

Qwerty: A Basis-Oriented Quantum Programming Language

AUSTIN J. ADAMS, Georgia Tech, USA

SHARJEEL KHAN, Georgia Tech, USA

JEFFREY S. YOUNG, Georgia Tech, USA

THOMAS M. CONTE, Georgia Tech, USA

Quantum computers have evolved from the theoretical realm into a race to large-scale implementations. This is due to the promise of revolutionary speedups, where achieving such speedup requires designing an algorithm that harnesses the structure of a problem using quantum mechanics. Yet many quantum programming languages today require programmers to reason at a low level of quantum gate circuitry. This presents a significant barrier to entry for programmers who have not yet built up an intuition about quantum gate semantics, and it can prove to be tedious even for those who have. In this paper, we present Qwerty, a new quantum programming language that allows programmers to manipulate qubits more expressively than gates, relegating the tedious task of gate selection to the compiler. Due to its novel basis type and easy interoperability with Python, Qwerty is a powerful framework for high-level quantum-classical computation.

1 INTRODUCTION

Quantum computers have evolved from the theoretical realm into an active and highly competitive commercial race to large-scale implementations. This is due to the promise of revolutionary speedups for important problems such as unstructured search and factoring large integers [19, 30, 45]. Achieving such speedup requires designing an algorithm that harnesses the structure of a problem using quantum mechanics [1, 46]. Yet even if a programmer understands how a quantum algorithm operates, realizing the algorithm using existing quantum programming languages requires a mastery of quantum gate engineering. This significant, abrupt gap between algorithms and low-level quantum gates suggests a need for a quantum programming language beyond gates.

Table 1 summarizes the computational constructs and key features of existing quantum programming languages. The “computational construct” column is meant as the main tool programmers realistically employ to express quantum operations. For instance, the theoretical work QuantumII explores a novel quantum programming language design. Ultimately though, after defining gates in terms of these new features, the paper’s example programs are presented in terms of gates [9, §9]. A few prior languages allow quantum programming without gates: Tower [57], OQIMP [22], Neko [38], and Aleph [36]. However, these languages are not suitable for efficient general-purpose quantum programming. Tower focuses only on data structures; OQIMP is designed specifically for synthesizing oracles (Quipper [17] contains similar functionality); Neko requires that the algorithm is compatible with a map-filter-reduce structure; and Aleph synthesizes only amplitude amplification and thus cannot achieve exponential speedup for factoring, for instance.

In this paper, we present **Qwerty**, a new quantum programming language. Thanks to Qwerty’s novel **basis** type and its ability to embed classical computation in quantum code, Qwerty allows programmers to implement algorithms with prospective quantum advantage without low-level gate engineering. Because Qwerty is embedded as a Python DSL, interoperability between Python and Qwerty is easy, making Qwerty a robust framework for mixed quantum-classical computation.

Qwerty is introduced through examples, first via the Bernstein–Vazirani algorithm (Section 3.1) and then many prominent quantum algorithms (Section 4). An introduction to quantum notation is provided for the non-expert (Section 2). Appendix A presents the soundness of the semantics and type system of Qwerty.

Table 1. Comparison with Prior Work on Advancing Quantum Programming

Name	Computational construct	Key feature(s)
QCL [33]	Gates	First quantum programming language
Scaffold [2, 23]	Gates	Integration with C++, oracle synthesis
Quipper [17]	Gates	Functional circuit construction
Qiskit [40]	Gates	Convenient Python integration, rich tooling
Q# [24, 50]	Gates	Standard library and functional programming
QCOR [27, 28]	Gates	C++ integration, physics features
OpenQASM 3 [11]	Gates	Adds structured control flow to circuits
QWIRE [35]	Gates	Linear qubit types
Silq [6], Qiskit++ [34]	Gates	Simplifies managing uncomputation efficiently
Qunity [52]	Gates	Unifies quantum and classical programming
Twist [58]	Gates	Prevents accidental entanglement bugs
Tower [57]	Linked lists	Quantum data structures
quAPL [32]	Gates	Introduces a quantum array language
QuantumΠ [9]	Gates	Defines a quantum PL using classical PLs
OQASM [22]	Gates	Basis-aware circuit-level programming
OQIMP [22]	Classical code	Verified synthesis of black-box oracles
Neko [38]	MapReduce	Programs have map-filter-reduce structure
Aleph [36]	Universes	User-friendly amplitude amplification
<i>Qwerty (this work)</i>	<i>Basis Translations</i>	Gate-free programming, Python integration

2 INTRODUCTION TO QUANTUM NOTATION

(This section introduces quantum computing notation. Readers who are familiar already can safely skip to the next section of the paper.)

The state of a qubit is a unit vector in a 2D complex vector space with an inner product — a *Hilbert space* — which we will write as \mathcal{H}_2 . In bra-ket (or Dirac) notation, $|\psi\rangle$ denotes a vector in \mathcal{H}_2 . The inner product of $|\phi\rangle$ and $|\psi\rangle$ is written as $\langle\phi|\psi\rangle$. Given two qubits with states $|\psi\rangle$ and $|\phi\rangle$, the state of the two-qubit system is written with the tensor product \otimes as $|\psi\rangle \otimes |\phi\rangle$; this product is a unit vector in a four-dimensional Hilbert space $\mathcal{H}_2 \otimes \mathcal{H}_2$. (It is common to write $|\psi\rangle |\phi\rangle$ as shorthand for $|\psi\rangle \otimes |\phi\rangle$.) This pattern continues, with every additional qubit doubling the dimension of the state space, i.e., doubling the number of complex numbers (*amplitudes*) needed to describe a state.

The vectors $|0\rangle$ and $|1\rangle$ represent the standard basis for a one-qubit (2D) space. For a two-qubit (4D) space, the vectors $|00\rangle$, $|01\rangle$, $|10\rangle$, and $|11\rangle$ are the standard basis. In general, the 2^n standard basis vectors for an n -qubit state are labeled with n -bit bitstrings. Typical measurement projects onto one of these standard basis states, with the bits labeling the basis state being the measurement outcome, and the likelihood being proportional to the norm of the projection. (Note that in fact, measurement can be performed in any basis, although this may not be convenient in hardware.) For example, the state $\sqrt{9/10}|00\rangle + \sqrt{1/10}|11\rangle$ is most likely to produce the measurement 00 when measuring in the standard basis, since the projection onto $|00\rangle$ has a larger norm than the projection onto $|11\rangle$.

Qubit state must evolve reversibly, which can be expressed as requiring it to evolve by a $2^n \times 2^n$ unitary operator U , like $|\psi'\rangle = U|\psi\rangle$. The unitary condition is that U is invertible and $\|U|\psi\rangle\| =$

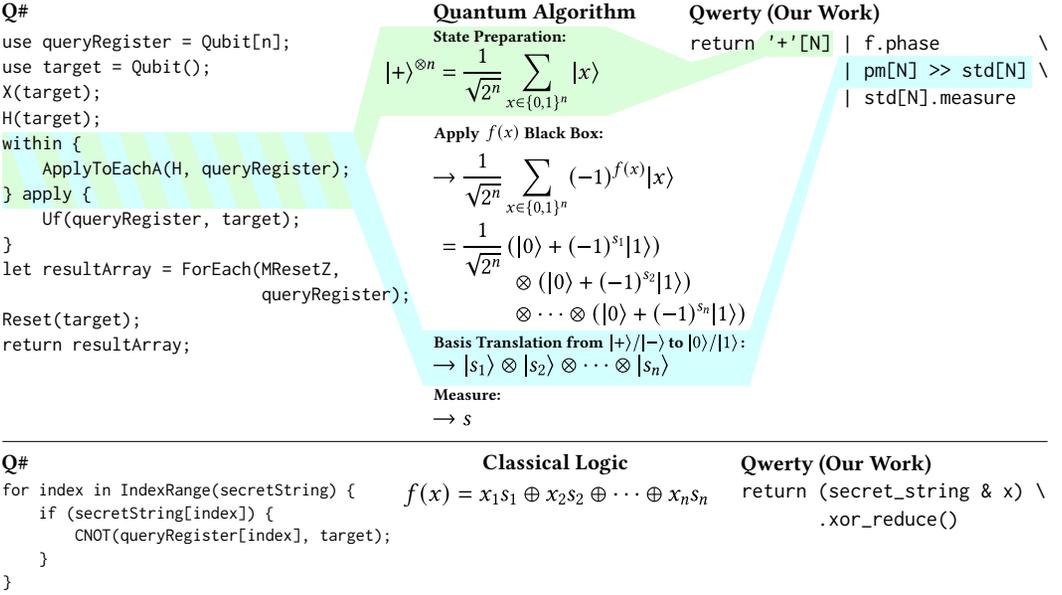


Fig. 1. A comparison between Q# and Qwerty implementations of the Bernstein–Vazirani algorithm [37], which finds a secret string s with only one invocation of black-box implementation of a classical function $f(x) = x \cdot s$.

$\| |\psi\rangle \|$ [3], i.e., that U^{-1} exists and that U always preserves the norm of any input $|\psi\rangle$. By the spectral decomposition, any unitary can be expressed as a diagonal matrix $U = \sum_{\lambda} \lambda |\lambda\rangle \langle \lambda|$ [30]. For example, the Pauli X matrix $\sigma_x = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$ can be diagonalized as $\sigma_x = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$ if the rows and columns are written in terms of the basis $|+\rangle, |-\rangle$ rather than the standard $|0\rangle, |1\rangle$ basis. (The vectors $|+\rangle \triangleq \frac{1}{\sqrt{2}} (|0\rangle + |1\rangle)$ and $|-\rangle \triangleq \frac{1}{\sqrt{2}} (|0\rangle - |1\rangle)$. Both are eigenvectors of σ_x .)

For a more thorough introduction to quantum computing and its notation, we direct readers to Chapters 1 and 2, respectively, of Nielsen and Chuang [30].

3 INTRODUCTION TO QWERTY

3.1 Motivating Example: Bernstein–Vazirani

To introduce Qwerty, we begin with an example using the well-known *Bernstein–Vazirani algorithm* [5, 14]. Assume there is a secret n -bit bitstring s and a “black-box” function that performs $f(x) = x_1 s_1 \oplus x_2 s_2 \oplus \cdots \oplus x_n s_n$. (Here, \oplus denotes XOR, and $x_i s_i$ denotes ANDing x_i with s_i .) The Bernstein–Vazirani algorithm returns s using only a single invocation of $f(x)$ – a classical solution would need n invocations ($f(100 \cdots 0)$ to get s_1 , $f(010 \cdots 0)$ to get s_2 , etc.).

The steps in the Bernstein–Vazirani algorithm are shown mathematically in the center column of Fig. 1. Each of the four steps corresponds to a key primitive (or feature) of Qwerty: (1) qubit *state preparation* (i.e., initialization), (2) application of the reversible version of $f(x)$, (3) basis translation, and (4) measurement. The first step prepares the initial state $|+\cdots+\rangle$. Then, when $f(x)$ is applied in the next step, each $|+\rangle$ in this state is changed to a $|-\rangle$ state if the corresponding bit of s is 1. The next step is the most pivotal step, where all qubits undergo the transformation $\alpha |+\rangle + \beta |-\rangle \rightarrow \alpha |0\rangle + \beta |1\rangle$. In Qwerty, this transformation is called a **basis translation** from the basis $|+\rangle, |-\rangle$ to the basis $|0\rangle, |1\rangle$. We call this a translation because when representing the input

<pre> use q = Qubit (); H(q); </pre> <p>(a) Q#: Preparing $+\rangle$</p>	<pre> '+' </pre> <p>(b) Qwerty: Preparing $+\rangle$</p>
<pre> use q = Qubit[3]; X(q[0]); X(q[2]); </pre> <p>(c) Q#: Preparing $101\rangle$</p>	<pre> '101' </pre> <p>(d) Qwerty: Preparing $101\rangle$</p>

Fig. 2. Side-by-side comparison of state preparation in traditional gate-based languages versus in Qwerty.

state as a linear combination of basis vectors, it *translates* the basis vectors element-wise into a different basis without touching the coefficients (i.e., the amplitudes)¹.

The right-hand column of Fig. 1 shows Bernstein–Vazirani written in Qwerty. Contrast this with the left-hand column of Fig. 1, which shows the same algorithm written in Q# [37], a prominent quantum programming language. Not only is Qwerty more concise, but it is also more expressive. For example, the highlighted Q# lines in Fig. 1 perform both routine state preparation and also the most crucial part of the algorithm, basis translation. By contrast, Fig. 1 shows that the Qwerty code has explicit syntax for both steps: preparing $|+\rangle^{\otimes n}$ (written as '+'[N]) and translating from the n -qubit plus/minus basis to the n -qubit standard basis (written as **pm**[N] >> **std**[N]). Compared to the mere H used in the Q# code, which represents a low-level Hadamard gate, these Qwerty constructs allow programmers to better express intent.

Quantum algorithms aimed at classical problems typically invoke classical functions on qubits, so language support for expressing this classical logic intuitively is just as important as convenient expression of quantum logic. The bottom of Fig. 1 compares expressions of such classical logic in Q# versus Qwerty. Even though the Bernstein–Vazirani black-box function $f(x)$ can be easily expressed with well-known classical logic gates, Q# requires either programming in quantum gates (bottom left of Fig. 1) or calling library functions that apply quantum gates. In Qwerty, on the other hand, programmers can express classical functions directly as classical logic, as shown in the bottom right of Fig. 1. Such classical functions can be invoked directly from Python and execute in the Python interpreter, or they can be instantiated inside quantum code (top right of Fig. 1) and lowered to quantum gates by the compiler. Combined with the aforementioned mechanisms for state preparation and basis translation, this functionality allows programmers to implement well-known quantum algorithms such as Grover’s or Shor’s efficiently without writing a single quantum gate.

3.2 Overview of Qwerty

In this section, we present the key Qwerty features that enable more expressive quantum programming.

3.2.1 The qubit type. The **qubit** type in Qwerty is *linear* [35, 43], meaning that every **qubit** must be used exactly once. This prevents attempting to duplicate qubit states, which violates the no-cloning theorem [30], or accidentally discarding a qubit entangled with other qubits, causing unexpected behavior [58].

¹Traditionally in quantum computing, this operation might be called a “change of basis”, but we have found this term is too easily confused with the linear algebra procedure [3] that changes only the representation of a vector, not its value.

<pre> ApplyToEach(H, q); (a) Q#: Converting many qubits from the $0\rangle/ 1\rangle$ basis to the $+\rangle/ -\rangle$ basis within { ApplyToEachA(H, q); ApplyToEachA(X, q); } apply { Controlled Z(Most(q), Tail(q)); } (c) Q#: The diffuser in Grover's algo- rithm [51], which multiplies all $+\rangle/ -\rangle$ ba- sis states by a negative phase except $+\rangle^{\otimes n}$ </pre>	<pre> std[N] >> pm[N] (b) Qwerty: Converting N qubits from the $0\rangle/ 1\rangle$ basis to the $+\rangle/ -\rangle$ basis -('+' [N] >> - '+' [N]) (d) Qwerty: The diffuser in Grover's algo- rithm, which multiplies all $+\rangle/ -\rangle$ basis states by a negative phase except $+\rangle^{\otimes n}$ </pre>
---	---

Fig. 3. Side-by-side comparison of typical operations in gate-oriented languages versus their more expressive equivalents in Qwerty.

Qubit literals. Programmers can introduce **qubits** using qubit literals, which efficiently prepare a subset of qubit states. For example, `'-'` prepares the state $|-\rangle$, `'110'` prepares the state $|110\rangle$, and so on. Fig. 2 demonstrates how Qwerty qubit literals are more succinct and expressive than equivalent code in gate-oriented languages.

Tensor product. In general, one can (loosely) view the tensor product as concatenating vectors or vector spaces [3, 30]. Thus, given that a string-like syntax is used for qubit literals, the typical string concatenation operator `+` is a natural fit for the tensor product. For instance, `'1' + '0' + '1'` is equivalent to `'101'`. Repeated tensor product is accomplished with `'0'[N]`, which is equivalent to $\underbrace{'0' + '0' + \dots + '0'}_N$.

3.2.2 The basis type. The heart of Qwerty is the **basis** type, which facilitates intuitive, higher-level programming without sacrificing efficiency. A **basis** in Qwerty represents exactly the straightforward concept of an orthonormal basis taught in undergraduate linear algebra courses [3], and Qwerty primitives for state evolution and measurement are defined in terms of a **basis**.

Basis literals. Qwerty includes built-in bases such as the standard $|0\rangle/|1\rangle$ basis (**std**), the $|+\rangle/|-\rangle$ basis (**pm**), the $|+i\rangle/|-i\rangle$ basis (**ij**)², and the N-qubit Fourier basis (**fourier**[N]) [30]. Programmers can construct more sophisticated bases by wrapping lists of qubit literals in curly brackets; for example, `{'0', '1'}` is equivalent to **std**. On the other hand, `{'0', phase(theta)*'1'}` is not, instead representing the basis consisting of $|0\rangle$ and $e^{i\theta}|1\rangle$.

Translation. Unitary state evolution in Qwerty is written using a *basis translation*, written `basisin >> basisout`. Fig. 3 shows some examples. For two bases `basisin` and `basisout` that span the same (sub)space of m qubits, the basis translation `basisin >> basisout` performs the following

²In deference to electrical engineers, Qwerty uses the literal `'j'` to represent $|-i\rangle$.

<pre>MResetZ (q);</pre> <p>(a) Q#: Measuring in the $0\rangle/ 1\rangle$ basis</p>	<pre>std.measure</pre> <p>(b) Qwerty: Measuring in the $0\rangle/ 1\rangle$ (std) basis</p>
<pre>Adjoint QFT (q);</pre> <pre>ForEach (MResetZ, q);</pre> <p>(c) Q#: Measuring in the Fourier basis</p>	<pre>fourier[N].measure</pre> <p>(d) Qwerty: Measuring in the Fourier basis</p>

Fig. 4. Side-by-side comparison of measurements traditional languages versus their equivalents in Qwerty.

transformation:

$$\begin{aligned}
 & \alpha_1 \left| \text{basis}_{\text{in}}^{(1)} \right\rangle + \alpha_2 \left| \text{basis}_{\text{in}}^{(2)} \right\rangle + \cdots + \alpha_n \left| \text{basis}_{\text{in}}^{(n)} \right\rangle \\
 & \quad \Downarrow \\
 & \alpha_1 \left| \text{basis}_{\text{out}}^{(1)} \right\rangle + \alpha_2 \left| \text{basis}_{\text{out}}^{(2)} \right\rangle + \cdots + \alpha_n \left| \text{basis}_{\text{out}}^{(n)} \right\rangle
 \end{aligned}$$

Basis translations can operate on subspaces, as shown in Fig. 3d, where the basis $\{ '+' [N] \}$ (written without braces as $'+' [N]$ as syntactic sugar) indicates that the translation $'+' [N] \gg - '+' [N]$ acts only on the subspace spanned by $|+\rangle^{\otimes N}$ — specifically, it maps $|+\rangle^{\otimes N} \mapsto -|+\rangle^{\otimes N}$ and leaves other $\text{pm}[N]$ basis states alone.

The basis translation primitive in Qwerty simplifies many popular operations in quantum algorithms — for instance, quantum Fourier transform (QFT) is performed with simply **std[N] >> **fourier**[N]**. Furthermore, this abstraction is cheap because translations map well to multi-controlled gates. (For a rigorous definition of the semantics of basis translations, see Appendix A.)

Translation syntactic sugar. For convenience, Qwerty includes b .**flip** and b .**rotate**(θ) constructs for every basis b that spans the full one-qubit space. b .**flip** swaps the basis vectors of b , and b .**rotate**(θ) rotates around b by θ radians. For example, **std.flip** is equivalent to **std >> { '1', '0' }**, and **std.rotate**(theta) is compiled to **std >> { **phase**(-theta/2)*'0', **phase**(theta/2)*'1' }**. Both examples coincide exactly with the X and $R_z(\theta)$ gates, respectively [30].

To facilitate further expressiveness, Qwerty also includes $q\ell$.**prep**, where $q\ell$ is a qubit literal (Section 3.2.1). This construct allows access to the plumbing by which the compiler lowers qubit literals. For example, an ordinary qubit literal $'10+'$ is equivalent to $'000' | '10+'.\text{prep}$, which in turn is equivalent to $'000' | \text{std.flip+id}+(\text{std}>>\text{pm})$. Observe that $'10+'.\text{prep}$ is much easier to read than **std.flip+id+(std>>pm)**, and the expected input state ($'000'$) is more explicit.

Measurement. It is common to view qubit measurement as being performed in some basis [30, §2.2.5], e.g., the standard basis or the $|+\rangle/|-\rangle$ basis. Rather than hard-coding bases in the names of bespoke standard library functions for measurement as languages like Q# do (with **MResetZ**, **MResetX**, etc.), Qwerty defines measurement as an operation on a **basis**. Fig. 4 compares measurement in Qwerty with measurement in traditional gate-oriented languages.

3.2.3 Reversible quantum functions. Many quantum subroutines such as phase estimation (Section 4.2.1) take black-box quantum operations as input, eventually executing them backwards or in a subspace. Qwerty includes features that facilitate this kind of modular programming. However, these features require that the quantum operation includes no qubit allocation, discarding, or

```

1 import sys
2 from qwerty import *
3
4 def bv(secret_string):
5     @classical[N](secret_string)
6     def f(secret_string: bit[N], x: bit[N]) -> bit:
7         return (secret_string & x).xor_reduce()
8
9     @qpu[N](f)
10    def kernel(f: cfunc[N,1]) -> bit[N]:
11        return '+'[N] | f.phase \
12                | pm[N] >> std[N] \
13                | std[N].measure
14
15    return kernel()
16
17 secret_string = bit.from_str(sys.argv[1])
18 print(bv(secret_string))

```

Fig. 5. A full Python module including the Qwerty Bernstein–Vazirani code from Fig. 1.

measurement³. (This is because it is absurd to un-measure a qubit, for example.) We call such operations *reversible quantum functions*.

Execution in a subspace. If b is a basis and f is a reversible quantum function, then the syntax $b \ \& \ f$ (or $f \ \& \ b$) yields a new function that executes f in the subspace identified by b . For example, the basis translation `'1' + std >> '1' + {'1', '0'}` can be written slightly more succinctly as `'1' & std >> {'1', '0'}` instead, as if the `'1'` portion of each basis was factored out. (Note that `&` has lower precedence than `>>`.)

Running code backwards. If f is a reversible quantum function, then $\sim f$ is a function that runs f backwards. For example, `~std.rotate(pi/4)` is equivalent to `std.rotate(-pi/4)`.

3.2.4 Quantum kernels. Using Qwerty, quantum kernels⁴ are written as Python functions with the `@qpu` annotation. Fig. 5 shows a full example of Bernstein–Vazirani written in Qwerty. The contents of `@qpu` kernels use Qwerty syntax and are sent through the Qwerty compiler. Specifically, they run on a quantum accelerator (or simulator), not in the Python interpreter. Drawing such a separation between Python and Qwerty code sidesteps complexities in analysis of Qwerty [35] and maintains the convenience of existing classical libraries for e.g. numerical optimization.

Dimension variables. Quantum kernels may have *dimension variables* (e.g., the N on line 9 of Fig. 5) to make them polymorphic in the number of qubits on which they operate. The compiler attempts to infer dimension variable based on the type of *captured* objects (e.g., the capture f of `kernel()`, specified on line 9 and whose type is declared on line 10 of Fig. 5). When the compiler cannot infer a dimension variable, such as N on line 5 of Fig. 6, the programmer must specify it using `[[...]]` notation as shown on line 10 of Fig. 6. (Instantiation with `[[...]]` can also be performed inside a quantum kernel, as used in line 12 of Fig. 13a.)

³Qubit allocation is permitted in reversible quantum functions if allocated temporary qubits (ancilla qubits) are discarded cleanly with `discardz`; see Section A.5 in Appendix A.

⁴Here, “kernel” is meant in the sense of an accelerator kernel such as a CUDA kernel.

```

1 import sys
2 from qerty import *
3
4 def ghz(n_qubits):
5     @qpu[N]
6     def kernel() -> bit[N]:
7         return '+' + '0'[N-1] | '1' & std.flip[N-1] \
8             | std[N].measure
9
10    return kernel[[n_qubits]](histogram=True,
11                               shots=2048)
12
13 n_qubits = int(sys.argv[1])
14 print_histogram(ghz(n_qubits))
15 # Prints:
16 # 00000000 -> 49.37%
17 # 11111111 -> 50.63%

```

Fig. 6. Preparing the Greenberger–Horne–Zeilinger (GHZ) state [18] in Qwerty. Measuring this entangled state yields either all 0s (line 16) or all 1s (line 17).

Controlled Z(Most(q), Tail(q));

(a) Q#: Flag $11 \cdots 1$ (all ones) as the answer using a multi-controlled Z gate [51]

return x.and_reduce()

(b) Qwerty: Flag $11 \cdots 1$ (all ones) as the answer by ANDing together all input bits

Fig. 7. Side-by-side comparison of a Grover’s oracle (criteria) that flags $11 \cdots 1$ as the answer written in both Q# versus Qwerty.

3.2.5 Classical embedding. Typically, quantum algorithms that solve classical problems execute classical logic on qubits. For example, Bernstein–Vazirani (Section 3.1) invokes $f(x)$ on a special superposition, and Grover’s algorithm queries the search criteria, a classical black box, on a superposition of possible answers [19, 30]. Although it is obvious that classical logic is most easily programmed classically, gate-oriented programming languages often require writing such classical logic as low-level quantum gates. Fig. 7 compares a Grover’s oracle written in Qwerty with one written in Q#. Note that x in Fig 7b has type **bit**[N] (N classical bits), not **qubit**[N] (N qubits).

Classical functions. Classical functions are defined similarly to **@qpu** kernels and may also have dimension variables and captures as seen on lines 5-7 of Fig. 5. Functions decorated with **@classical** can be invoked from Python code to run them classically inside the Python interpreter rather than on qubits.

Instantiation. To invoke a **@classical** function f on qubits, a quantum kernel must *instantiate* f . The syntax $f.xor_embed$ realizes f as a Bennett embedding [4], which has the well-known form $B_f |x\rangle |y\rangle \triangleq |x\rangle |y \oplus f(x)\rangle$. However, many algorithms expect some $P_f |x\rangle \triangleq (-1)^{f(x)} |x\rangle$ instead, which the syntax $f.phase$ summons. (This abstracts away the implementation detail of converting a Bennett embedding into this form using $|-\rangle$ ancillas [30, §6.1.1].) Finally, if f is already reversible, as in the case of the multiplier used in Shor’s [30, 41], writing $f.inplace(f_inv)$ applies f in-place, i.e., $U_f |x\rangle \triangleq |f(x)\rangle$. Note that an implementation of f^{-1} (named f_inv for example) is currently required [4].

4 ALGORITHMS EXPRESSED IN QWERTY

This section shows Qwerty more concretely via Qwerty implementations of well-known algorithms. All examples compile and run as-is using the Qwerty compiler and runtime. However, some examples may contain more line breaks than typical Python code in order to fit within page margins.

This section presents Qwerty implementations of the following algorithms:

- §4.1: Oracle-based quantum algorithms
 - §4.1.1: Deutsch’s algorithm
 - §4.1.2: Deutsch–Jozsa algorithm
 - §4.1.3: Bernstein–Vazirani algorithm
 - §4.1.4: Quantum period finding
 - §4.1.5: Simon’s algorithm
- §4.2: Algorithms for factoring integers
 - §4.2.1: Quantum phase estimation
 - §4.2.2: Quantum order finding
 - §4.2.3: Shor’s algorithm
- §4.3: Quantum search algorithms
 - §4.3.1: Grover’s unstructured search
 - §4.3.2: Yoder–Low–Chuang fixed-point amplitude amplification
 - §4.3.3: Niroula–Nam string matching

(Examples begin on the next page and are intentionally laid out as one algorithm per page for easier reader navigation.)

4.1 Oracle-Based Algorithms

Many quantum algorithms designed to solve classical problems operate by running a classical black-box function (known as an oracle) on a superposition state and measuring the outcome. We begin with some well-known members of this group of algorithms because they include the most approachable examples.

4.1.1 Deutsch’s algorithm. Deutsch’s algorithm takes a black box function $f : \{0, 1\} \rightarrow \{0, 1\}$ as input and determines whether f is constant using a single invocation of f [10, 13, 30]. Specifically, the algorithm returns $f(0) \oplus f(1)$. Fig. 8a shows Deutsch’s algorithm and some example black boxes implemented in Qwerty.

Wrapping algorithms in Python functions such as `deutsch()` on line 3 of Fig. 8a is idiomatic Qwerty. This way, quantum kernels such as `kernel()` on line 5 of Fig. 8a can capture algorithm inputs (e.g., f in Fig. 8a) or the results of classical pre-processing. Wrapper functions may also perform classical post-processing before returning (e.g., Deutsch–Jozsa in Section 4.1.2).

The notation `cfunc` on line 5 of Fig. 8a is shorthand for the type of a function from bit to bit (`func[[bit], bit]`). Currently, Qwerty requires specifying the type of all captures (f in this case).

Line 6 of Fig. 8a performs the following steps:

- (1) Prepare $|+\rangle$ via the qubit literal `'+'`.
- (2) Apply $|x\rangle \mapsto (-1)^{f(x)}|x\rangle$, where $x \in \{0, 1\}$, using `f.phase` (see Section 3.2.5).
- (3) Measure in the $|+\rangle/|-\rangle$ (plus/minus) basis with `pm.measure`. Measuring $|+\rangle$ or $|-\rangle$ yields classical bits 0 and 1, respectively.

The `|` operator used on line 6 of Fig. 8a is a *pipe*, analogous to a Unix shell pipe. Qubits flow left-to-right through Qwerty pipelines.

Fig. 8b shows some Python code for invoking the Qwerty code from Fig. 8a. Line 1 of Fig. 8b defines `naive_classical()`, an equivalent classical implementation that performs two invocations of f rather than the single invocation required by the quantum algorithm. This demonstrates that Qwerty `@classical` functions (e.g.,

```

1 from qwerty import *
2
3 def deutsch(f):
4     @qpu(f)
5     def kernel(f: cfunc) -> bit:
6         return '+' | f.phase | pm.measure
7
8     return kernel()
9
10 @classical
11 def balanced(x: bit) -> bit:
12     return ~x
13
14 @classical
15 def constant(x: bit) -> bit:
16     return bit[1](0b1)

```

(a) Qwerty code for Deutsch’s algorithm

```

1 def naive_classical(f):
2     return f(bit[1](0b0)) \
3         ^ f(bit[1](0b1))
4
5 print('Balanced f:')
6 print('Classical: f(0) xor f(1) =',
7       naive_classical(balanced))
8 print('Quantum: f(0) xor f(1) =',
9       deutsch(balanced))
10
11 print('\nConstant f:')
12 print('Classical: f(0) xor f(1) =',
13       naive_classical(constant))
14 print('Quantum: f(0) xor f(1) =',
15       deutsch(constant))

```

(b) Python for testing Deutsch’s algorithm

Fig. 8. Deutsch’s algorithm in Qwerty

lines 11 and 15 of Fig. 8a) may be invoked classically from Python, which is useful for testing.

4.1.2 Deutsch–Jozsa algorithm. The Deutsch–Jozsa algorithm is a generalization of Deutsch’s algorithm (Section 4.1.1). The input remains a black-box classical function f , except f has N input bits instead of 1. f is assumed to be either constant or *balanced* (returning 0 for exactly half its domain) [10, 12, 30]; the algorithm determines whether f is constant or balanced using a single invocation of f .

Compared to line 4 of Fig. 8a, line 4 of Fig. 9a introduces the syntax $[N]$ in `@qpu[N]`. Here, N is a *dimension variable* allowing the programmer to specify dimensions in terms of N rather than hard-coding fixed numbers. For example, on line 7 of Fig. 9a, the syntax `'+'[N]` prepares N duplicate $|+\rangle$ states side-by-side. If $N = 3$, for instance, then this expression would expand to `'+'[3]` or equivalently `'+++'`.

However, the code in Fig. 9a never explicitly specifies N , because the compiler can infer N . Specifically, the portion `f: cfunc[N,1]` of line 5 of Fig. 9a instructs the compiler to infer N from the number of input bits of the capture `f`. (The type `cfunc[N,1]` is syntactic sugar for the more verbose `func[[bit[N]],bit[1]]`, which means a function whose arguments are only a `bit[N]` and whose result is a `bit[1]`.)

Lines 7–8 of Fig. 9a are the quantum portion of the algorithm. The pipeline shown does the following:

- (1) Prepare $|+\rangle^{\otimes N}$.
- (2) Apply $|x\rangle \mapsto (-1)^{f(x)}|x\rangle$, where $x \in \{0, 1\}^N$ (see Section 3.2.5).
- (3) Measure in the N -qubit **plus/minus** ($|+\rangle/|-\rangle$) basis, called **pm** in Qwerty.

In the last step, measuring $|+\rangle^{\otimes N}$ yields classical bits $00\dots 0$. (Measuring in **pm**[N] aligns with Deutsch and Jozsa’s original formulation [12].)

The `x.xor_reduce()` operation on line 24 of Fig. 9a XORs all bits of `x` together, producing a single-bit result.

Fig. 9b shows Python code for invoking the Qwerty Deutsch–Jozsa code written in Fig. 9a. For brevity, we assume all of Fig. 9 resides in the same Python module (`.py` file), but Fig. 9b could also import Fig. 9a if desired, as both can be typical Python modules.

```

1 from qwerty import *
2
3 def deutsch_jozsa(f):
4     @qpu[N](f)
5     def kernel(f: cfunc[N,1]) \
6         -> bit[N]:
7         return '+'[N] | f.phase \
8             | pm[N].measure
9
10    if int(kernel()) == 0:
11        return 'constant'
12    else:
13        return 'balanced'
14
15    @classical
16    def constant(x: bit[4]) -> bit:
17        # f(x) = 1
18        return bit[1](0b1)
19
20    @classical
21    def balanced(x: bit[4]) -> bit:
22        # f(x) = 1 for half the inputs
23        # and f(x) = 0 for the other half
24        return x.xor_reduce()

```

(a) Qwerty code for the Deutsch–Jozsa algorithm

```

1 def naive_classical(f, n_bits):
2     answers = [0, 0]
3     for i in range(2**(n_bits-1)+1):
4         answer = int(f(bit[n_bits](i)))
5         answers[answer] += 1
6
7     if 0 in answers:
8         return 'constant'
9     else:
10        return 'balanced'
11
12    print('Constant test:')
13    print('Classical:',
14        naive_classical(constant, 4))
15    print('Quantum:',
16        deutsch_jozsa(constant))
17
18    print('\nBalanced test:')
19    print('Classical:',
20        naive_classical(balanced, 4))
21    print('Quantum:',
22        deutsch_jozsa(balanced))

```

(b) Python for testing the Deutsch–Jozsa algorithm

Fig. 9. Deutsch–Jozsa algorithm in Qwerty

4.1.3 *Bernstein–Vazirani algorithm.* Section 3.1 describes Bernstein–Vazirani [5, 14], with Fig. 1 showing the steps of the algorithm.

Unlike the previous examples of Deutsch’s (Fig. 8a) and Deutsch–Jozsa (Fig. 9a), Fig. 10a shows the use of a *basis translation* on line 8. The syntax `pm[N] >> std[N]` performs a basis translation from the N -qubit **plus/minus** basis (`pm[N]`) to the N -qubit **standard** basis (`std[N]`). For each qubit, this performs the mapping $|+\rangle \mapsto |0\rangle$ and $|-\rangle \mapsto |1\rangle$.

Writing `pm` is shorthand for the basis literal `{'+', '-'}`, and similarly `std` is syntactic sugar for `{'0', '1'}`. Thus, `pm[N] >> std[N]` behaves identically to `{'+', '-'}[N] >> {'0', '1'}[N]`, a more verbose form where the element-wise translation `'+' ↦ '0'` and `'-' ↦ '1'` is more explicit.

The `[N]` suffix takes the repeated tensor product of an expression (such as a basis) N times. For example, if $N = 2$, then `{'0', '1'}[N]` becomes `{'0', '1'}+{'0', '1'}`. This `[N]` syntax applies to functions as well: the code `({'+', '-'} >> {'0', '1'})[N]` is also equivalent to the operation performed by line 8 of Fig. 10a, as would be `(pm >> std)[N]`.

Furthermore, lines 8 and 9 in Fig. 10a are exactly equivalent to `pm[N].measure` in Deutsch–Jozsa (line 8 of Fig. 9a). This is true because any `b.measure` where `b` is an N -qubit non-`std` basis compiles to `(b>>std[N]) | std[N].measure`.

Fig. 10b shows Python code for invoking the Qwerty code in Fig. 10a, including a naïve classical implementation (line 3) which requires N invocations of the black box `f` rather than just one. The classical code demonstrates that the `bit` class included in the Qwerty Python runtime can conveniently be sliced like a `list` (line 7 of Fig. 10b) or manipulated with bitwise operations like an `int` (line 8 of Fig. 10b).

```

1 from qwerty import *
2
3 def bv(f):
4     @qpu[N](f)
5     def kernel(f: cfunc[N,1]) \
6         -> bit[N]:
7         return '+'[N] | f.phase \
8             | pm[N] >> std[N] \
9             | std[N].measure
10
11     return kernel()
12
13 def get_black_box(secret_string):
14     @classical[N](secret_string)
15     def f(secret_string: bit[N],
16         x: bit[N]) -> bit:
17         return (secret_string & x) \
18             .xor_reduce()
19
20     return f

```

(a) Qwerty code for the Bernstein–Vazirani algorithm

```

1 import sys
2
3 def naive_classical(f, n_bits):
4     secret_found = bit[n_bits](0b0)
5     x = bit[n_bits](0b1 << (n_bits-1))
6     for i in range(n_bits):
7         secret_found[i] = f(x)
8         x = x >> 1
9     return secret_found
10
11 secret_str = \
12     bit.from_str(sys.argv[1])
13 n_bits = len(secret_str)
14 black_box = get_black_box(secret_str)
15
16 print('Classical:',
17     naive_classical(black_box,
18                     n_bits))
19 print('Quantum:', bv(black_box))

```

(b) Python for testing the Bernstein–Vazirani algorithm

Fig. 10. Bernstein–Vazirani algorithm in Qwerty

4.1.4 Period finding. Given a black box function $f : \{0, 1\}^M \rightarrow \{0, 1\}^N$ as input, the quantum period finding algorithm returns the smallest positive r such that $f(x) = f(x + r)$, i.e., the period of f [29, 30].

Lines 8-11 of Fig. 11a specify the quantum subroutine. The pipeline does the following:

- (1) Prepare $|+\rangle^{\otimes M} |0\rangle^{\otimes N}$.
- (2) Apply $|x\rangle |y\rangle \mapsto |x\rangle |y \oplus f(x)\rangle$. (Observe the XOR, \oplus , that is the namesake for the syntax `f.xor_embed` used.)
- (3) Measure the first register in the M -qubit Fourier basis [30, §5.1] and discard the second register.

By explicitly requesting to project the first register to an M -qubit Fourier basis state, line 10 is more expressive than gate-oriented code, which would manually invoke the inverse quantum Fourier transform (IQFT) and measure in the $|0\rangle/|1\rangle$ basis. By contrast, the Qwerty compiler automatically synthesizes the same code (specifically `fourier[N]>>std[N]|std[N].measure`) as discussed in Section 4.1.3.

Qwerty allows taking tensor products of functions, such as `fourier[M].measure` (line 10 of Fig. 11a) and `discard[N]` (line 11 of Fig. 11a). Calling the resulting function, written as `fourier[M].measure + discard[N]`, does the following: (1) splits the input tuple of $M + N$ qubits into two tuples of M and N bits, respectively; (2) runs both functions independently; and (3) merges their result tuples: in this example, the tuple with M bits (returned by `fourier[M].measure`) is concatenated with the empty tuple $()$ (returned by `discard[N]`). The resulting value is a tuple of M bits, which explains why the return value of `kernel()` written on line 7 of Fig. 11a is `bit[M]`.

The Qwerty Python runtime contains some convenience functions for post-processing. For example, `bit[M].as_bin_frac()` (used on lines 14 and 15 of Fig. 11a) returns a Python `Fraction` instance [39] representing the bits interpreted as a binary fraction [30, §5.1].

Fig. 11b shows classical Python code that invokes the Qwerty code in Fig. 11a. The particular f used in this example (line 24 of Fig. 11a)

```

1 import math
2 from qwerty import *
3
4 def period_finding(black_box):
5     @qpu[M,N](black_box)
6     def kernel(black_box: cfunc[M,N]) \
7         -> bit[M]:
8         return '+'[M] + '0'[N] \
9             | black_box.xor_embed \
10            | fourier[M].measure \
11              + discard[N]
12
13 result1, result2 = kernel(shots=2)
14 l_over_r1 = result1.as_bin_frac()
15 l_over_r2 = result2.as_bin_frac()
16 r = math.lcm(l_over_r1.denominator,
17              l_over_r2.denominator)
18 return r
19
20 def get_black_box(n_bits_in,
21                  n_bits_out,
22                  n_mask_bits):
23     @classical[M,N,K]
24     def f(x: bit[M]) -> bit[N]:
25         return bit[N-K](0b0), x[M-K:]
26
27     return f[[n_bits_in,
28              n_bits_out,
29              n_mask_bits]]

```

(a) Qwerty code for period finding

```

1 n_bits_in = int(sys.argv[1])
2 n_bits_out = int(sys.argv[2])
3 n_masked = int(sys.argv[3])
4 black_box = get_black_box(n_bits_in,
5                            n_bits_out,
6                            n_masked)
7 if period_finding(black_box) \
8     == 2**n_masked:
9     print('success!')
10 else:
11     print('oops...')

```

(b) Python for testing period finding

Fig. 11. Period finding in Qwerty

returns the zero-extended lower K bits of the input string, making the period 2^K . The `[[...]]` syntax on lines 27-29 of Fig. 11a instantiates f with its dimension variables M , N , and K set to the desired values (see Section 3.2.4).

4.1.5 Simon’s algorithm. Suppose a 2-to-1 classical function $f : \{0, 1\}^N \rightarrow \{0, 1\}^N$ satisfies $f(x) = f(y) \Rightarrow x = y \oplus s$ when $x \neq y$. Given a black-box implementation of f , Simon’s algorithm returns the secret string s [47, 48].

Fig. 12a is the first example shown making use of `id`, the identity operation on a qubit. It has the type of a function from one qubit to one qubit, so `id[N]` has the type of a function from `qubit[N]` to `qubit[N]`.

Also in contrast with previous examples presented, Simon’s algorithm requires multiple invocations of the quantum kernel (`kernel()`) on lines 6-13 of Fig. 12a). Specifically, each nonzero measurement becomes a row of an $(N-1) \times N$ matrix of bits (lines 15-22 of Fig. 12a). For the sake of brevity, we relegate the Python code performing Gaussian elimination on this matrix to a different module not shown here (imported on line 2 of Fig. 12a). This post-processing can be implemented efficiently using existing highly-optimized Python libraries such as `numpy` [21].

The whole algorithm may need to be retried if this classical post-processing fails; this is accomplished by `simon_post()` throwing a `Retry` exception and line 25 catching it. Ordinary Python exception handling works here, with no new quantum-specific features needed.

The classical logic on lines 32-34 of Fig. 12a was determined by writing out the truth table for an example function obeying the property stated at the beginning of this section and solving 3 Karnaugh maps. Writing this black box as classical logic only once is more convenient than hand-writing separate quantum and classical oracles and manually proving their behavior matches.

The Python code invoking Simon’s in Fig. 12b includes a classical solution that demonstrates — at least informally — the exponential speedup of Simon’s algorithm over classical algorithms [47]: compare the $O(2^N)$ classical loop on line 3 of Fig. 12b with the polynomial-time quantum code (Fig. 12a).

```

1 from qwerty import *
2 from simon_post import simon_post, \
3                         Retry
4
5 def simon(f):
6     @qpu[N](f)
7     def kernel(f: cfunc[N]) -> bit[N]:
8         return '+'[N] + '0'[N] \
9             | f.xor_embed \
10            | (std[N] >> pm[N]) \
11            + id[N] \
12            | std[N].measure \
13            + discard[N]
14
15     while True:
16         rows = []
17         while True:
18             row = kernel()
19             if int(row) != 0:
20                 rows.append(row)
21                 if len(rows) >= row.n_bits-1:
22                     break
23         try:
24             return simon_post(rows)
25         except Retry:
26             print('retrying...')
27             continue
28
29 @classical
30 def black_box(q: bit[3]) -> bit[3]:
31     return \
32         (~q[0]& q[2]|q[0]&~q[2]|~q[1],
33         ~q[0]&~q[2]|q[0]& q[2],
34         ~q[0]&~q[2]|q[0]& q[2]|~q[1])

```

(a) Qwerty code for Simon’s algorithm

```

1 def naive_classical(f, n_bits):
2     out_to_x = {}
3     for i in range(2**n_bits):
4         x = bit[n_bits](i)
5         out = f(x)
6         if out in out_to_x:
7             return x ^ out_to_x[out]
8         out_to_x[out] = x
9
10 print('Classical:',
11       naive_classical(black_box, 3))
12 print('Quantum:', simon(black_box))

```

(b) Python for testing Simon’s algorithm

Fig. 12. Simon’s algorithm in Qwerty

4.2 Factoring

The excitement sparked by Shor’s 1994 discovery of an efficient quantum algorithm for factoring integers [44, 45] still fuels interest in quantum computing today. This section shows how to factor in Qwerty using phase estimation as described by Cleve et al. [10] and Nielsen and Chuang [30].

4.2.1 Quantum phase estimation. Unlike previous examples, quantum phase estimation (QPE) is primarily a building block on which other algorithms are built. QPE finds an eigenvalue of a provided operator (up to some precision); specifically, provided a unitary U and state $|u\rangle$ such that $U|u\rangle = e^{i\varphi}|u\rangle$, QPE estimates φ [10, 30].

The parameter `prep_eigvec` (lines 3 and 7 of Fig. 13a) is a function responsible for preparing the eigenvector $|u\rangle$. The syntax `prep_eigvec()` used on line 10 of Fig. 13a is syntactic sugar for calling `prep_eigvec` with an empty tuple as an argument, i.e., `() | prep_eigvec`.

QPE repeatedly applies U raised to increasing powers of 2, i.e., U^{2^0} , then U^{2^1} , then U^{2^2} , and so on. An efficient U will effect this exponentiation without exponential cost (e.g., line 15 of Fig. 13b). The Qwerty formulation facilitates this by making the exponent j of U^{2^j} a dimension variable of `op` (see Section 3.2.4), specifically a *free* dimension variable. The syntax `[[...]]` on line 8 of Fig. 13a indicates that `op` (U) has a free dimension variable, and line 12 uses `[[j]]` to instantiate `op` with the free dimension variable set to j , thus instantiating U^{2^j} efficiently. Lines 10-15 of Fig. 13b show how `op` could be implemented – note that `J` is a free dimension variable.

To achieve repeated applications of U^{2^j} without hard-coding some fixed number of them, Qwerty supports loop-like repeated applications as written in line 11-13 in Fig. 13a. The point of these repeated applications is to approximate a Fourier basis state which line 14 of Fig. 13a will identify. Yet to accomplish this, U^{2^j} must be applied in different subspaces. Because `op` (U) is a black box, though, typical syntax for restricting a basis translation to a subspace (e.g., Fig. 3d) is not applicable. Thus, as discussed in Section 3.2.3,

```

1 from qwerty import *
2
3 def qpe(precision, prep_eigvec, op,
4       n_shots):
5     @qpu[M,T](prep_eigvec, op)
6     def kernel(
7         prep_eigvec: qfunc[0,M],
8         op: rev_qfunc[M][[...]]) \
9         -> bit[T]:
10        return '+'[T] + prep_eigvec() \
11              | (std[T-1-j]+'1'+std[j]
12                & op[[j]]
13                  for j in range(T)) \
14              | fourier[T].measure \
15                + discard[M]
16
17 k_inst = kernel[[precision]]
18 for meas in k_inst(shots=n_shots):
19     yield meas.as_bin_frac()

```

(a) Qwerty code for quantum phase estimation

```

1 import sys
2
3 phi = float(sys.argv[1])
4 precision = int(sys.argv[2])
5
6 @qpu
7 def prep1() -> qubit:
8     return '1'
9
10 @qpu[J](phi)
11 @reversible
12 def rot(phi: angle,
13        q: qubit) -> qubit:
14     return q | std >> \
15           {'0', phase(2*pi*phi*2**J)*'1'}
16
17 print('Expected:', phi)
18 phi_got, = qpe(precision, prep1, rot,
19               n_shots=1)
20 print('Actual:', float(phi_got))

```

(b) Qwerty for testing quantum phase estimation

Fig. 13. Quantum phase estimation in Qwerty

the operator `&` used in lines 11-12 of Fig. 13a runs U^{2^j} in individual `'1'` subspaces from right to left. Note that the `std[...]`s on line 11 are effectively padding, since running in both the `'0'` and `'1'` subspaces means running on the whole space.

Fig. 13b shows an example of using QPE.

4.2.2 Order finding. A key ingredient of Shor’s factoring algorithm, quantum order finding determines the multiplicative order of x modulo N , which is defined as the smallest positive integer r such that $x^r \equiv 1 \pmod{N}$. Here, x and N are coprime positive integers [30, 42].

Although it is possible to implement order finding using period finding (Section 4.1.4) instead [30, 44, 45], the Qwerty code in Fig. 15 calls out to QPE (Section 4.2.1) on line 22. The operator `mult` passed to QPE is an in-place multiplier $|y\rangle \mapsto |xy \bmod N\rangle$ (line 21), and the eigenvector is an intricate superposition that (incredibly) is equal to $|00 \cdots 0\rangle \otimes |1\rangle$ (lines 10-12 in Fig. 15) [10, 30]. (The calculations for necessary QPE precision and qubit count on lines 6-8 of Fig. 15 are due to Nielsen and Chuang [30].)

The in-place multiplier is straightforward to define in Qwerty. Lines 14-16 of Fig. 15 define a classical function `xymodN` that multiplies its input y times x^{2^j} modulo N (x , j , and N are dimension variables defined on line 14). Assuming the input y is already a least positive residue modulo N , `xymodN` is reversible because it permutes the least positive residues modulo N . Thus, it can be instantiated in-place as desired (line 21 of Fig. 15). As noted in Section 3.2.5, though, Qwerty currently requires a reverse implementation as well. Thankfully, undoing the multiplication by x only means multiplying by the modular

```

1 import sys
2
3 def naive_classical(x, modN):
4     for r in range(1, modN):
5         if x**r % modN == 1:
6             return r
7
8 err = 0.2
9 x = int(sys.argv[1])
10 modN = int(sys.argv[2])
11 if math.gcd(x, modN) != 1:
12     raise ValueError('invalid x, modN')
13
14 print('Classical:',
15       naive_classical(x, modN))
16 print('Quantum:',
17       order_finding(err, x, modN))

```

Fig. 14. Python for testing order finding

```

1 import math
2 from qwerty import *
3 from qpe import qpe
4
5 def order_finding(epsilon, x, modN):
6     L = math.ceil(math.log2(modN))
7     t = 2*L + 1 + math.ceil(
8         math.log2(2+1/(2*epsilon)))
9
10 @qpu[M]
11 def one() -> qubit[M]:
12     return '0'[M-1] + '1'
13
14 @classical[X,N,M,J]
15 def xymodN(y: bit[M]) -> bit[M]:
16     return X**2**J * y % N
17
18 x_inv = pow(x, -1, modN)
19 fwd = xymodN[[x,modN,L,...]]
20 rev = xymodN[[x_inv,modN,L,...]]
21 mult = fwd.inplace(rev)
22 frac1, frac2 = qpe(t, one, mult, 2)
23
24 def denom(frac):
25     cf = cfraction.from_fraction(frac)
26     for c in reversed(
27         cf.convergents()):
28         if c.denominator < modN:
29             return c.denominator
30
31 return math.lcm(denom(frac1),
32                denom(frac2))

```

Fig. 15. Qwerty code for order finding

inverse x^{-1} instead. Lines 19 and 20 instantiate the forward and reverse multipliers, respectively (see Section 3.2.4). The ellipses `...` on each line ask Qwerty to leave J as a free dimension variable as `qpe()` expects (Section 4.2.1).

The classical post-processing makes use of the `cfraction` (continued fraction) type included in the Qwerty Python runtime to convert the `fractions.Fraction` [39] returned by `qpe()` to a continued fraction and choose a convergent. Choosing the last convergent whose denominator is less than N (lines 24-29 of Fig. 15) is due to Watkins [54], and taking the least common multiple (line 31 of Fig. 15) is due to Nielsen and Chuang [30].

Fig. 14 shows some Python code for invoking the `order_finding()` subroutine from Fig. 15.

4.2.3 Shor’s algorithm. Shor’s algorithm finds a nontrivial factor of a positive integer N (i.e., a factor other than 1 or N). It achieves this using a reduction from factoring to order finding (Section 4.2.2) [30, 45]. In other words, Shor’s algorithm is typically classical code that calls out to a quantum order finding subroutine (Section 4.2.2). Consequently, Fig. 16a is purely Python code, although it calls `order_finding()` (Fig. 15) on line 18.

Because it consists entirely of Python, this implementation of Shor’s itself resembles the pseudocode from Nielsen and Chuang [30, §5.3.2]. Also in Python is Fig. 16b, an example command-line program invoking the algorithm.

```

1 import math
2 import random
3 from qwerty import *
4
5 from order_finding import \
6     order_finding
7
8 def shors(epsilon, num):
9     if num % 2 == 0:
10        return 2
11
12    x = random.randint(2, num-1)
13    if (y := math.gcd(x, num)) > 1:
14        print('Got lucky! Skipping '
15              'quantum subroutine...')
16        return y
17
18    r = order_finding(epsilon, x, num)
19
20    if r % 2 == 0 \
21        and pow(x, r//2, num) != -1:
22        if (gcd := math.gcd(x**(r//2)-1,
23                            num)) > 1:
24            return gcd
25        if (gcd := math.gcd(x**(r//2)+1,
26                            num)) > 1:
27            return gcd
28
29    raise Exception("Shor's failed")

```

(a) Python code for Shor’s algorithm

```

1 import sys
2
3 err = 0.2
4 num = int(sys.argv[1])
5 print('Nontrivial factor of', num,
6       'is', shors(err, num))

```

(b) Python for testing Shor’s algorithm

Fig. 16. Shor’s algorithm in Qwerty

4.3 Search

Often, a strategy to take advantage of program structure using quantum mechanical properties is unknown — this is where unstructured search is most useful [1]. This section shows how search can be implemented in Qwerty and ends with an example of unstructured search in action.

4.3.1 Grover’s algorithm. Given a black box implementing $f : \{0, 1\}^N \rightarrow \{0, 1\}$ (i.e., an oracle), Grover’s algorithm finds all $x \in \{0, 1\}^N$ such that $f(x) = 1$ (i.e., all answers to the oracle). The algorithm consists of many iterations, each of which consists of the oracle followed by a special reflection called the Grover diffuser. Applying the right number of iterations is crucial and depends on the number of answers [7, 19, 20, 30].

A single Grover iteration is shown on lines 5-10 of Fig. 17a. Line 9 of Fig. 17a applies the transformation $|x\rangle \mapsto (-1)^{f(x)} |x\rangle$ (see Section 3.2.5), and line 10 performs the diffuser from Fig. 3d.

Lines 16-17 of Fig. 17a apply I iterations of Grover’s starting with the equal superposition $|+\rangle[N]$. (In Python, $_$ is common variable name for an unused value, such as the loop variable in this case.) The `[[n_iter]]` syntax on line 20 of Fig. 17a instantiates `kernel()` with the proper number of iterations, which may be calculated with the Python code starting at line 25 of Fig. 17a (an implementation of a formula due to Nielsen and Chuang [30, §6.1.4]). The measurements are post-processed on lines 22-23 of Fig. 17a by invoking the oracle classically to filter out incorrect output.

Fig. 17b shows an example of running Grover’s algorithm. The oracle `all_ones()` on lines 3-5 of Fig. 17b ANDs all bits of the input x together, thus outputting 1 only for the input $11 \cdots 1$.

```

1 import math
2 from qwerty import *
3
4 def grover(oracle, n_iter, n_shots):
5     @qpu[N](oracle)
6     def grover_iter(oracle: cfunc[N,1],
7                    q: qubit[N]) \
8         -> qubit[N]:
9         return q | oracle.phase \
10            | -('+'[N] >> -'+'[N])
11
12     @qpu[N,I](grover_iter)
13     def kernel(grover_iter: qfunc[N]) \
14         -> bit[N]:
15         return \
16             '+'[N] | (grover_iter
17                    for _ in range(I)) \
18                    | std[N].measure
19
20     kern_inst = kernel[[n_iter]]
21     results = kern_inst(shots=n_shots)
22     return {r for r in set(results)
23            if oracle(r)}
24
25 def get_n_iter(n_qubits, n_answers):
26     n = 2**n_qubits
27     m = n_answers
28     theta = 2*math.acos(
29         math.sqrt((n-m)/n))
30     rnd = lambda x: math.ceil(x-0.5)
31     return rnd(math.acos(
32         math.sqrt(m/n))/theta)

```

(a) Qwerty code for Grover’s algorithm

```

1 import sys
2
3 @classical[N]
4 def all_ones(x: bit[N]) -> bit:
5     return x.and_reduce()
6 n_ans = 1
7
8 n_qubits = int(sys.argv[1])
9 oracle = all_ones[[n_qubits]]
10 n_iter = get_n_iter(n_qubits, n_ans)
11 answers = grover(oracle, n_iter,
12                n_shots=32)
13 for answer in answers:
14     print(answer)

```

(b) Qwerty for testing Grover’s algorithm

Fig. 17. Grover’s algorithm in Qwerty

4.3.2 Fixed-point amplitude amplification. Despite its historical significance, Grover’s algorithm suffers from practical difficulties: first, it may malfunction if more than half of the search space are answers [30]. Second, implementers are struck with a “soufflé problem” [8]: it is easy to ‘overcook’ the delicate state by applying too many Grover iterations or ‘undercook’ it by applying too few. Applying the right number of iterations requires knowing the number of answers, which may be impractical. Yoder, Low, and Chuang propose a fixed-point search algorithm [55] that addresses these issues.

The algorithm is perhaps most easily understood as a special case of the quantum singular value transform (QSVT) [16, 26]. Lines 18-21 of Fig. 19 rotate around the space of answers, and lines 22-27 rotate around the initial state (which a prepares). The particular rotation angles are quite technical and depend on the lower bound for the number of answers and the acceptable error; the separate `fix_pt_phases` module (imported on line 2 of Fig 19) uses the existing `pyqsp` Python library to generate these phases [26].

```

1 import sys
2
3 @qpu[N]
4 @reversible
5 def a(q: qubit[N]) -> qubit[N]:
6     return q | '+'[N].prep
7
8 @classical[N]
9 def oracle_(x: bit[N]) -> bit:
10    return ~x[:N-1].and_reduce()
11
12 n_qubits = int(sys.argv[1])
13 oracle = oracle_[[n_qubits]]
14 orig_prob = 1/2**n_qubits
15 res = fix_pt_amp(a, oracle,
16                 orig_prob, 0.98,
17                 histogram=True)
18 print_histogram(res)
19 # Prints:
20 # 00 -> 48.93%
21 # 01 -> 49.46%
22 # 10 -> 0.44%
23 # 11 -> 1.17%
```

Fig. 18. Qwerty for testing fixed-point amplitude amplification

```

1 from qwerty import *
2 from fix_pt_phases import get_phases
3
4 def fix_pt_amp(a, oracle, orig_prob,
5               new_prob=0.98,
6               n_shots=2048,
7               histogram=False):
8     phis = get_phases(orig_prob,
9                       new_prob)
10
11 @qpu[N,K,D](phis, a, oracle)
12 def amp_iter(phis: angle[2*D],
13             a: rev_qfunc[N],
14             oracle: cfunc[N,1],
15             q: qubit[N+1]) \
16             -> qubit[N+1]:
17     return \
18         q | oracle.xor_embed \
19           | id[N] + \
20           | std.rotate(phis[[2*K]]) \
21           | oracle.xor_embed \
22           | ~a + id \
23           | '0'[N] & std.flip \
24           | id[N] + \
25           | std.rotate(phis[[2*K+1]]) \
26           | '0'[N] & std.flip \
27           | a + id
28
29 @qpu[N,D](phis, a, amp_iter)
30 def kernel(
31     phis: angle[2*D], a: qfunc[N],
32     amp_iter: qfunc[N+1][[...]]) \
33     -> bit[N]:
34     return '0'[N+1] \
35           | a + id \
36           | (amp_iter[[k]]
37             for k in range(D)) \
38           | std[N].measure + discard
39
40 return kernel(shots=n_shots,
41              histogram=histogram)
```

Fig. 19. Qwerty code for fixed-point amplitude amplification

Line 22 of Fig. 19 uses `~a` to un-prepare the initial state. The `@reversible` decorator on line 4 of Fig. 18 facilitates this by requiring that `a()` has no irreversible operations (Section 3.2.3).

Fig. 18 shows a test invocation of the algorithm with an oracle identifying states whose first $N-1$ bits are *not* all 1s. The original probability on line 14 is a drastic under-estimate, yet the result is not overcooked (lines 20-23 of Fig. 18).

4.3.3 *Niroula–Nam substring matching.* Niroula and Nam propose a quantum algorithm capable of finding indices of a substring in $O(\sqrt{N}(\log^2 N + \log M))$ time (compare with the best known classical time complexity $\Theta(N + M)$) [31]. The algorithm works by preparing a superposition of all possible cyclic bit rotations of the haystack string and amplifying cases where the needle matches the beginning of the haystack. Measuring the rotation offset register yields the indices.

Fig. 20a shows an implementation of substring matching in Qwerty. (It assumes the length of both the string and pattern are powers of 2.) The $K(k)$ syntax seen on line 9 is syntactic sugar for explicitly setting a dimension variable K using a Python `int` variable named k ; this is equivalent to using `[[...]]` when inference fails, as seen on e.g. line 13 of Fig. 18.

The `shift_and_cmp()` operation (lines 9-16 of Fig. 20a) is the heart of Niroula–Nam: taking in the three registers of the algorithm, it cyclically left-shifts the haystack as mentioned and XORs the beginning of it with the needle (line 16). If the needle was found, then the last portion of the output of `shift_and_cmp()` should be all zeros. This is precisely what the oracle (lines 31-35 of Fig. 20a) is built to identify.

This example uses fixed-point amplitude amplification (Section 4.3.2) to amplify the cases where the substring is found (line 37 of Fig. 20a). The input `a`, defined on lines 18-29 of Fig. 20a, prepares the state on which to perform the amplification assuming the input is $|00\dots 0\rangle$. The `.prep` keyword described in Section 3.2.2 is useful here (lines 26-27) to encode the needle and haystack as qubits. `shift_and_cmp()` is then executed in-place (Section 3.2.5). (`shift_and_cmp` is written a second time on line 29 because it is its own inverse.)

For convenience, Qwerty allows the operand of `.prep` (Section 3.2.2) to be a `bit[N]`, as in `pat.prep` on line 27 of Fig. 20a. If `pat` were `1011`, for example, then `pat.prep` would behave exactly as `'1011'.prep`.

Fig. 20b calls `match()` (from Fig. 20a) inside typical Python code. Line 7 shows an abbreviated example output given an example input.

```

1 import math
2 from qwerty import *
3 from fix_pt_amp import fix_pt_amp
4
5 def match(string, pat):
6     n, m = len(string), len(pat)
7     k = math.ceil(math.log2(n))
8
9     @classical[K(k),N(n),M(m)]
10    def shift_and_cmp(off: bit[K],
11                      string: bit[N],
12                      pat: bit[M]) \
13                      -> bit[K+N+M]:
14        return off, \
15               string, \
16               string.rotl(off)[:M] ^ pat
17
18    @qpu[K(k),N,M](string, pat,
19                  shift_and_cmp)
20    @reversible
21    def a(string: bit[N], pat: bit[M],
22          shift_and_cmp: cfunc[K+N+M],
23          q: qubit[K+N+M]) \
24          -> qubit[K+N+M]:
25        return \
26            q | '+'[K].prep + string.prep \
27              + pat.prep \
28              | shift_and_cmp \
29              .inplace(shift_and_cmp)
30
31    @classical[K(k),N(n),M(m)]
32    def oracle(off: bit[K],
33              string: bit[N],
34              pat: bit[M]) -> bit:
35        return (~pat).and_reduce()
36
37    ret = fix_pt_amp(a, oracle, 1/n)
38    return {int(result[:k])
39            for result in set(ret)
40            if oracle(result)}

```

(a) Qwerty code for Niroula–Nam substring matching

```

1 import sys
2 string = bit.from_str(sys.argv[1])
3 pat = bit.from_str(sys.argv[2])
4 print('Matching indices:')
5 for index in match(string, pat):
6     print(index)
7 # Output for inputs 1010 and 10: 0, 2

```

(b) Python for testing Niroula–Nam substring matching

Fig. 20. Niroula–Nam substring matching in Qwerty

5 CONCLUSION

Qwerty is focused on algorithms for quantum information processing as opposed to quantum physical system modeling (e.g., Hamiltonian simulation) [15, 25]. Our goal in creating Qwerty is to help non-experts reason about quantum computation while abstracting away the complexity of gate engineering. Qwerty achieves this through abstractions such as qubit literals based on a string analogy, embedding of classical functions in quantum kernels, and particularly the **basis** type. These constructs provide a programmer with a rich suite of primitives to realize algorithms as code. Embedding Qwerty in Python achieves both approachability and convenience for the classical component of quantum-classical programs. This is demonstrated through Qwerty implementations of significant quantum algorithms such as Grover’s and Shor’s.

ACKNOWLEDGMENTS

The authors thank Elton Pinto and Eugene Dumitrescu for helpful discussions on quantum programming language design. We acknowledge support for this work from NSF planning grant #2016666, “Enabling Quantum Computer Science and Engineering” and through the ORNL STAQCS project. This research was supported in part through research infrastructure and services provided by the Rogues Gallery testbed [56] hosted by the Center for Research into Novel Computing Hierarchies (CRNCH) at Georgia Tech. The Rogues Gallery testbed is primarily supported by the National Science Foundation (NSF) under NSF Award Number #2016701. Any opinions, findings and conclusions, or recommendations expressed in this material are those of the author(s), and do not necessarily reflect those of the NSF.

REFERENCES

- [1] Scott Aaronson. 2022. *How Much Structure Is Needed for Huge Quantum Speedups?* Technical Report arXiv:2209.06930. arXiv. <http://arxiv.org/abs/2209.06930> arXiv:2209.06930 [quant-ph].
- [2] Ali Javadi Abhari, Arvin Faruque, Mohammad Javad Dousti, Lukas Svec, Oana Catu, Amlan Chakrabati, Chen-Fu Chiang, Seth Vanderwilt, John Black, Fred Chong, Margaret Martonosi, Martin Suchara, Ken Brown, Massoud Pedram, and Todd Brun. 2012. *Scaffold: Quantum Programming Language*. Technical Report. Princeton University Department of Computer Science. 43 pages.
- [3] Sheldon Axler. 2023. *Linear Algebra Done Right* (4th ed. 2024 edition ed.). Springer, Cham, Switzerland.
- [4] Charles H. Bennett. 1989. Time/Space Trade-Offs for Reversible Computation. *SIAM J. Comput.* 18, 4 (Aug. 1989), 766–776. <https://doi.org/10.1137/0218053> Publisher: Society for Industrial and Applied Mathematics.
- [5] Ethan Bernstein and Umesh Vazirani. 1993. Quantum complexity theory. In *Proceedings of the twenty-fifth annual ACM symposium on Theory of Computing (STOC '93)*. Association for Computing Machinery, New York, NY, USA, 11–20. <https://doi.org/10.1145/167088.167097>
- [6] Benjamin Bichsel, Maximilian Baader, Timon Gehr, and Martin Vechev. 2020. Silq: a high-level quantum language with safe uncomputation and intuitive semantics. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 286–300. <https://doi.org/10.1145/3385412.3386007>
- [7] Michel Boyer, Gilles Brassard, Peter Høyer, and Alain Tapp. 1998. Tight Bounds on Quantum Searching. *Fortschritte der Physik* 46, 4-5 (1998), 493–505. [https://doi.org/10.1002/\(SICI\)1521-3978\(199806\)46:4/5<493::AID-PROP493>3.0.CO;2-P](https://doi.org/10.1002/(SICI)1521-3978(199806)46:4/5<493::AID-PROP493>3.0.CO;2-P)
- [8] Gilles Brassard. 1997. Searching a Quantum Phone Book. *Science* 275, 5300 (Jan. 1997), 627–628. <https://doi.org/10.1126/science.275.5300.627> Publisher: American Association for the Advancement of Science.
- [9] Jacques Carette, Chris Heunen, Robin Kaarsgaard, and Amr Sabry. 2023. The Quantum Effect: A Recipe for QuantumPi. <https://doi.org/10.48550/arXiv.2302.01885> arXiv:2302.01885 [quant-ph].
- [10] R. Cleve, A. Ekert, C. Macchiavello, and M. Mosca. 1998. Quantum algorithms revisited. *Proceedings of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences* 454, 1969 (Jan. 1998), 339–354. <https://doi.org/10.1098/rspa.1998.0164> Publisher: Royal Society.
- [11] Andrew Cross, Ali Javadi-Abhari, Thomas Alexander, Niel De Beaudrap, Lev S. Bishop, Steven Heidel, Colm A. Ryan, Prasad Sivarajah, John Smolin, Jay M. Gambetta, and Blake R. Johnson. 2022. OpenQASM 3: A Broader and Deeper Quantum Assembly Language. *ACM Transactions on Quantum Computing* 3, 3 (Sept. 2022), 12:1–12:50. <https://doi.org/10.1145/3505636>

- [12] David Deutsch and Richard Jozsa. 1997. Rapid solution of problems by quantum computation. *Proceedings of the Royal Society of London. Series A: Mathematical and Physical Sciences* 439, 1907 (Jan. 1997), 553–558. <https://doi.org/10.1098/rspa.1992.0167>
- [13] David Deutsch and Roger Penrose. 1997. Quantum theory, the Church–Turing principle and the universal quantum computer. *Proceedings of the Royal Society of London. A. Mathematical and Physical Sciences* 400, 1818 (Jan. 1997), 97–117. <https://doi.org/10.1098/rspa.1985.0070> Publisher: Royal Society.
- [14] S. D. Fallek, C. D. Herold, B. J. McMahon, K. M. Maller, K. R. Brown, and J. M. Amini. 2016. Transport implementation of the Bernstein–Vazirani algorithm with ion qubits. *New Journal of Physics* 18, 8 (Aug. 2016), 083030. <https://doi.org/10.1088/1367-2630/18/8/083030>
- [15] Richard P. Feynman. 1982. Simulating physics with computers. *International Journal of Theoretical Physics* 21, 6 (June 1982), 467–488. <https://doi.org/10.1007/BF02650179>
- [16] András Gilyén, Yuan Su, Guang Hao Low, and Nathan Wiebe. 2019. *Quantum singular value transformation and beyond: exponential improvements for quantum matrix arithmetics*. Technical Report. arXiv. 193–204 pages. <http://arxiv.org/abs/1806.01838> arXiv:1806.01838 [quant-ph].
- [17] Alexander S. Green, Peter LeFanu Lumsdaine, Neil J. Ross, Peter Selinger, and Benoît Valiron. 2013. Quipper: a scalable quantum programming language. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 333–342. <https://doi.org/10.1145/2491956.2462177>
- [18] Daniel M. Greenberger, Michael A. Horne, and Anton Zeilinger. 2007. Going Beyond Bell’s Theorem. *arXiv:0712.0921 [quant-ph]* (Dec. 2007). <http://arxiv.org/abs/0712.0921> arXiv: 0712.0921.
- [19] Lov K. Grover. 1996. A fast quantum mechanical algorithm for database search. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of Computing (STOC '96)*. Association for Computing Machinery, New York, NY, USA, 212–219. <https://doi.org/10.1145/237814.237866>
- [20] Lov K. Grover. 1997. Quantum Mechanics Helps in Searching for a Needle in a Haystack. *Physical Review Letters* 79, 2 (July 1997), 325–328. <https://doi.org/10.1103/PhysRevLett.79.325> Publisher: American Physical Society.
- [21] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. 2020. Array programming with NumPy. *Nature* 585, 7825 (Sept. 2020), 357–362. <https://doi.org/10.1038/s41586-020-2649-2>
- [22] Liyi Li, Finn Voichick, Kesha Hietala, Yuxiang Peng, Xiaodi Wu, and Michael Hicks. 2022. Verified compilation of Quantum oracles. *Proceedings of the ACM on Programming Languages* 6, OOPSLA2 (Oct. 2022), 146:589–146:615. <https://doi.org/10.1145/3563309>
- [23] Andrew Litteken, Yung-Ching Fan, Devina Singh, Margaret Martonosi, and Frederic T. Chong. 2020. An updated LLVM-based quantum research compiler with further OpenQASM support. *Quantum Science and Technology* 5, 3 (May 2020), 034013. <https://doi.org/10.1088/2058-9565/ab8c2c>
- [24] Thomas Lubinski, Cassandra Granade, Amos Anderson, Alan Geller, Martin Roetteler, Andrei Petrenko, and Bettina Heim. 2022. Advancing hybrid quantum–classical computation with real-time execution. *Frontiers in Physics* 10 (2022), 940293. <https://www.frontiersin.org/articles/10.3389/fphy.2022.940293>
- [25] Yuri Manin. 1980. Computable and uncomputable. Soviet Radio Publishing House.
- [26] John M. Martyn, Zane M. Rossi, Andrew K. Tan, and Isaac L. Chuang. 2021. Grand Unification of Quantum Algorithms. *PRX Quantum* 2, 4 (Dec. 2021), 040203. <https://doi.org/10.1103/PRXQuantum.2.040203>
- [27] Alexander Mccaskey, Thien Nguyen, Anthony Santana, Daniel Claudino, Tyler Kharazi, and Hal Finkel. 2021. Extending C++ for Heterogeneous Quantum-Classical Computing. *ACM Transactions on Quantum Computing* 2, 2 (July 2021), 6:1–6:36. <https://doi.org/10.1145/3462670>
- [28] Tiffany M. Mintz, Alexander J. McCaskey, Eugene F. Dumitrescu, Shirley V. Moore, Sarah Powers, and Pavel Lougovski. 2020. QCOR: A Language Extension Specification for the Heterogeneous Quantum-Classical Model of Computation. *ACM Journal on Emerging Technologies in Computing Systems* 16, 2 (March 2020), 22:1–22:17. <https://doi.org/10.1145/3380964>
- [29] Michele Mosca. 1999. *Quantum Computer Algorithms*. Ph. D. Dissertation. University of Oxford.
- [30] Michael A. Nielsen and Isaac L. Chuang. 2010. *Quantum Computation and Quantum Information: 10th Anniversary Edition* (1st edition ed.). Cambridge University Press, Cambridge ; New York.
- [31] Pradeep Niroula and Yunseong Nam. 2021. A quantum algorithm for string matching. *npj Quantum Information* 7, 1 (Feb. 2021), 1–5. <https://doi.org/10.1038/s41534-021-00369-3>
- [32] Santiago Núñez-Corrales, Marcos Frenkel, and Bruno Abreu. 2023. quAPL: Modeling Quantum Computation in an Array Programming Language. In *2023 IEEE International Conference on Quantum Computing and Engineering (QCE)*.

- IEEE, Bellevue, WA, USA, 1001–1012. <https://doi.org/10.1109/QCE57702.2023.00114>
- [33] Bernhard Ömer. 2000. *Quantum Programming in QCL*. Ph. D. Dissertation. Technical University of Vienna.
- [34] Anouk Paradis, Benjamin Bichsel, Samuel Steffen, and Martin Vechev. 2021. Unqomp: synthesizing uncomputation in Quantum circuits. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 222–236. <https://doi.org/10.1145/3453483.3454040>
- [35] Jennifer Paykin, Robert Rand, and Steve Zdancewic. 2017. QWIRE: a core language for quantum circuits. *ACM SIGPLAN Notices* 52, 1 (Jan. 2017), 846–858. <https://doi.org/10.1145/3093333.3009894>
- [36] Andres Paz. 2023. Aleph. <https://github.com/anpaz/aleph>
- [37] Andres Paz, Cassandra Granade, Mathias Soeken, Guen Prawiroatmodjo, Dmitry Vasilevsky, Alex Hansen, and Cesar Cortes. 2023. Sample: Bernstein-Vazirani algorithm. <https://github.com/microsoft/qsharp/blob/5a40497f/samples/algorithms/BernsteinVazirani.qs>
- [38] Elton Pinto. 2023. Neko: A quantum map-filter-reduce programming language. In *Student Research Competition (SRC) (Symposium on Principles of Programming Languages (POPL '23))*. Boston, MA, USA. <https://www.eltonpinto.me/assets/work/neko-popl23src.pdf>
- [39] Python Software Foundation. 2024. fractions — Rational numbers. <https://docs.python.org/3/library/fractions.html>
- [40] Qiskit contributors. 2023. Qiskit: An Open-source Framework for Quantum Computing. <https://doi.org/10.5281/zenodo.2573505>
- [41] Rich Rines and Isaac Chuang. 2018. High Performance Quantum Modular Multipliers. <https://doi.org/10.48550/arXiv.1801.01081> arXiv:1801.01081 [quant-ph].
- [42] Kenneth Rosen. 2010. *Elementary Number Theory and Its Application, 6th Edition* (6th edition ed.). Pearson.
- [43] Peter Selinger and Benoit Valiron. 2006. A lambda calculus for quantum computation with classical control. *Mathematical Structures in Computer Science* 16, 3 (June 2006), 527–552. <https://doi.org/10.1017/S0960129506005238> Publisher: Cambridge University Press.
- [44] P.W. Shor. 1994. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*. 124–134. <https://doi.org/10.1109/SFCS.1994.365700>
- [45] Peter W. Shor. 1999. Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer. *SIAM Rev.* 41, 2 (Jan. 1999), 303–332. <https://doi.org/10.1137/S0036144598347011> Publisher: Society for Industrial and Applied Mathematics.
- [46] Peter W. Shor. 2003. Why haven't more quantum algorithms been found? *J. ACM* 50, 1 (Jan. 2003), 87–90. <https://doi.org/10.1145/602382.602408>
- [47] D.R. Simon. 1994. On the power of quantum computation. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*. 116–123. <https://doi.org/10.1109/SFCS.1994.365701>
- [48] Daniel R. Simon. 1997. On the Power of Quantum Computation. *SIAM J. Comput.* 26, 5 (Oct. 1997), 1474–1483. <https://doi.org/10.1137/S0097539796298637> Publisher: Society for Industrial and Applied Mathematics.
- [49] Kartik Singhal, Kesha Hietala, Sarah Marshall, and Robert Rand. 2022. Q# as a Quantum Algorithmic Language. arXiv:2206.03532 [cs.PL]
- [50] Krysta Svore, Alan Geller, Matthias Troyer, John Azariah, Christopher Granade, Bettina Heim, Vadym Kliuchnikov, Mariia Mykhailova, Andres Paz, and Martin Roetteler. 2018. Q#: Enabling Scalable Quantum Computing and Development with a High-level DSL. In *Proceedings of the Real World Domain Specific Languages Workshop 2018 (RWDSL2018)*. Association for Computing Machinery, New York, NY, USA, 1–10. <https://doi.org/10.1145/3183895.3183901>
- [51] Dmitry Vasilevsky and Cesar Cortes. 2023. Sample: Grover's search algorithm. <https://github.com/microsoft/qsharp/blob/d70b544eb78e43c4ec6a88fe8825192930ff47a0/samples/algorithms/Grover.qs>
- [52] Finn Voichick, Liyi Li, Robert Rand, and Michael Hicks. 2023. Qunity: A Unified Language for Quantum and Classical Computing (Extended Version). *Proceedings of the ACM on Programming Languages* 7, POPL (Jan. 2023), 921–951. <https://doi.org/10.1145/3571225> arXiv:2204.12384 [quant-ph].
- [53] Philip Wadler. 1991. Is there a use for linear logic? *ACM SIGPLAN Notices* 26, 9 (May 1991), 255–273. <https://doi.org/10.1145/115866.115894>
- [54] Jacob Watkins. 2023. Continued fractions with Shor's algorithm: which convergent? Quantum Computing Stack Exchange. <https://quantumcomputing.stackexchange.com/a/32182/>
- [55] Theodore J. Yoder, Guang Hao Low, and Isaac L. Chuang. 2014. Fixed-Point Quantum Search with an Optimal Number of Queries. *Physical Review Letters* 113, 21 (Nov. 2014), 210501. <https://doi.org/10.1103/PhysRevLett.113.210501> Publisher: American Physical Society.
- [56] Jeffrey S. Young, Jason Riedy, Thomas M. Conte, Vivek Sarkar, Prasanth Chatarasi, and Sriseshan Srikanth. 2019. Experimental Insights from the Rogues Gallery. In *2019 IEEE International Conference on Rebooting Computing (ICRC)*. 1–8. <https://doi.org/10.1109/ICRC.2019.8914707>

- [57] Charles Yuan and Michael Carbin. 2022. Tower: data structures in Quantum superposition. *Proceedings of the ACM on Programming Languages* 6, OOPSLA2 (Oct. 2022), 134:259–134:288. <https://doi.org/10.1145/3563297>
- [58] Charles Yuan, Christopher McNally, and Michael Carbin. 2022. Twist: sound reasoning for purity and entanglement in Quantum programs. *Proceedings of the ACM on Programming Languages* 6, POPL (Jan. 2022), 30:1–30:32. <https://doi.org/10.1145/3498691>

A MINI-QWERTY LANGUAGE

In this appendix, we present the soundness of the semantics and type system of Qwerty. In order to do so, we define *Mini-Qwerty*, a formalized subset of Qwerty. We define the core features of the Mini-Qwerty language, including syntax (Section A.1), typing (Section A.2), semantics (Section A.3), and type safety (Section A.4). The language formalization below draws from Selinger and Valiron’s quantum λ -calculus [43], the μ Q language due to Yuan et al. [58], and the formalization of Q# by Singhal et al. [49].

(Types)	$\tau ::= \tau_1 \rightarrow \tau_2 \mid \tau_1 \xrightarrow{\text{rev}} \tau_2 \mid \tau_1 \otimes \tau_2 \mid () \mid \text{qubit} \mid \text{bit} \mid \text{basis}$
(Terms)	$t ::= e \mid b$
(Values)	$v ::= q_i \mid x \mid 0 \mid 1 \mid bf \mid v + v$
(Expressions)	$e ::= e_1 e_2 \mid bf \mid x \mid \ell \mid e + e \mid e[n] \mid () \mid \mathbf{phase}(\theta) * e$
(Built-In Functions)	$bf ::= b.\mathbf{measure} \mid b_1 \gg b_2 \mid b \& e \mid \sim e \mid \mathbf{id} \mid \mathbf{discard}$
(Literals)	$\ell ::= 0 \mid 1 \mid q_i \mid q\ell$
(Qubit Literal)	$q\ell ::= q_1[n]$
(Basis)	$b ::= b_1 + b_2 \mid b[n] \mid \mathbf{std} \mid \mathbf{pm} \mid \mathbf{ij} \mid \mathbf{fourier}[n] \mid \{bv_1, bv_2, \dots, bv_m\}$
(Basis Vector)	$bv ::= q\ell \mid \mathbf{phase}(\theta) * bv$

Fig. 21. Mini-Qwerty syntax. Henceforth, $n \geq 0$, $m > 0$, and $\theta \in \mathbb{R}$.

A.1 Mini-Qwerty Syntax

Fig. 21 defines Mini-Qwerty types and syntax. (Section A.5 discusses how this syntax is realized in the Python DSL.) Similar to prior work [35, 58], Mini-Qwerty has both linear types and nonlinear types. In particular, functions and classical types (namely the bit type) are nonlinear, but any type holding a qubit (i.e., a qubit or a tuple including qubits) is linear.

In Mini-Qwerty, the tensor product is associative for both terms and types; for example, the type $\text{qubit} \otimes (\text{qubit} \otimes \text{qubit})$ is identical to the type $(\text{qubit} \otimes \text{qubit}) \otimes \text{qubit}$, and the term $t_1 + (t_2 + t_3)$ is exactly the term $(t_1 + t_2) + t_3$. We show later that this facilitates taking the tensor product of functions, a major Qwerty idiom. We denote $()$ as an empty tensor product for both expressions and types. For a type τ , we write $\tau[n]$ as shorthand for an n -fold tensor product of τ , i.e., $\bigotimes_{i=1}^n \tau$.

For simplicity, we assume a Mini-Qwerty program is an expression, although Mini-Qwerty can be easily extended to represent a program with multiple functions and embeddings of classical functions (Section 3.2.5). The expressions are built on typical expressions like applications and variables alongside specific quantum expressions like bases and built-in functions. Our built-in functions are either categorized as reversible or not, where reversible generally means no qubits are measured or discarded⁵. The reversible distinction for functions (called “adjointable” by Singhal et al. [49]) is important for the Mini-Qwerty $\&$ and \sim operators (the predicator and reverser, respectively), which can only act on reversible functions; we elaborate in Section A.3.

⁵For simplicity, Mini-Qwerty also requires that reversible functions return all input qubits in exactly the same order they were passed.

$$\begin{array}{c}
\frac{}{x : \tau \vdash_{\emptyset} x : \tau} \text{TVAR} \quad \frac{}{\cdot \vdash_{\emptyset} 0 : \text{bit}} \text{T0} \quad \frac{}{\cdot \vdash_{\emptyset} 1 : \text{bit}} \text{T1} \quad \frac{}{\cdot \vdash_{\emptyset} \text{q}[m] : \text{qubit}[m|\text{q}]}} \text{TQLIT} \\
\frac{}{\cdot \vdash_{\{i\}} q_i : \text{qubit}} \text{TQ} \quad \frac{}{\Gamma \vdash_{\emptyset} () : ()} \text{TUNIT} \quad \frac{}{\cdot \vdash_{\emptyset} \mathbf{std} : \text{basis}} \text{TSTD} \quad \frac{}{\cdot \vdash_{\emptyset} \mathbf{pm} : \text{basis}} \text{TPM} \quad \frac{}{\cdot \vdash_{\emptyset} \mathbf{ij} : \text{basis}} \text{TIJ} \\
\frac{}{\cdot \vdash_{\emptyset} \mathbf{fourier}[n] : \text{basis}[n]} \text{TFOURIER} \quad \frac{}{\cdot \vdash_{\emptyset} \mathbf{id} : \text{qubit} \xrightarrow{\text{rev}} \text{qubit}} \text{TID} \quad \frac{}{\cdot \vdash_{\emptyset} \mathbf{discard} : \text{qubit} \rightarrow ()} \text{TDISCARD} \\
\frac{\Gamma_1 \vdash_{\Delta_1} bv_1 : \text{qubit}[m_1] \quad \Gamma_2 \vdash_{\Delta_2} bv_2 : \text{qubit}[m_1] \quad \cdots \quad \Gamma_m \vdash_{\Delta_{m_2}} bv_{m_2} : \text{qubit}[m_1] \quad \forall_{i=1}^{m_2} |bv_i\rangle = m_1 \quad \exists_{\sigma \in \{\sigma_x, \sigma_y, \sigma_z\}} \forall_{i=1}^{m_2} \sigma^{\otimes n} |bv_i\rangle = \lambda_i |bv_i\rangle \quad \forall_{i \neq j=1}^{m_2} \forall \theta \in \mathbb{R}, e^{i\theta} |bv_i\rangle \neq |bv_j\rangle}{\Gamma_1, \Gamma_2, \dots, \Gamma_{m_2} \vdash_{\emptyset} \{bv_1, \dots, bv_{m_2}\} : \text{basis}[m_1]} \text{TBASIS}
\end{array}$$

Fig. 22. Mini-Qwerty type rules for values and bases

$$\begin{array}{c}
\frac{\Gamma_1 \vdash_{\Delta_1} t_1 : \tau_1 \quad \Gamma_2 \vdash_{\Delta_2} t_2 : \tau_2}{\Gamma_1, \Gamma_2 \vdash_{\Delta_1 \sqcup \Delta_2} t_1 + t_2 : \tau_1 \otimes \tau_2} \text{TTENSOR} \quad \frac{\Gamma \vdash_{\emptyset} t : \tau \quad \tau \neq \tau_1 \otimes \text{qubit} \otimes \tau_2}{\Gamma \vdash_{\emptyset} t[n_2] : \tau[n_2]} \text{TNFOLD} \\
\frac{\Gamma \vdash_{\Delta} e : \text{qubit}[m]}{\Gamma \vdash_{\Delta} \mathbf{phase}(\theta) * e : \text{qubit}[m]} \text{TPHASE} \quad \frac{\Gamma_1 \vdash_{\Delta_1} e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma_2 \vdash_{\Delta_2} e_2 : \tau_1}{\Gamma_1, \Gamma_2 \vdash_{\Delta_1 \sqcup \Delta_2} e_1 e_2 : \tau_2} \text{TAPP} \\
\frac{\Gamma \vdash_{\emptyset} b : \text{basis}[m] \quad \text{span}(b) = \mathcal{H}_2^{\otimes m}}{\Gamma \vdash_{\emptyset} \mathbf{b.measure} : \text{qubit}[m] \rightarrow \text{bit}[m]} \text{TMEASURE} \quad \frac{\Gamma_1 \vdash_{\emptyset} b_1 : \text{basis}[m] \quad \Gamma_2 \vdash_{\emptyset} b_2 : \text{basis}[m] \quad \text{span}(b_1) = \text{span}(b_2)}{\Gamma_1, \Gamma_2 \vdash_{\emptyset} b_1 \gg b_2 : \text{qubit}[m] \xrightarrow{\text{rev}} \text{qubit}[m]} \text{TBTRANS} \\
\frac{\Gamma_1 \vdash_{\emptyset} b : \text{basis}[m_1] \quad \Gamma_2 \vdash_{\Delta} e : \text{qubit}[m_2] \xrightarrow{\text{rev}} \text{qubit}[m_2]}{\Gamma_1, \Gamma_2 \vdash_{\Delta} b \& e : \text{qubit}[m_1 + m_2] \xrightarrow{\text{rev}} \text{qubit}[m_1 + m_2]} \text{TPRED} \quad \frac{\Gamma \vdash_{\Delta} e : \text{qubit}[m] \xrightarrow{\text{rev}} \text{qubit}[m]}{\Gamma \vdash_{\Delta} \sim e : \text{qubit}[m] \xrightarrow{\text{rev}} \text{qubit}[m]} \text{TREV}
\end{array}$$

Fig. 23. Mini-Qwerty type rules for operations

$$\begin{array}{c}
\frac{\tau_1 <: \tau_2 \quad \Gamma \vdash_{\Delta} e : \tau_1}{\Gamma \vdash_{\Delta} e : \tau_2} \text{TSUB} \quad \frac{}{\tau_1 \xrightarrow{\text{rev}} \tau_2 <: \tau_1 \rightarrow \tau_2} \text{SREV} \\
\frac{\left(\left(\bigotimes_{i=1}^{n_1} \tau_i \right) \rightarrow \left(\bigotimes_{j=1}^{n_2} \tau_j \right) \right) \otimes \left(\left(\bigotimes_{k=1}^{n_3} \tau_k \right) \rightarrow \left(\bigotimes_{\ell=1}^{n_4} \tau_{\ell} \right) \right)}{<: \left(\left(\bigotimes_{i=1}^{n_1} \tau_i \right) \otimes \left(\bigotimes_{k=1}^{n_3} \tau_k \right) \right) \rightarrow \left(\left(\bigotimes_{j=1}^{n_2} \tau_j \right) \otimes \left(\bigotimes_{\ell=1}^{n_4} \tau_{\ell} \right) \right)} \text{STENSFUNC}
\end{array}$$

Fig. 24. Mini-Qwerty subtyping rules

Bit literals are 0 and 1 as usual. \mathbf{id} is the single-qubit identity function and is useful for padding functions out to apply to more qubits. The syntax \gg performs the crucial *basis translation* introduced in Section 3.2.2. (We describe bases in more detail in the next section.) The operator $\mathbf{phase}(\theta) *$ imparts a phase $e^{i\theta}$ on qubits or basis vectors.

Although qubit literals are written as string literals in the Python DSL per Section 3.2.1, we avoid overspecializing for Python and permit a more abstract form q of qubit literals in Mini-Qwerty. We assume each q is a nonempty sequence of the symbols $0, 1, +, -, i,$ and j . The notation $|\text{q}\rangle$ denotes the length of this sequence. $|\text{q}\rangle$ denotes the quantum state that q represents — for example, if $\text{q} = +10-$, then $|\text{q}\rangle = |+\rangle \otimes |10\rangle \otimes |-\rangle$. The $[n]$ suffix following q prepares n duplicate versions of q .

Finally, q_i , which represents a reference to the qubit at index i , is not written by programmers; it is used only during evaluation [58].

$$\begin{aligned}
(|\psi_i\rangle)_{i=1}^{n_1} \otimes (|\phi_j\rangle)_{j=1}^{n_2} &\triangleq (|\psi_i\rangle \otimes |\phi_j\rangle)_{i=1,2,\dots,n_1; j=1,2,\dots,n_2} \\
\text{veclist}(b_1 + b_2) &\triangleq \text{veclist}(b_1) \otimes \text{veclist}(b_2) \\
\text{veclist}(b[n]) &\triangleq \text{veclist}(\bigoplus_{i=1}^n b) \\
\text{veclist}(\mathbf{std}) &\triangleq |0\rangle, |1\rangle \\
\text{veclist}(\mathbf{pm}) &\triangleq |+\rangle, |-\rangle \\
\text{veclist}(\mathbf{ij}) &\triangleq |+i\rangle, |-i\rangle \\
\text{veclist}(\mathbf{fourier}[n]) &\triangleq |F_0\rangle, |F_1\rangle, \dots, |F_{2^n-1}\rangle \\
&\quad \text{with } |F_j\rangle \triangleq \frac{1}{\sqrt{2^n}} \sum_{k=0}^{2^n-1} e^{i2\pi jk/2^n} |k\rangle \\
\text{vec}(\mathbf{phase}(\theta) * bv) &\triangleq e^{i\theta} \text{vec}(bv) \\
\text{vec}(q[n]) &\triangleq \bigotimes_{i=1}^n |q\rangle \\
\text{veclist}(\{bv_1, bv_2, \dots, bv_m\}) &\triangleq \text{vec}(bv_1), \text{vec}(bv_2), \dots, \text{vec}(bv_m) \\
|b^1\rangle, |b^2\rangle, \dots, |b^m\rangle &\triangleq \text{veclist}(b) \\
\text{span}(b) &\triangleq \text{span}(|b^1\rangle, |b^2\rangle, \dots, |b^m\rangle) \\
|b| &\triangleq m \text{ for which } |b^1\rangle \in \mathcal{H}_2^{\otimes m}
\end{aligned}$$

Fig. 25. Definition of properties of a basis b , including the number of qubits across which it is defined (written $|b|$) and the list of vectors it represents. This list is an orthonormal basis if b is well-typed. The notation $(\cdot)_{i=1}^n$ denotes a comma-separated list of length n .

A.2 Mini-Qwerty Type System

Fig. 22 and Fig. 23 show the typing rules for Mini-Qwerty. In our typing rules, a term (t) with type (τ) based on context (Γ) and qubit index context (Δ) is well-typed in the given judgement: $\Gamma \vdash_{\Delta} t : \tau$. The qubit index context Δ is a set of positive integers representing the set of all qubit indices i in all q_i values in the expression. We write $A \sqcup B$ to denote the union of disjoint sets A and B , and $[n]$ is defined as the set $\{1, 2, \dots, n\}$. When the type is linear (qubits, or tensor product holding qubits), we only allow exchange rules on the context Γ to maintain linearity [35, 43, 53, 58]. In the other cases (for non-quantum types), we also allow weakening and contraction.

Fig. 24 defines the subtyping rules for Mini-Qwerty. SREV exists for programmer convenience, since a reversible function should be usable anywhere an ordinary irreversible function could be. The STENSFUNC rule facilitates a unique feature of Qwerty: the ability to use the tensor product of functions (i.e., tuples of functions) in an application. For example, applying $(\mathbf{id} + \mathbf{discard} + \mathbf{id})$ to three qubits would preserve the leftmost and rightmost qubits, but discard the middle qubit.

The rule TBASIS mandates that a basis literal $\{bv_1, bv_2, \dots, bv_m\}$ meets all the following conditions, which guarantee that it satisfies the linear algebraic definition of an orthonormal basis [3]:

- (1) All bv_i must have the same number of qubits (denoted $|bv_i|$).
- (2) No bv_i s are expressed as a mixture of symbols from the pairs $0/1$, $+/-$, and i/j . In mathematical terms, either all $|bv_i\rangle$ are eigenstates of $\sigma_x^{\otimes |bv_i|}$, or all are eigenstates of $\sigma_y^{\otimes |bv_i|}$ instead, or all are eigenstates of $\sigma_z^{\otimes |bv_i|}$ instead. This helps ensure all q_i s are linearly independent; for example, the list of vectors $|0\rangle, |1\rangle, |+\rangle$ is linearly dependent.
- (3) bv_i s cannot be repeated, even with a different phase. Mathematically, for any $1 \leq i, j \leq m$ such that $i \neq j$, there exists no $\theta \in \mathbb{R}$ such that $e^{i\theta} |bv_i\rangle = |bv_j\rangle$. This also helps guarantee linear independence.

$$\begin{array}{c}
\frac{}{[\psi], \mathbf{id} \ q_i \longrightarrow [\psi], q_i} \text{EID} \quad \frac{}{[\psi], \mathbf{discard} \ q_i \longrightarrow [\psi], ()} \text{EDISCARD} \\
\frac{}{[\psi], \mathbb{Q}[m] \longrightarrow [\psi] \otimes \bigotimes_{i=1}^m |\mathbb{Q}\rangle, \mathbf{+}_{j=1}^{m|\mathbb{Q}|} \ q_{ij}} \text{EQLIT} \\
\frac{|b_1| = |b_2| = m}{[\psi], (b_1 \gg b_2)(\mathbf{+}_{j=1}^m \ q_{ij}) \longrightarrow [U_{b_1 \rightarrow b_2}^{\vec{i}} \psi], \mathbf{+}_{j=1}^m \ q_{ij}} \text{EBTRANS} \\
\frac{|b| = m}{[\psi], b.\mathbf{measure}(\mathbf{+}_{j=1}^m \ q_{ij}) \xrightarrow{p_{b,j}^{\vec{i}}(\psi)} \left[\frac{M_{b,j}^{\vec{i}}}{\sqrt{p_{b,j}^{\vec{i}}(\psi)}} |\psi\rangle, \mathbb{B}_{m,j}^{m-1} + \mathbb{B}_{m,j}^{m-2} + \dots + \mathbb{B}_{m,j}^0 \right]} \text{EMEASURE} \\
\frac{}{[\psi], e[n] \longrightarrow [\psi], \mathbf{+}_{j=1}^n \ e} \text{ENFOLD} \quad \frac{[\psi], e \longrightarrow [\psi'], e'}{[\psi], \mathbf{phase}(\theta) * e \longrightarrow [\psi'], \mathbf{phase}(\theta) * e'} \text{EPHASE1} \\
\frac{}{[\psi], \mathbf{phase}(\theta) * (\mathbf{+}_{j=1}^m \ q_{ij}) \longrightarrow [e^{i\theta} \psi], \mathbf{+}_{j=1}^m \ q_{ij}} \text{EPHASE2} \\
\frac{[\psi'], e(\mathbf{+}_{j=1}^m \ q_{ij}) \longrightarrow [U^{\vec{i}} \psi'], \mathbf{+}_{j=1}^m \ q_{ij} \quad |\psi\rangle, |\psi'\rangle \in \mathcal{H}_2^{\otimes n}}{[\psi], (\sim e)(\mathbf{+}_{j=1}^m \ q_{ij}) \longrightarrow [(U^{\vec{i}})^\dagger \psi], \mathbf{+}_{j=1}^m \ q_{ij}} \text{EREV} \\
\frac{|b| = m_2 \quad [\psi'], e(\mathbf{+}_{j=1}^{m_1} \ q_{ij}) \longrightarrow [U^{\vec{i}} \psi'], \mathbf{+}_{j=1}^{m_1} \ q_{ij} \quad |\psi\rangle, |\psi'\rangle \in \mathcal{H}_2^{\otimes n}}{[\psi], (b \ \& \ e)((\mathbf{+}_{k=1}^{m_2} \ q_{i_k}^p) + (\mathbf{+}_{j=1}^{m_1} \ q_{ij})) \longrightarrow [C_b U^{\vec{i}p, \vec{i}} \psi], (\mathbf{+}_{k=1}^{m_2} \ q_{i_k}^p) + (\mathbf{+}_{j=1}^{m_1} \ q_{ij})} \text{EPRED} \\
\frac{[\psi], e_1 \longrightarrow [\psi'], e'_1 \quad n_1 + n_2 > 0}{[\psi], (\mathbf{+}_{j=1}^{n_1} \ v_j) + e_1 + (\mathbf{+}_{k=2}^{n_2} \ e_k) \longrightarrow [\psi'], (\mathbf{+}_{j=1}^{n_1} \ v_j) + e'_1 + (\mathbf{+}_{k=2}^{n_2} \ e_k)} \text{ETENSOR} \\
\frac{[\psi], v_1(\mathbf{+}_{j=1}^{n_1} \ v'_j) \longrightarrow [\psi'], e \quad n_1 \leq n_2}{[\psi], (\mathbf{+}_{k=1}^{n_1} \ v_k)(\mathbf{+}_{j=1}^{n_2} \ v'_j) \longrightarrow [\psi'], e + ((\mathbf{+}_{k=2}^m \ v_k)(\mathbf{+}_{j=n_1+1}^{n_2} \ v'_j))} \text{ETENSAPP} \\
\frac{[\psi], e_1 \xrightarrow{p} [\psi'], e'_1}{[\psi], e_1 \ e_2 \xrightarrow{p} [\psi'], e'_1 \ e_2} \text{EAPP1} \quad \frac{[\psi], e \xrightarrow{p} [\psi'], e'}{[\psi], v \ e \xrightarrow{p} [\psi'], v \ e'} \text{EAPP2}
\end{array}$$

Fig. 26. Mini-Qwerty evaluation rules

The **TMEASURE** rule requires that the operand b of $b.\mathbf{measure}$ is a basis for the full n -qubit Hilbert space ($\mathcal{H}_2^{\otimes n}$). The judgment of b as well-typed ensures that b is an orthonormal list; if this orthonormal list spans $\mathcal{H}_2^{\otimes n}$, then b represents an n -qubit basis, meaning the resulting measurement operators (Section A.3) satisfy the completeness equation [30, §2.2.3]. A basis translation $b_1 \gg b_2$, on the other hand, mandates that b_1 and b_2 span the same spaces (TBTRANS), preserving unitarity. (Fig. 25 formally defines the span of a basis b .)

The predictor operator $b \ \& \ e$ returns a new function which runs the function e only in the subspace b . Here, e must be a reversible function – specifically, this means it must have the effect of a unitary on the state, meaning it cannot perform measurements or discard qubits [49].

A.3 Mini-Qwerty Semantics

Fig. 26 shows the evaluation rules for Mini-Qwerty. Because classical control hardware executes a Mini-Qwerty program, causing side effects on a quantum state, we represent the state of a Mini-Qwerty program as a pair $[\psi], e$ of a quantum state $|\psi\rangle$ and a Mini-Qwerty expression e [43, 57].

The initial quantum state may be chosen as $|\rangle = [1]$, a 1×1 matrix. (For notational convenience, we denote $\text{span}(|\rangle)$ as $\mathcal{H}_2^{\otimes 0}$.)

The evaluation relation $[|\psi\rangle, e] \xrightarrow{p} [|\psi'\rangle, e']$ includes a probability p to account for quantum nondeterminism [43, 57]. Often p is omitted, as in $[|\psi\rangle, e] \longrightarrow [|\psi'\rangle, e']$, which should be read as letting $p = 1$. Currently, due to the possibility of different measurement results, only EMEASURE introduces a probability p which may not be equal to 1.

The rule EDISCARD permits the programmer to explicitly discard a qubit reference q_i . The hardware or runtime may reset q_i to $|0\rangle$ for later use, but Mini-Qwerty does not implement this for simplicity. Conversely, rule EQLIT prepares n copies of the requested state $|\psi\rangle$; a runtime may repurpose previously discarded qubits, but for notational simplicity we simply expand $|\psi\rangle$ to include the new qubits. If $|\psi\rangle \in \mathcal{H}_2^{\otimes n}$, then the indices i_j in EQLIT are defined as $i_j \triangleq n + j$ to ensure the new q_{i_j} point to the fresh qubits.

EBTRANS describes the behavior of basis translations, introduced in Section 3.2.2. (In all rules, the notation U^{i_1, i_2, \dots, i_n} , abbreviated as $U^{\vec{i}}$, means U should be applied to the qubits at indices i_1, i_2, \dots, i_n .) In essence, $U_{b_1 \rightarrow b_2}$ is $\text{diag}(1, 1, \dots, 1)$ where the columns are written in the basis b_1 , and the rows are written in the basis b_2 . To define this more rigorously, first suppose $\text{span}(b_1) = \text{span}(b_2)$ (defined in Fig. 25) and the vectors of the bases they represent are $|b_i^k\rangle$, $i \in \{1, 2\}$ (defined formally in Fig 25). If $|b_1^k\rangle$ is extended to an orthonormal basis $|e_1^k\rangle$ of $\mathcal{H}_2^{\otimes n}$ with Gram–Schmidt [3, 30], then let $|e_2^k\rangle$ be $|b_2^k\rangle$ extended with the same vectors by which $|b_1^k\rangle$ was extended. Then we define $U_{b_1 \rightarrow b_2} = \sum_k |e_2^k\rangle \langle e_1^k|$.

Measurement, defined by EMEASURE, also takes a basis operand. If $|b| = m$, then measurement operators $M_{b,i} \triangleq |b^i\rangle \langle b^i|$ are projectors onto the 2^m basis states of b . The notation $p_{b,j}^{\vec{i}}(|\psi\rangle) = \langle \psi | M_{b,j}^{\vec{i}} | \psi \rangle$ describes the probability of observing outcome $1 \leq j \leq 2^m$ on the qubits at indices \vec{i} of $|\psi\rangle$. We represent the measurement outcome itself as m -bit binary: $\mathbb{B}_{m,j}$ is $j-1$ expressed in m -bit binary, and $\mathbb{B}_{m,j}^k$ is bit k , where $\mathbb{B}_{m,j}^0$ is the least significant.

EPRED defines the behavior of the predicator ($\&$), which predicates a function running on a set of qubits (q_{i_j}) on a subspace of predicate qubits (q_{i_p}). Here we describe the $C_b U$ notation used in the rule. Using Gram–Schmidt, extend $|b^k\rangle$, the basis represented by b , to an orthonormal basis of $\mathcal{H}_2^{\otimes n_2}$, denoting vectors by which $|b^k\rangle$ would be extended as $|e^\ell\rangle$ [3, 30]. Then if U is an n -qubit unitary, let $C_b U = (\sum_\ell |e^\ell\rangle \langle e^\ell|) \otimes I_{2^n} + (\sum_k |b^k\rangle \langle b^k|) \otimes U$.

Besides the aforementioned basis-oriented quantum primitives, ETENSAPP defines how exactly tensor products of functions are applied in Mini-Qwerty. The functions in the tensor product are evaluated left-to-right. The number of values to peel off and pass to the leftmost function (n_1 in ETENSAPP) is determined based on the prototype of the function, which must specify the number of inputs; for example, extending Fig. 21 to include lambdas would require an annotation to specify the number of input values, like $\lambda(x_1: \tau_1, x_2: \tau_1, \dots, x_n: \tau_n).e$.

A.4 Mini-Qwerty Properties

As with any new language, we need to prove progress and preservation. These properties prove the type safety of Mini-Qwerty. Progress states a type-safe expression means it is either a value or can do one step of evaluation. Preservation states a type-safe expression with an evaluation step to a new expression should be type-safe. We state the mathematical formulas as Theorem A.1 and Theorem A.2 respectively.

THEOREM A.1. (Progress) If $\cdot \vdash_\Delta e : \tau$, either e is a value or $\forall |\psi\rangle \in \mathcal{H}_2^{\otimes n}$ where $\Delta \subseteq [n]$, $[|\psi\rangle, e] \xrightarrow{p} [|\psi'\rangle, e']$.

THEOREM A.2. (Preservation) *If $\Gamma \vdash_n e : \tau$ and $[|\psi\rangle, e] \xrightarrow{P} [|\psi'\rangle, e']$ and $|\psi\rangle \in \mathcal{H}_2^{\otimes n}$ with $\Delta \subseteq [n]$, then $\Gamma \vdash_{\Delta'} e' : \tau$ and $|\psi'\rangle \in \mathcal{H}_2^{\otimes n'}$ where $\Delta' \subseteq [n']$.*

Progress can be proved by induction on derivation of $\cdot \vdash_{\Delta} e : \tau$. Preservation is an induction on derivation of $[|\psi\rangle, e] \xrightarrow{P} [|\psi'\rangle, e']$ [58].

THEOREM A.3. (Universality) *Mini-Qwerty is universal.*

PROOF. The following basis translation performs a unitary transformation exactly equal to an $R_z(\theta)$:

$$\{0[1], 1[1]\} \gg \{\mathbf{phase}(-\theta/2) * 0[1], \mathbf{phase}(\theta/2) * 1[1]\}$$

Similarly, the following acts exactly as an $R_y(\theta)$:

$$\{i[1], j[1]\} \gg \{\mathbf{phase}(-\theta/2) * i[1], \mathbf{phase}(\theta/2) * j[1]\}$$

Additionally, the following applies a global phase of θ :

$$\{0[1], 1[1]\} \gg \{\mathbf{phase}(\theta) * 0[1], \mathbf{phase}(\theta) * 1[1]\}$$

Then using a ZYZ decomposition [30, §4.2], Mini-Qwerty is capable of executing any one-qubit unitary by applying the aforementioned 3 functions with different choices of θ .

Furthermore, a CNOT can be performed in Mini-Qwerty with the following basis translation:

$$\{1[1]\} + \{0[1], 1[1]\} \gg \{1[1]\} + \{1[1], 0[1]\}$$

Thus, by the universality of single qubit gates and CNOTs [30, §4.5.2], Mini-Qwerty is universal. \square

A.5 Realization in the Python DSL

Syntax differences. Qwerty is largely an embedding of Mini-Qwerty in Python, but there are some departures from the syntax in Fig. 21. The most notable is that application in Qwerty is written as $e_2 \mid e_1$ rather than $e_1 e_2$ as in Mini-Qwerty (Fig. 21) — that is, the order is reversed from before, with the input first and the function second. This change makes Qwerty code read from left-to-right like a Unix shell pipeline, where earlier operations are written before later operations. This pipeline-like style of programming avoids the tedious repeated variable definitions common in languages with linear qubit types [35, 58]. Fig. 27 shows an example of this.

Otherwise, Mini-Qwerty syntax is materialized in Qwerty using common Python syntax: for example, the sequence `q` is a Python string literal containing only `'0'`, `'1'`, `'+'`, `'-'`, `'i'`, and `'j'`. (As discussed in Section 3.2.1, combined with qubit literals as string literals, the choice of `+` lines up with intuition of the tensor product as concatenation.) In Qwerty, the types in Figure 21 are written similarly to typical Python type annotations: $\text{qubit}[n_1] \rightarrow \text{qubit}[n_2]$ is written as `qfunc[n1, n2]`, $\text{qubit}[n_1] \xrightarrow{\text{rev}} \text{qubit}[n_2]$ as `rev_qfunc[n1, n2]`, and $\text{bit}[n_1] \rightarrow \text{bit}[n_2]$ as `cfunc[n1, n2]`.

Syntactic sugar. There are also several cases of syntactic sugar in Qwerty versus Mini-Qwerty: in Qwerty, the unary operator `-` is equivalent to `phase(π)*`, and programmers can write `q` instead of `q[n]` to imply $n = 1$. Furthermore, a qubit literal `q[n]` can be used wherever a basis can be used and will automatically be promoted to a singleton basis literal `{q[n]}`. Qwerty also supports writing a predicator as `e & b` in addition to the `b & e` syntax supported by Mini-Qwerty.

```

(* ... *)
let (p0 : qubit<M>,
    p1 : qubit<M>) = p in
let (p0 : qubit<M>,
    p1 : qubit<M>) = (H p0, H p1) in
let (p0 : qubit<M>,
    p1 : qubit<M>) = (Z p0, Z p1) in
let (p0 : qubit<M>,
    p1 : qubit<M>) = CZ (p0, p1) in
let (p0 : qubit<M>,
    p1 : qubit<M>) = (H p0, H p1) in
(* ... *)

```

(a) Twist: The diffuser in Grover's algorithm [58]

```

# ...
q | pm[N] >> std[N]
  | -( '0'[N] >> -'0'[N])
  | std[N] >> pm[N]
# ...

```

(b) Qwerty: The diffuser in Grover's algorithm written more verbosely than in Fig. 3d

Fig. 27. Side-by-side comparison of the Grover diffuser in Twist versus Qwerty to demonstrate the succinctness afforded by the pipe | operator.

Classical functions. Mini-Qwerty focuses on code decorated with `@qpu` (Section 3.2.4), but as discussed in Section 3.2.5, Qwerty also supports classical code decorated with `@classical`. The type system for the `@classical` DSL is straightforward, guaranteeing that e.g. binary bitwise operations are performed on bitvectors with the same dimension. In the Qwerty `@qpu` DSL, a classical function `f` can be instantiated using `f.xor_embed`, `f.phase`, or `f.inplace(f_inv)` as Section 3.2.5 describes. This means Qwerty extends Mini-Qwerty with more type rules in the spirit of the following:

$$\frac{\Gamma \vdash_0 f : \text{bit}[n_1] \rightarrow \text{bit}[n_2]}{\Gamma \vdash_0 f.\text{xor_embed}(e) : \text{qubit}[n_1 + n_2] \xrightarrow{\text{rev}} \text{qubit}[n_1 + n_2]} \text{TXOREMBED}$$

Qwerty type checking. When type-checking the body of a Qwerty `@qpu` kernel, the type context Γ is initialized with the types of captures and arguments, and the resulting type of the expression should match the result type defined by the function signature. Type checking fails when a `@qpu` kernel calls a non-reversible function yet is decorated with `@reversible` to indicate its type is $\tau_1 \xrightarrow{\text{rev}} \tau_2$ (e.g., line 4 of Fig. 18 in Section 4.3.2). Note that while naïvely implementing basis-related checks based on definitions in Fig. 25 could cause type checking to take time exponential in the number of qubits, the Qwerty compiler uses straightforward optimizations to prevent this.

Semantic differences. In Qwerty, `phase(θ)*` can be applied to functions (e.g., `phase(θ) * id`), whereas `phase(θ)*` can only be applied to qubits in Mini-Qwerty (e.g., `phase(θ) * 0[1]`). More significantly, the `EDISCARD` rule in Fig. 26 describes the intent of `discard` faithfully in abandoning its qubit operand, but in a realistic, efficient implementation, reusing discarded qubits is crucial. There are two situations where a programmer would discard a qubit: (1) at the end of an algorithm when the measurement result of a dirty qubit is unneeded, as seen in many of the examples in Section 4; and (2) returning a clean ($|0\rangle$) qubit to the pool of ancilla qubits maintained by the quantum runtime. In Qwerty, the `discard` built-in function handles case (1) and resets a qubit to $|0\rangle$ using measurement; the `discardz` built-in handles case (2) and returns the qubit to the ancilla pool without measurement.