysis* 33 (ACM, 2017); https://doi.org/10.1145/3126908.3126947
51. Kornyik, M. & Vukics, A. The Monte Carlo wave-function method: a robust adaptive algorithm and a study in convergence. *Comput. Phys. Commun.* **238**, 88–101 (2019).
52. Aaronson, S. & Gottesman, D. Improved simulation of stabilizer circuits. *Phys. Rev. A* **70**, 052328 (2004).
53. Steiger, D. S., Häner, T. & Troyer, M. ProjectQ: an open source software framework for quantum computing. Preprint at https://arxiv.org/abs/1612.08091 (2016).
54. Reiher, M., Wiebe, N., Svore, K. M., Wecker, D. & Troyer, M. Elucidating reaction mechanisms on quantum computers. *Proc. Natl Acad. Sci. USA* **114**, 7555–7560 (2017).
55. Rand, R., Paykin, J. & Zdancewic, S. QWIRE practice: formal verification of quantum circuits in Coq. In *Proc. 14th International Conference on Quantum Physics and Logic, QPL 2017*, *EPTCS Vol. 266* 119–132 https://doi.org/10.4204/EPTCS.266.8 (Open Publishing Association, 2018).
56. Shi, Y. et al. Contract-based verification of a realistic quantum compiler. Preprint at https://arxiv.org/abs/1908.08963 (2019).
57. Ying, M. Toward automatic verification of quantum programs. *Formal Aspects Comput.* **31**, 3–25 (2019).
58. Nielsen, M. A. & Chuang, I. L. *Quantum Computation and Quantum Information: 10th Anniversary Edition* (Cambridge University Press, 2011).
59. Microsoft Quantum Documentation; https://docs.microsoft.com/quantum
60. Qiskit documentation; https://qiskit.org/documentation/
61. Kluyver, T. et al. Jupyter Notebooks — a publishing format for reproducible computational workflows. In *Positioning and Power in Academic Publishing: Players, Agents and Agendas, 20th International Conference on Electronic Publishing* 87–90 https://doi.org/10.3233/978-1-61499-649-1-87 (2016).
62. IBM Quantum Experience; https://quantum-computing.ibm.com/
63. Experience quantum impact with Azure Quantum. *Microsoft* https://cloudblogs.microsoft.com/quantum/2019/11/04/announcing-microsoft-azure-quantum/ (2019).
64. Ho, A. & Bacon, D. Announcing Cirq: an open source framework for NISQ algorithms. *Google AI Blog* https://ai.googleblog.com/2018/07/announcing-cirq-open-source-framework.html (2018).
65. Rigetti, C. Introducing Rigetti quantum cloud services. *Rigetti* https://medium.com/rigetti/introducing-rigetti-quantum-cloud-services-c6005729768c (2018).
66. Alpine Quantum Technologies (AQT); https://www.aqt.eu/solutions/
67. QuTech Quantum Inspire Home. https://www.quantum-inspire.com/
68. Barr, J. Amazon Braket — get started with quantum computing. *AWS News Blog* https://aws.amazon.com/blogs/aws/amazon-braket-get-started-with-quantum-computing/ (2019).
69. PyQuil. *GitHub* https://github.com/rigetti/pyquil
70. Steiger, D. S., Häner, T. & Troyer, M. ProjectQ: an open source software framework for quantum computing. *Quantum* **2**, 49 (2018).
71. ProjectQ. *GitHub* https://github.com/ProjectQ-Framework/ProjectQ
72. Paykin, J., Rand, R. & Zdancewic, S. Qwire: a core language for quantum circuits. In *Proc. 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017* 846–858 (AM, 2017); https://doi.org/10.1145/3009837.3009894
73. QWIRE. *GitHub* https://github.com/inQWIRE/QWIRE
74. Amy, M. & Gheorghiu, V. staq — A full-stack quantum processing toolkit. *Quantum Sci. Technol.* **5**, 034016 (2019).
75. staq. *GitHub* https://github.com/softwareQinc/staq
76. Killoran, N. et al. Strawberry fields: a software platform for photonic quantum computing. *Quantum* **3**, 129 (2019).
77. Strawberry Fields. *GitHub* https://github.com/XanaduAI/strawberryfields
78. Sivarajah, S. et al. t|ket⟩: a retargetable compiler for NISQ devices. *Quantum Sci. Technol.* https://doi.org/10.1088/2058-9565/ab8e92 (2020).
79. t|ket⟩. *GitHub* https://github.com/CQCL/pytket
80. McCaskey, A. J., Lyakh, D. I., Dumitrescu, E. F., Powers, S. S. & Humble, T. S. XACC: a system-level software infrastructure for heterogeneous quantum-classical computing. Preprint at https://arxiv.org/abs/1911.02452 (2019).
81. XACC. *GitHub* https://github.com/eclipse/xacc

82. QuTiP Documentation; http://qutip.org/documentation.html
83. Johansson, J., Nation, P. & Nori, F. QuTiP: an open-source Python framework for the dynamics of open quantum systems. *Comput. Phys. Commun.* **183**, 1760–1772 (2012).
84. Johansson, J., Nation, P. & Nori, F. QuTiP 2: a Python framework for the dynamics of open quantum systems. *Comput. Phys. Commun.* **184**, 1234–1240 (2013).
85. LaRose, R. Overview and comparison of gate level quantum software platforms. *Quantum* **3**, 130 (2019).
86. Mosca, M., Roetteler, M. & Selinger, P. Quantum programming languages (Dagstuhl Seminar 18381). *Dagstuhl Rep.* **8**, 112–132 (2018).
87. Bennett, C. H. et al. Teleporting an unknown quantum state via dual classical and Einstein–Podolsky–Rosen channels. *Phys. Rev. Lett.* **70**, 1895–1899 (1993).
88. Brassard, G., Braunstein, S. & Cleve, R. Teleportation as a quantum computation. *Physica D* **120**, 43–47 (1998).
89. Docker, Inc; http://www.docker.com/
90. Heim, B. Q# 0.6: language features and more. *Microsoft* https://devblogs.microsoft.com/qsharp/qsharp-06-language-features-and-more/ (2019).
91. Geller, A. Why do we need Q#? *Microsoft* https://devblogs.microsoft.com/qsharp/why-do-we-need-q/ (2018).
92. Rios, F. & Selinger, P. A categorical model for a quantum circuit description language. In *Proc. 14th International Conference on Quantum Physics and Logic, QPL 2017* 164–178 (2017); https://doi.org/10.4204/EPTCS.266.11
93. Geller, A. What are qubits? *Microsoft* https://devblogs.microsoft.com/qsharp/what-are-qubits/ (2019).
94. Language Server Protocol; https://microsoft.github.io/language-server-protocol/
95. Mykhailova, M. & Svore, K. M. Teaching quantum computing through a practical software-driven approach: experience report. In *Proc. 51st ACM Technical Symposium on Computer Science Education, SIGCSE '20* 1019–1025 (ACM, 2020); https://doi.org/10.1145/3328778.3366952
96. Q# samples. *GitHub* https://github.com/microsoft/Quantum
97. Q# libraries. *GitHub* https://github.com/microsoft/QuantumLibraries
98. Abraham, H. et al. Qiskit: an open-source framework for quantum computing. *Zenodo* https://doi.org/10.5281/zenodo.2562110 (2019).
99. Cross, A. The IBM Q experience and QISKit open-source quantum computing software. In *APS March Meeting Abstracts*, **2018**, L58.003 (2018).
100. Qiskit/qiskit-aqt-provider. *GitHub* https://github.com/Qiskit/qiskit-aqt-provider (2020).
101. Qiskit/qiskit-honeywell-provider. *GitHub* https://github.com/Qiskit/qiskit-honeywell-provider (2020).
102. Asfaw, A. et al. *Learn Quantum Computation Using Qiskit* https://qiskit.org/
103. Jupyter/jupyter-book. *GitHub* https://github.com/jupyter/jupyter-book
104. Paetznick, A. & Svore, K. M. Repeat-until-success: non-deterministic decomposition of single-qubit unitaries. *Quantum Inf. Comput.* **14**, 1277–1301 (2014).
105. Broughton, M. et al. TensorFlow Quantum: a software framework for quantum machine learning. Preprint at https://arxiv.org/abs/2003.02989 (2020).
106. McClean, J. R. et al. OpenFermion: the electronic structure package for quantum computers. Preprint at https://arxiv.org/abs/1710.07629 (2017).
107. Abadi, M. et al. TensorFlow: a system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* 265–283 (2016).
108. Cirq source code remarks. *GitHub* https://github.com/quantumlib/Cirq/blob/master/cirq/google/engine/engine.py (2018).
109. Green, A. S., Lumsdaine, P. L., Ross, N. J., Selinger, P. & Valiron, B. Quipper: a scalable quantum programming language. In *Proc. 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* 333–342 (ACM, 2013); https://doi.org/10.1145/2491956.2462177
110. Smith, J. M., Ross, N. J., Selinger, P. & Valiron, B. Quipper: concrete resource estimation in quantum algorithms. Preprint at https://arxiv.org/abs/1412.0625 (2014).
111. Anticoli, L., Piazza, C., Taglialegne, L. & Zuliani, P. Verifying quantum programs: from Quipper to QPMC. Preprint at https://arxiv.org/abs/1708.06312 (2017).
112. Mahmoud, M. Y. & Felty, A. P. Formalization of metatheory of the Quipper quantum programming language in a linear logic. *J. Autom. Reason.* **63**, 967–1002 (2019).
113. Ross, N. J. Algebraic and logical methods in quantum computation. Preprint at https://arxiv.org/abs/1510.02198 (2015).
114. Fu, P., Kishida, K., Ross, N. J. & Selinger, P. A tutorial introduction to quantum circuit programming in dependently typed proto-quipper. In *Reversible Computation - 12th International Conference, RC 2020, Proceedings* (eds Lanese, I. & Mariusz, R.) 153-168 (2020).
115. Childs, A. M., Maslov, D., Nam, Y. S., Ross, N. J. & Su, Y. Toward the first quantum simulation with quantum speedup. *Proc. Natl Acad. Sci. USA* **115**, 9456–9461 (2018).
116. Abhari, A. J. et al. *Scaffold: Quantum Programming Language* Technical Report TR-934-12 (Princeton Univ., 2012).
117. Abhari, A. J. et al. ScaffCC: scalable compilation and analysis of quantum programs. *Parallel Comput.* **45**, 2–17 (2015).
118. Abhari, A. J. et al. *ScaffCC User Manual* (2018).
119. Soeken, M., Frehse, S., Wille, R. & Drechsler, R. Revkit: a toolkit for reversible circuit design. *Multiple Valued Log. Soft Comput.* **18**, 55–65 (2012).
120. Khammassi, N. QX Quantum Computer Simulator; http://www.quantum-studio.net/
121. Scaffold. *GitHub* https://github.com/epiqc/ScaffCC
122. Kliuchnikov, V. Wrong QASM output for teleportation circuit. *GitHub* https://github.com/epiqc/ScaffCC/issues/28 (2018).
123. Javadi-Abhari, A. et al. Optimized surface code communication in superconducting quantum computers. In *Proc. 50th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-50 '17* 692–705 (ACM, 2017); https://doi.org/10.1145/3123939.3123949
124. The Quipper System https://www.mathstat.dal.ca/~selinger/quipper/doc/

Acknowledgements
The authors thank P. Selinger, J. Ross, A. Javadi-Abhari and M. Martonosi for their input and advice. The authors thank V. Kliuchnikov for discussions and for sharing his perspectives.

Author contributions
B.H., M.S., S.M. and C.G. researched data for the article. B.H., M.S., M.R., M.T. and K.S. contributed substantially to the discussion of the content. B.H., M.S., S.M., C.G. and A.G. wrote the manuscript. B.H., M.R., A.G. and K.S. reviewed and edited the manuscript before submission.

Competing interests
All authors are employed by Microsoft.

Peer review information
Nature Reviews Physics thanks Paul Nation, Mingsheng Ying and the other, anonymous, reviewer(s) for their contribution to the peer review of this work.

Supplementary information
Supplementary information is available for this paper at https://doi.org/10.1038/s42254-020-00245-7.