DOI:10.1145/2699415

The Quipper language offers a unified general-purpose programming framework for quantum computation.

BY BENOÎT VALIRON, NEIL J. ROSS, PETER SELINGER, D. SCOTT ALEXANDER, AND JONATHAN M. SMITH

Programming the Quantum Future

THE EARLIEST COMPUTERS, like the ENIAC, were rare and heroically difficult to program. That difficulty stemmed from the requirement that algorithms be expressed in a "vocabulary" suited to the particular hardware available, ranging from function tables for the ENIAC to more conventional arithmetic and movement operations on later machines. Introduction of symbolic programming languages, exemplified by FORTRAN, solved a major difficulty for the next generation of computing devices by enabling specification of an algorithm in a form more suitable for human understanding, then translating this specification to a form executable by the machine. The "programming language" used for such specification bridged a semantic gap between the human and the computing device. It provided two important features: high-level abstractions, taking care of automated bookkeeping, and modularity, making it easier to reason about sub-parts of programs.

Quantum computation is a computing paradigm where data is encoded in the state of objects governed by the laws of quantum physics. Using quantum techniques, it is possible to design algorithms that outperform their best-known conventional, or classical, counterparts.

While quantum computers were envisioned in the 20th century, it is likely they will become real in the 21st century, moving from laboratories to commercial availability. This provides an opportunity to apply the many lessons learned from programming classical computing devices to emerging quantum computing capabilities.

Quantum Coprocessor Model

How would programmers interact with a device capable of performing quantum operations? Our purpose here is not to provide engineering blueprints for building an actual quantum computer; see Meter and Horsman[13] for a discussion of that agenda. What we describe is a hypothetical quantum architecture in enough detail to cover how one would go about programming it.

Viewed from the outside, quantum computers perform a set of specialized operations, somewhat analogous to a floating-point unit or a graphics coprocessor. We therefore envision the quantum computer as a kind of coprocessor that is controlled by a classical computer, as shown schematically in Figure 1. The classical computer per-

» key insights

- **Quantum computer science is a new discipline dealing with the practical integration of all aspects of quantum computing, from an abstract algorithm in a research paper all the way to physical operations.**
- **The programs written in a quantum programming language should be as close as possible to informal high-level descriptions, with output suitable for the quantum coprocessor model.**
- **Other important aspects of the quantum programming environment include automated offline resource estimates prior to deployment and facilities for testing, specification, and verification.**

ILLUSTRATION BY FUTUREDELUXE

forms operations (such as compilation, conventional bookkeeping, correctness checking, and preparation of code and data) for the quantum unit. The quantum coprocessor performs only the quantum operations (such as initializations, unitary operations, and measurements). This model of quantum computation is known as Knill's QRAM model[11] and is believed to ultimately be the most likely realization of quantum computers.[13]

Certain hardware-intensive low-level control operations (such as quantum error correction) may optionally be integrated directly into the quantum unit. We envision the quantum unit containing a high-speed, specialized firmware in charge of such a low-level "quantum runtime." The quantum firmware is specific to each physical realization of a quantum coprocessor, programmed separately off site. Although tightly dependent on the physical specifications of the particular hardware, the quantum firmware is independent of the algorithms to be run.

The source code of any quantum programs resides on the classical unit. Through a conventional classical compilation, it produces executable code to be run on the conventional computer. We envision the quantum coprocessor will communicate with its classical controller through a message queue on which the classical computer is able to send elementary instructions (such as "allocate a new quantum bit," "rotate quantum bit x," and "measure quantum bit y"). After an operation is performed, the classical computer can read the results from the message queue. In this model, the control flow of an algorithm is classical; tests and loops are performed on the classical device. Both classical and quantum data are first-class objects.

Via the message queue, the classical runtime receives feedback (such as the results of measurements) from the quantum unit. Depending on the algorithm, this feedback may occur only at the end of the quantum computation (batch-mode operation) or interleaved with the generation of elementary instructions (online operation). The possibility of online operation raises additional engineering challenges, as it requires the classical controller to be fast enough to interact with the quantum runtime in real time. On the other hand, many common quantum algorithms require only batch-mode operation. We assume a quantum programming model flexible enough to address either type of operation.

As with a conventional programming environment, we separate the logical data structures from their physical

Figure 1. Mixed computation in the quantum coprocessor model.

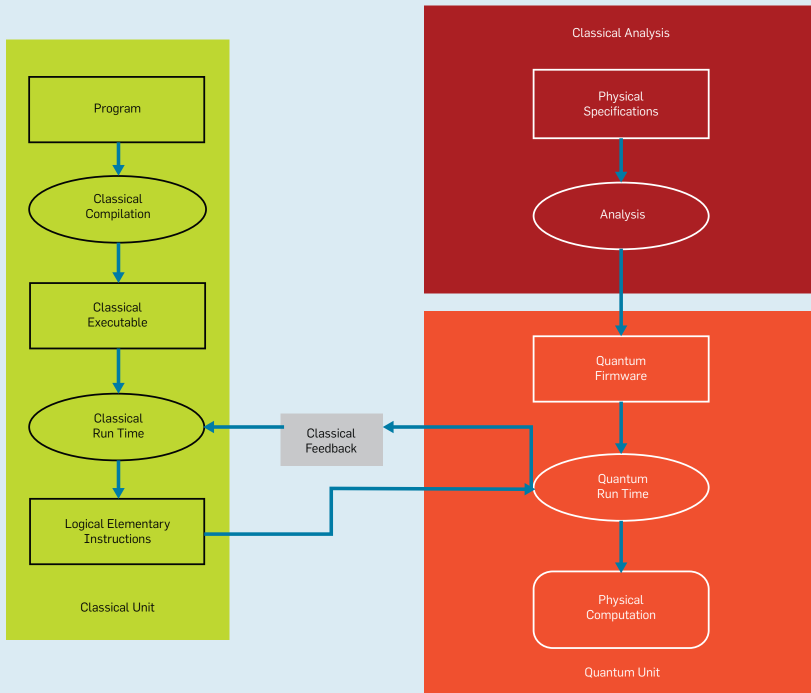


Figure 2. A quantum circuit.

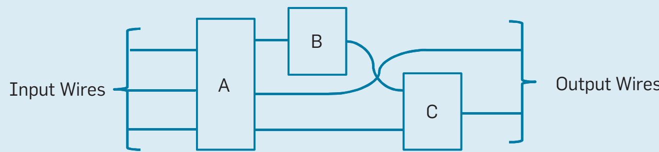
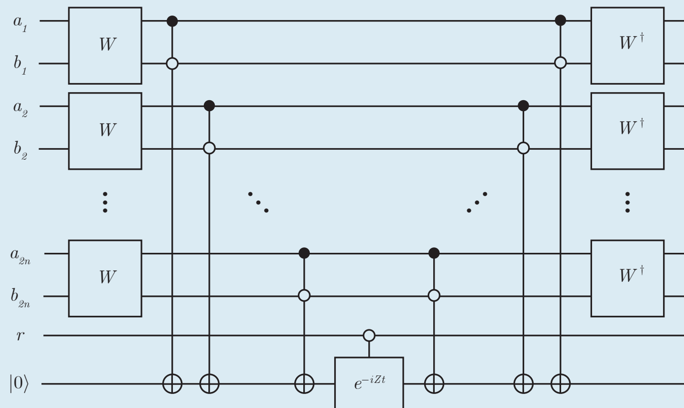


Figure 3. A quantum circuit fragment.



representation on the hardware. In our proposed paradigm, the algorithms are implemented at the logical level, but the quantum bits are physically encoded at the hardware level. The tasks of mapping logical quantum bits and operations to stable physical representations, and of applying suitable error correction, are left to the compiler and to the quantum firmware.

Describing Quantum Algorithms

To motivate the need for an expressive quantum programming language (QPL), we briefly consider some of the ways quantum algorithms are typically specified in the literature. A quantum algorithm generally consists of a mix of classical and quantum operations. The quantum parts of an algorithm are usually aggregated into quantum circuits, in which quantum gates are represented by boxes and quantum bits by wires, as in Figure 2. Some restrictions apply to circuits. For example, they cannot contain loops, so wires must flow in only one direction. The gates can have multiple inputs and outputs. With the exception of gates corresponding to the creation and destruction (measurement) of quantum bits, elementary operations are always unitary transformations, implying they must have the same number of inputs and outputs.

A typical description of a quantum algorithm consists of one or more of the following pieces, which may be specified at various levels of formality:

Mathematical equations. These can be used to describe state preparations, unitary transformations, and measurements. For example, Harrow et al.[9] described a quantum algorithm for solving linear systems of equations. A certain subcircuit of the algorithm is defined as follows:

$$\sum_{t=1}^{T}\begin{pmatrix} |t+1\rangle\langle t|\otimes U_t \\ +|t+T+1\rangle\langle t+T|\otimes I \\ +|t+2T+1 \bmod 3T\rangle\langle t+2T|\otimes U^{\dagger}_{3T+1-t} \end{pmatrix}$$

Figure 4. Inversion and repetition.

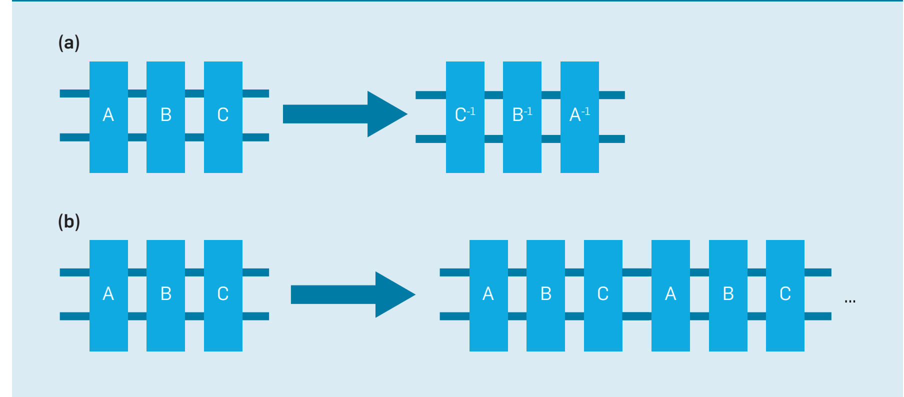


Figure 5. An initialize-run-measure loop.

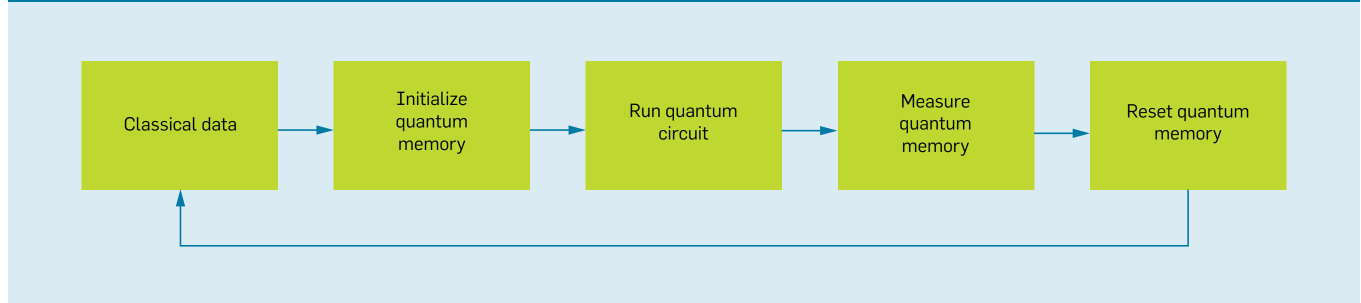


Figure 6. A circuit with feedback from intermediate measurements.

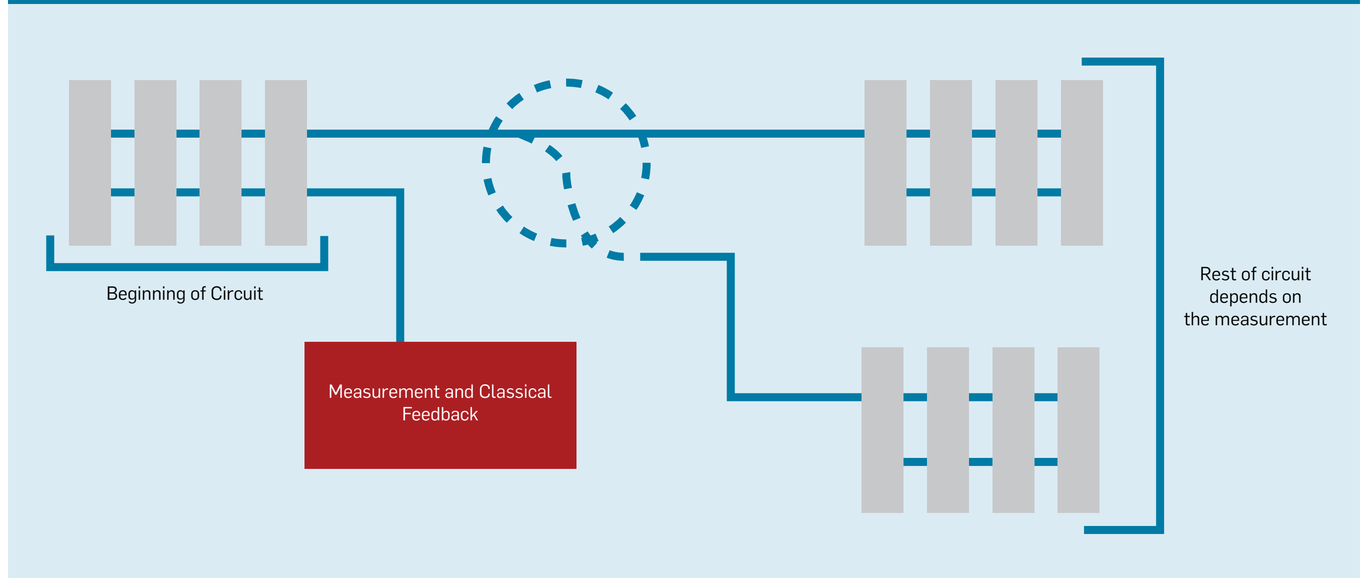
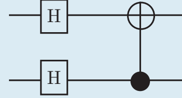


Figure 7. A procedural example.

```
mycirc :: Qubit -> Qubit -> Circ (Qubit, Qubit)
mycirc a b = do
  a <- hadamard a
  b <- hadamard b
  (a,b) <- controlled_not a b
  return (a,b)
```



Invocation of known quantum subroutines. Examples include the quantum Fourier transform, phase estimation, amplitude amplification, and random walks. For example, the algorithm in Harrow et al.[9] asks to "decompose $|b\rangle$ in the eigenvector basis, using phase estimation";

Oracles. These are classical computable functions that must be made reversible and encoded as quantum operations. They are often described at a very high level; for example, Burham et al.[3] defined an oracle as "the truth value of the statement $f(x) \leq f(y)$";

Circuit fragments. For example, the circuit in Figure 3 is from Childs et al.[4] Note, strictly speaking, the figure describes a *family* of quantum circuits, parameterized by a rotation angle t and a size parameter n, as indicated by ellipses ". . ." in the informal circuit. In a formal implementation, this parameter dependency must be made explicit;

High-level operations on circuits. Examples include "inversion," where a circuit is reversed, and "iteration," where a circuit is repeated, as in Figure 4; and

Classical control. Many algorithms involve interaction between the classical unit and the quantum unit. This interaction can take the form of simple iteration, running the same quantum circuit multiple times from scratch, as in Figure 5, or of feedback, where the quantum circuit is generated on the fly, possibly based on the outcome of previous measurements, as in Figure 6.

Requirements for QPLs

Ideally, a quantum programming language should permit programmers to implement quantum algorithms at a level of abstraction that is close to how one naturally thinks about the algorithm. If the algorithm is most naturally described by a mathematical formula, then the programming language should support such a description where possible. Similarly, if the algorithm is most naturally described by a sequence of low-level gates, the programming language should support this description as well.

The standard methods used to present many algorithms in the literature can therefore be taken as guidelines in the design of a language. Knill[11] laid out requirements for quantum programming, including:

Allocation and measurement. Make it possible to allocate and measure quantum registers and apply unitary operations;

Reasoning about subroutines. Permit reasoning about quantum subroutines, reversing a subroutine, and conditioning a subroutine over the state of a quantum register; and

Building quantum oracles. Enable the programmer to build a quantum oracle from the description of a classical function.

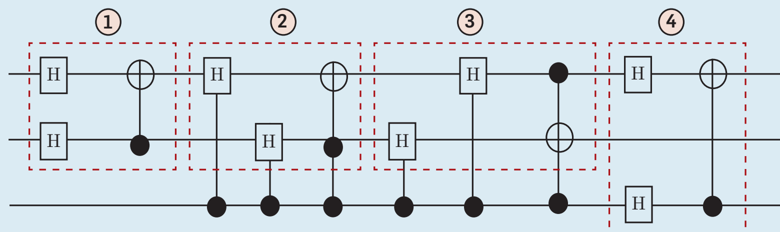
Our experience implementing[a] quantum algorithms suggests some additional features that would be requirements for a quantum programming language:

Quantum data types. In classical languages, data types are used to permit the programmer to think abstractly about data instead of managing individual bits or words. For example, in most situations, a floating-point number is best viewed as a primitive data type supporting certain arithmetic operations, rather than an array of 64 bits comprising an exponent, a mantissa, and a sign. Likewise, many quantum algorithms specify richer data types, so the language should also provide these abstractions. For example, the Quantum Linear System Algorithm[9] requires manipulation of quantum integers and quantum real and complex numbers that can be represented through a floating-point or fixed-precision encoding. Another example is Hallgren's algorithm[8] for computing the class number of a real quadratic number field. One type of data that occurs in this algorithm, and that must be put into quantum superposition, is the type of ideals in an algebraic number field;

a By "implementing" an algorithm, we mean realizing it as a computer program; we do not mean we have actually run the programs on a quantum computer, although we have run parts of the algorithms on quantum simulators.

Figure 8. A block structure example.

```
mycirc2 :: Qubit -> Qubit -> Qubit -> Circ (Qubit, Qubit, Qubit)
mycirc2 a b c = do
(1) mycirc a b
    with_controls c $ do
(2)   mycirc a b
(3)   mycirc b a
(4) mycirc a c
    return (a,b,c)
```



Specification and verification. In classical programming, there is a variety of techniques for ascertaining the correctness of programs, including compile-time type checking, runtime type checking, formal verification, and debugging. Among them, formal verification is arguably the most reliable but also the most costly. The availability of strong compile-time guarantees requires very carefully designed programming languages. Debugging is cheap and useful and therefore ubiquitous in classical-program development.

In quantum computation, the cost of debugging is likely to be quite high. To begin with, observing a quantum system can change its state. A debugger for a quantum program would therefore necessarily give incomplete information about its state when run on actual quantum hardware. The alternative is to use a quantum simulator for debugging. But this is not practical due to the exponential cost of simulating quantum systems. Moreover, it can be expected that the initial quantum computers will be rare and expensive to run and therefore that the cost of runtime errors in quantum code will initially be much higher than in classical computing. This shifts the cost-benefit analysis for quantum programming toward strong compile-time correctness guarantees, as well as formal specification and verification.

A quantum programming language should have a sound, well-defined semantics permitting mathematical specifications of program behavior and program correctness proofs. It is also beneficial for the language to

Figure 9. A circuit operator example.

```
timestep :: Qubit -> Qubit -> Qubit -> Circ (Qubit, Qubit, Qubit)
timestep a b c = do
    mycirc a b
    qnot c `controlled` (a,b)
    reverse_simple mycirc (a,b)
    return (a,b,c)
```

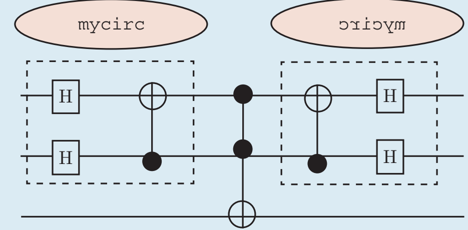


Figure 10. A circuit transformer example.

```
timestep2 :: Qubit -> Qubit -> Qubit -> Circ (Qubit, Qubit, Qubit)
timestep2 = decompose_generic Binary timestep
```

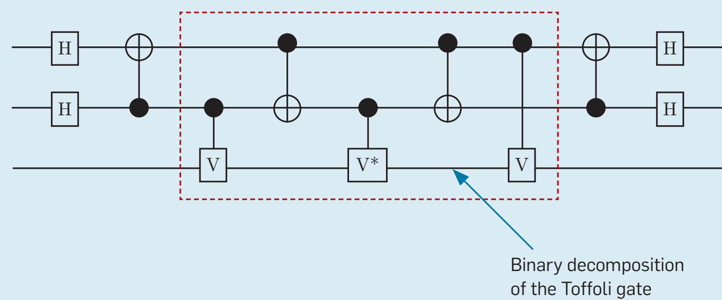


Figure 11. A functional-to-reversible translation example.

```
build_circuit
f :: [Bool] -> Bool
f as = case as of
  [] -> False
  [h] -> h
  h:t -> h 'bool_xor' f t

unpack template_f :: [Qubit] -> Circ Qubit
```

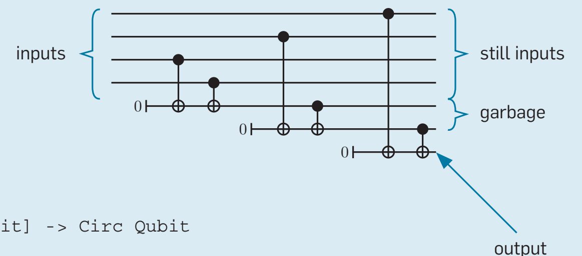


Figure 12. The circuit from Figure 11 made reversible.

```
classical_to_reversible :: (Datable a, QCData b) => (a -> Circ b) -> (a,b) -> Circ (a,b)
classical_to_reversible (unpack template_f)
```

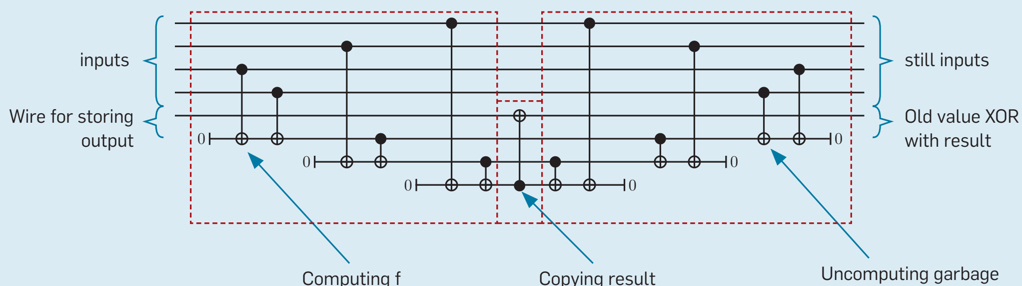


Figure 13. A procedural example.

```
import Quipper

w :: (Qubit,Qubit) -> Circ (Qubit,Qubit)
w = named_gate "W"

toffoli :: Qubit -> (Qubit,Qubit) -> Circ Qubit
toffoli d (x,y) =
   qnot d `controlled` x .==. 1 .&&. y .==. 0

eiz_at :: Qubit -> Qubit -> Circ ()
eiz_at d r =
   named_gate_at "eiZ" d `controlled` r .==. 0

circ :: [(Qubit,Qubit)] -> Qubit -> Circ ()
circ ws r = do
   label (unzip ws,r) (("a","b"),"r")
   with_ancilla $ \d -> do
   mapM_ w ws
   mapM_ (toffoli d) ws
   eiz_at d r
   mapM_ (toffoli d) (reverse ws)
   mapM_ (reverse_generic w) (reverse ws)
   return ()

main = print_generic EPS circ (replicate 3 (qubit,qubit)) qubit
```

have a strong static type system that can guarantee the absence of most runtime errors (such as violations of the no-cloning property of quantum information);[b] and

Resource sensitivity and resource estimation. At first, quantum computers will probably not have many qubits. The language should thus include tools to estimate the resources required for running a particular piece of code (such as number of qubits, number of elementary gates, or other rel-

b The absence of cloning is already guaranteed by the physics, regardless of what the programming language does. However, one could similarly say the absence of an illegal memory access is guaranteed by a classical processor's page-fault mechanism. It is nevertheless desirable to have a programming language that can guarantee *prior to running the program* that the compiled program will never attempt to access an illegal memory location or, in the case of a quantum programming language, will not attempt to apply a controlled-not gate to qubits n and m, where $n = m$.

Figure 14. The circuit generated by the code in Figure 13, with three qubit pairs.

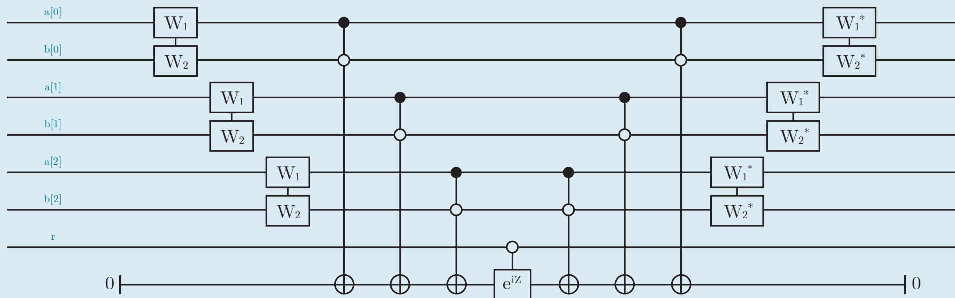
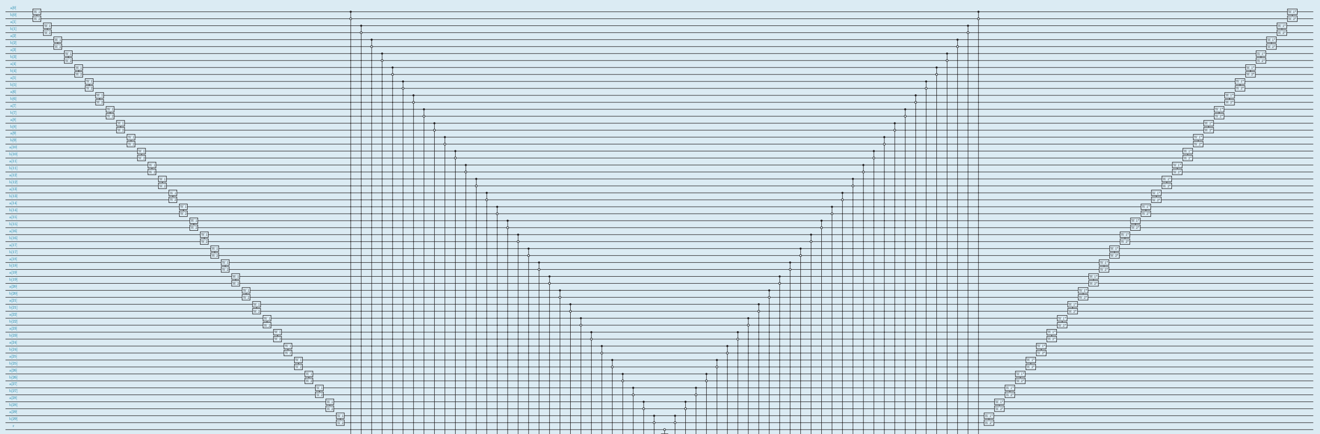


Figure 15. The circuit generated by the code in Figure 13, with 30 qubit pairs.



evant resources) prior to deployment. One particular issue for language designers is how to handle quantum error correction. As the field advances, a decision must be made as to whether error correction should be exposed to the programmer (potentially allowing for optimization by hand) or whether it is more efficient to let the compiler or some other tool apply error correction automatically. Due to the potential for undesirable interactions between quantum error correction (which adds redundancy) and the optimization step of a compiler (which removes redundancy), the design and implementation of any quantum programming language must be aware of the requirements of quantum error correction.

Figure 16. The `calcRweights` function.

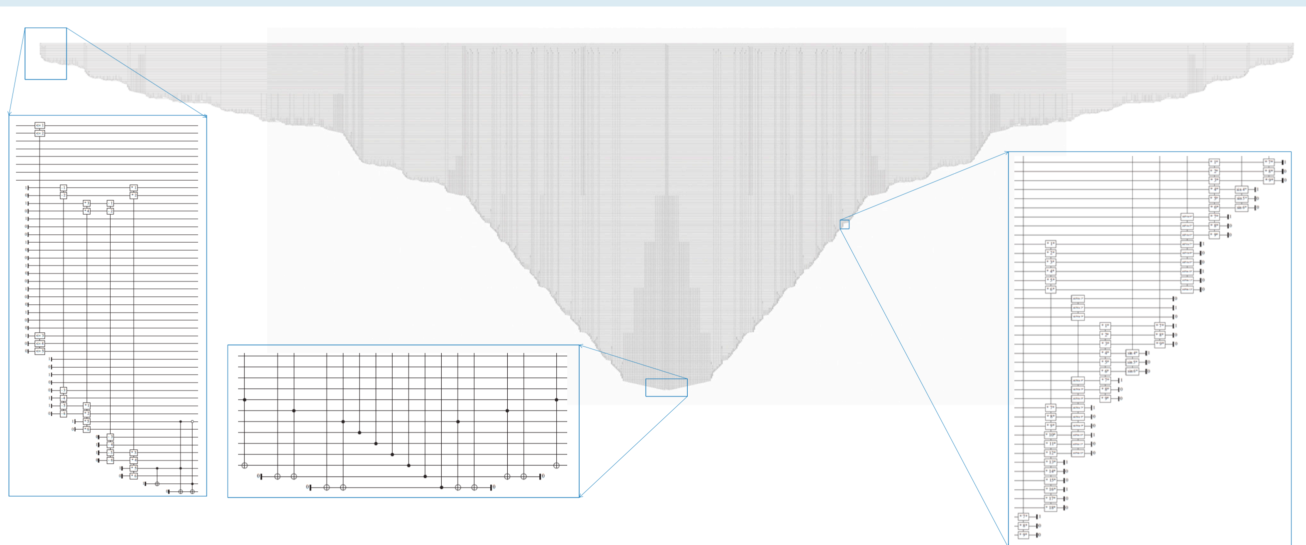
```
calcRweights y nx ny lx ly k theta phi =
     let (xc',yc') = edgetoxy y nx ny in
     let xc = (xc'-1.0)*lx - ((fromIntegral nx)-1.0)*lx/2.0 in
     let yc = (yc'-1.0)*ly - ((fromIntegral ny)-1.0)*ly/2.0 in
     let (xg,yg) = itoxy y nx ny in
     if (xg == nx) then
         let i = (mkPolar ly (k*xc*(cos phi)))*(mkPolar 1.0 (k*yc*(sin phi)))*
                 ((sinc (k*ly*(sin phi)/2.0)) :+ 0.0) in
         let r = ( cos(phi) :+ k*lx )*((cos (theta - phi))/lx :+ 0.0) in i * r
     else if (xg==2*nx-1) then
         let i = (mkPolar ly (k*xc*cos(phi)))*(mkPolar 1.0 (k*yc*sin(phi)))*
                 ((sinc (k*ly*sin(phi)/2.0)) :+ 0.0) in
         let r = ( cos(phi) :+ (- k*lx))*((cos (theta - phi))/lx :+ 0.0) in i * r
     else if ( (yg==1) && (xg<nx) ) then
         let i = (mkPolar lx (k*yc*sin(phi)))*(mkPolar 1.0 (k*xc*cos(phi)))*
                 ((sinc (k*lx*(cos phi)/2.0)) :+ 0.0) in
         let r = ( (- sin phi) :+ k*ly )*((cos(theta - phi))/ly :+ 0.0) in i * r
     else if ( (yg==ny) && (xg<nx) ) then
         let i = (mkPolar lx (k*yc*sin(phi)))*(mkPolar 1.0 (k*xc*cos(phi)))*
                 ((sinc (k*lx*(cos phi)/2.0)) :+ 0.0) in
         let r = ( (- sin phi) :+ (- k*ly) )*((cos(theta - phi)/ly) :+ 0.0) in i * r
     else 0.0 :+ 0.0
```

Prior Work on QPLs

Several quantum programming languages have been developed by researchers around the world.[5] Some, including van Tonder's quantum lambda calculus,[18] are primarily intended as theoretical tools. The first quantum programming language intended for practical use was arguably Ömer's QCL,[14] a C-style imperative language supporting "structured quantum programming." QCL provides simple registers but no high-level quantum data types. It could also benefit from greater support for specification and verification. Partly building on Ömer's work, Bettelli et al.[2] proposed a language that is an extension of C++. The guarded command language qGCL of Sanders and Zuliani[16] hints at a language for program specification.

The first quantum programming language in the style of functional programming was the quantum lambda calculus of Selinger and Valiron,[17] providing a unified framework for manipulating classical and quantum data. The quantum lambda calculus has a well-defined mathematical semantics that guarantees the absence of runtime errors in a well-typed program. The language is easily extended with inductive data types (such as lists) and recursion. One shortcoming of the quantum lambda calculus, however, is that it does not separate circuit construction from circuit evaluation. It thus lacks the ability to manipulate quantum circuits as data, as well as the ability to au-

Figure 17. The `calcRweights` circuit.



tomatically construct unitary circuits from a classical description. These problems were partly addressed by the Quantum IO Monad of Green and Altenkirch,[7] a functional language that is a direct predecessor of Quipper.

The Quipper Language

Building on this previous work, we introduce Quipper, a functional language for quantum computation. We chose to implement Quipper as a deeply embedded domain-specific language (EDSL) inside the host language Haskell; see Gill[6] for an overview of EDSLs and their embedding in Haskell. Quipper is intended to offer a unified general-purpose programming framework for quantum computation. Its main features are:

Hardware independence. Quipper's paradigm is to view quantum computation at the level of logical circuits. The addition of error-correcting codes and mapping to hardware are left to other components further down the chain of compilation;

Extended circuit model. The initialization and termination of qubits is explicitly tracked for the purpose of ancilla management;

Hierarchical circuits. Quipper features subroutines at the circuit level, or "boxed subcircuits," permitting a compact representation of circuits in memory;

Versatile circuit description language. Quipper permits multiple programming styles and can handle both procedural and functional paradigms of computation. It also permits high-level manipulations of circuits with programmable operators;

Two runtimes. A Quipper program typically describes a family of circuits that depends on some classical parameters. The first runtime is "circuit generation," and the second runtime is "circuit execution." In batch-mode operation, as discussed earlier, these two runtimes take place one after the other, whereas in online operation, they may be interleaved;

Parameter/input distinction. Quipper has two notions of classical data: "parameters," which must be known at circuit-generation time, and "inputs," which may be known only at circuit-execution time. For example, the type `Bool` is used for Boolean parameters, and the type `Bit` is used for Boolean inputs;

Extensible data types. Quipper offers an abstract, extensible view of quantum data using Haskell's powerful type-class mechanism; and

Automatic generation of quantum oracles. Many quantum algorithms require some nontrivial classical computation to be made reversible and then lifted to quantum operation. Quipper has facilities for turning an ordinary Haskell program into a reversible circuit. This feature is implemented using a self-referential Haskell feature known as "Template Haskell" that enables a Haskell program to inspect and manipulate its own source code.

Quipper Feature Highlights

We briefly highlight some of Quipper's features with code examples:

Procedural paradigm. In Quipper, qubits are held in variables, and gates are applied one at a time. The type of a circuit-producing function is distinguished by the keyword `Circ` after the arrow, as in Figure 7. The function `mycirc` inputs `a` and `b` of type `Qubit` and outputs a pair of qubits while generating a circuit;

Block structure. Functions generating circuits can be reused as subroutines to generate larger circuits. Operators (such as `with_control`) can take an entire block of code as an argument, as in Figure 8. Note "`do`" introduces an indented block of code, and "`$`" is an idiosyncrasy of Haskell syntax that can be ignored by the reader here;

Circuit operators. Quipper can treat circuits as data and provide high-level operators for manipulating whole circuits. For example, the operator `reverse_simple` reverses a circuit, as in Figure 9;

Circuit transformers. Quipper provides user-programmable "circuit transformers" as a mechanism for modifying a circuit on a gate-by-gate basis. For example, the `timestep` circuit in Figure 9 can be decomposed into binary gates using the `Binary` transformer, as in Figure 10; and

Automated functional-to-reversible translation. Quipper provides a special keyword `build_circuit` for automatically synthesizing a circuit from an ordinary functional program, as in Figure 11. The resulting circuit can be made reversible with the operator `classical_to_reversible`, as in Figure 12.

Experience with Quipper

We have used Quipper to implement seven nontrivial quantum algorithms from the literature, based on documents provided by the Quantum Computer Science program of the U.S. Intelligence Advanced Research Projects Activity (IARPA).[10] All of these algorithms can be run, in the sense that we can print the corresponding circuits for small parameters and perform automated gate counts for circuits of less tractable sizes. Each of these algorithms (see the table here) solves some problem believed to be classically hard, and each algorithm provides an asymptotic quantum speedup, though not necessarily an exponential one. These seven algorithms cover a broad spectrum of quantum techniques; for example, the table includes several algorithms that use the Quantum Fourier Transform, phase estimation, Trotterization, and amplitude amplification. IARPA selected the algorithms for being comparatively complex, which

A selection of quantum algorithms.

Algorithm	Description
Binary Welded Tree[4]	Finds a labeled node in a graph by performing a quantum walk
Boolean Formula[1]	Evaluates an exponentially large Boolean formula using quantum simulation; QCS version computes a winning strategy for the game of Hex
Class Number[8]	Approximates the class group of a real quadratic number field
Ground State Estimation[19]	Computes the ground state energy level of a molecule
Quantum Linear Systems[9]	Solves a very large but sparse system of linear equations
Unique Shortest Vector[15]	Finds the shortest vector in an *n*-dimensional lattice
Triangle Finding[12]	Finds the unique triangle inside a very large dense graph

is why some better-known but technically simpler algorithms (such as Shor's factorization algorithm) were not included.

Using Quipper, we are able to perform semi- or completely automated logical-gate-count estimations for each of the algorithms, even for very large problem sizes. For example, in the case of the triangle-finding algorithm, the command

```
./tf -f gatecount
     -o orthodox
     -n 15 -l 31 -r 6
```

produces the gate count for the complete algorithm on a graph of 2^{15} vertices using an oracle with 31-bit integers and a Hamming graph of tuple size 2^6. This command runs to completion in less than two minutes on a laptop computer and produces a count of 30,189,977,982,990 gates and 4,676 qubits for this instance of the algorithm.

Examples. As a further illustration, here are two subroutines written in Quipper:

Procedural example. First, we formalize the circuit family in Figure 3. This circuit implements the time step for a quantum walk in the Binary Welded Tree algorithm.[4] It inputs a list of pairs of qubits (a_i, b_i), and a single qubit r. It first generates an ancilla, or scratch-space, qubit in state $|0\rangle$. It then applies the two-qubit gate W to each of the pairs (a_i, b_i), followed by a series of doubly controlled NOT-gates acting on the ancilla. After a middle gate e^{iZt}, it applies all the gates in reverse order. The ancilla ends up in the state $|0\rangle$ and is no longer needed. The Quipper code is in Figure 13, yielding the circuit in Figure 14. If one replaces the "3" with a "30" in the main function, one obtains a larger instance of this circuit family, as in Figure 15; and

A functional-to-reversible translation. This example is from the Quantum Linear Systems algorithm.[9] Among other things, this algorithm contains an oracle calculating a vector r of complex numbers; Figure 16 shows the core function of the oracle. Note it relies heavily on algebraic and transcendental operations on real and complex numbers (such as `sin`, `cos`, `sinc`, and `mkPolar`), as well as on subroutines (such as `edgetoxy` and `itoxy`) not shown in Figure 16. This function is readily processed using Quipper's automated circuit-generation facilities. Algebraic and transcendental functions are mapped automatically to quantum versions provided by an existing Quipper library for fixed-point real and complex arithmetic. The result is the rather large circuit in Figure 17.

Conclusion

Practical quantum computation requires a tool chain extending from abstract algorithm descriptions down to the level of physical particles. Quantum programming languages are an important aspect of this tool chain. Ideally, such a language enables a quantum algorithm to be expressed at a high level of abstraction, similar to what may be found in a research paper, and translates it down to a logical circuit. We view this logical circuit as an intermediate representation that can then be further processed by other tools, adding quantum control and error correction, and finally passed to a real-time system controlling physical operations.

Quipper is an example of a language suited to a quantum coprocessor model. We demonstrated Quipper's feasibility by implementing several large-scale algorithms. The design of Quipper solved some major programming-language challenges associated with quantum computation, but there is still much to do, particularly in the design of type systems for quantum computing. As an embedded language, Quipper is confined to using Haskell's type system, providing many important safety guarantees. However, due to Haskell's lack of support for linear types, some safety properties (such as the absence of attempts to clone quantum information) are not adequately supported. The design of ever better type systems for quantum computing is the subject of ongoing research. C

References

1. Ambainis, A., Childs, A.M., Reichardt, B.W., Špalek, R., and Zhang, S. Any AND-OR formula of size N can be evaluated in time $N^{1/2+o(1)}$ on a quantum computer. *SIAM Journal on Computing 39*, 2 (2010), 2513–2530.
2. Bettelli, S., Calarco, T., and Serafini, L. Toward an architecture for quantum programming. *The European Physical Journal D 25*, 2 (2003), 181–200.
3. Burham, H., Durr, C., Heiligman, M., Hoyer, P., Magniez, F., Santha, M., and de Wolf, R. Quantum algorithms for element distinctness. In *Proceedings of the 16th Annual IEEE Conference on Computational Complexity* (Chicago, June 18–21). IEEE Computer Society Press, 2001, 131–137.
4. Childs, A. M., Cleve, R., Deotto, E., Farhi, E., Gutmann, S., and Spielman, D.A. Exponential algorithmic speedup by a quantum walk. In *Proceedings of the 35th Annual ACM Symposium on Theory of Computing* (San Diego, CA, June 9–11). ACM Press, New York, 2003, 59–68.
5. Gay, S.J. Quantum programming languages: Survey and bibliography. *Mathematical Structures in Computer Science 16*, 4 (2006), 581–600.
6. Gill, A. Domain-specific languages and code synthesis using Haskell. *Commun. ACM 57*, 6 (June 2014), 42–49.
7. Green, A. and Altenkirch, T. The quantum IO monad. In *Semantic Techniques in Quantum Computation*, S. Gay and I. Mackie, Eds. Cambridge University Press, Cambridge, U.K., 2009, 173–205.
8. Hallgren, S. Polynomial-time quantum algorithms for Pell's equation and the principal ideal problem. *Journal of the ACM 54*, 1 (Mar. 2007), 4:1–4:19.
9. Harrow, A.W., Hassidim, A., and Lloyd, S. Quantum algorithm for solving linear systems of equations. *Physical Review Letters 103*, 15 (Oct. 2009), 150502-1–150502-4.
10. IARPA Quantum Computer Science Program. *Broad Agency Announcement IARPA-BAA-10-02*, Apr. 2010; https://www.fbo.gov/notices/637e87ac1274d030ce2ab69339ccf93c
11. Knill, E.H. *Conventions for Quantum Pseudocode. Technical Report LAUR-96-2724.* Los Alamos National Laboratory, Los Alamos, NM, 1996.
12. Magniez, F., Santha, M., and Szegedy, M. Quantum algorithms for the triangle problem. *SIAM Journal on Computing 37*, 2 (2007), 413–424.
13. Meter, R.V. and Horsman, C. A blueprint for building a quantum computer. *Commun. ACM 56*, 10 (Oct. 2013), 84–93.
14. Ömer, B. *Quantum Programming in QCL.* Master's Thesis. Institute of Information Systems, Technical University of Vienna, Vienna, Austria, 2000; tph.tuwien.ac.at/~oemer/qcl.html
15. Regev, O. Quantum computation and lattice problems. *SIAM Journal on Computing 33*, 3 (2004), 738–760.
16. Sanders, J.W. and Zuliani, P. Quantum programming. In *Proceedings of the Fifth International Conference on Mathematics of Program Construction, Vol. 1837 of Lecture Notes in Computer Science* (Ponte de Lima, Portugal, July 3–5). Springer-Verlag, Berlin Heidelberg, 2000, 80–99.
17. Selinger, P. and Valiron, B. A lambda calculus for quantum computation with classical control. *Mathematical Structures in Computer Science 16*, 3 (2006), 527–552.
18. van Tonder, A. A lambda calculus for quantum computation. *SIAM Journal of Computation 33*, 5 (2004), 1109–1135.
19. Whitfield, J.D., Biamonte, J., and Aspuru-Guzik, A. Simulation of electronic structure Hamiltonians using quantum computers. *Molecular Physics 109*, 5 (Mar. 2011), 735–750.

Benoît Valiron (benoit.valiron@monoidal.net) is an assistant professor in the engineering school CentraleSupélec and a researcher in the Computer Science Laboratory of the Université Paris Sud, Paris, France.

Neil J. Ross (neil.jr.ross@gmail.com) is a Ph.D. candidate at Dalhousie University, Halifax, Nova Scotia, Canada.

Peter Selinger (selinger@mathstat.dal.ca) is a professor of mathematics at Dalhousie University, Halifax, Nova Scotia, Canada.

D. Scott Alexander (salexander@appcomsci.com) is a chief scientist at Applied Communication Science, Basking Ridge, NJ.

Jonathan M. Smith (jms@cis.upenn.edu) is the Olga and Alberico Pompa Professor of Engineering and Applied Science and a professor of computer and information science at the University of Pennsylvania, Philadelphia, PA.

Copyright held by authors.
Publication rights licensed to ACM. $15.00

Watch the authors discuss their work in this exclusive *Communications* video. http://cacm.acm.org/videos/programming-the-quantum-future