Emerging languages and representations for quantum computing

Monday, October 5, 2020

Rutgers University

Yipeng Huang

Progress and questions about QAOA on Cirq?

https://rutgers.instructure.com/courses/73314/assignments/1017995

Position statement for this graduate seminar

- Quantum computer engineering has become important.

- Requires computer systems expertise beyond quantum algorithms and quantum device physics.

All the quantum computer abstractions we don’t yet have right now

0. Quantum computer support for quantum computer engineering
1. Fault-tolerant, error-corrected quantum algorithms
2. ***Mature, high level quantum programming languages***
3. Universally accepted quantum ISAs
4. Uniform, fully connected quantum device architectures
5. Reliable quantum gates and qubits

Quantum programming and correctness

I. Correct quantum programs
   A. Verification (proofs)
   B. Validation (debugging & assertions)

II. Quantum programming: from abstractions to execution
   A. Creation / discovery of algorithms
   B. Specification of algorithm as a procedure
   C. Compilation to native instructions while maximizing performance
   D. Execution in target machine with facilities for validation

Quantum programming and correctness

I. ***Correct quantum programs***
   A. Verification (proofs)
   B. Validation (debugging & assertions)

II. Quantum programming: from abstractions to execution
   A. Creation / discovery of algorithms
   B. Specification of algorithm as a procedure
   C. Compilation to native instructions while maximizing performance
   D. Execution in target machine with facilities for validation

Approaches to Software Reliability

- Social
  - Code reviews
  - Extreme/Pair programming
- Methodological
  - Design patterns
  - Test-driven development
  - Version control
  - Bug tracking
- Technological
  - "lint" tools, static analysis
  - Fuzzers, random testing
- Mathematical
  - Sound type systems
  - Formal verification

Less "formal": Lightweight, inexpensive techniques (that may miss problems)

This isn't an either/or tradeoff… a spectrum of methods is needed!

Even the most "formal" argument can still have holes:

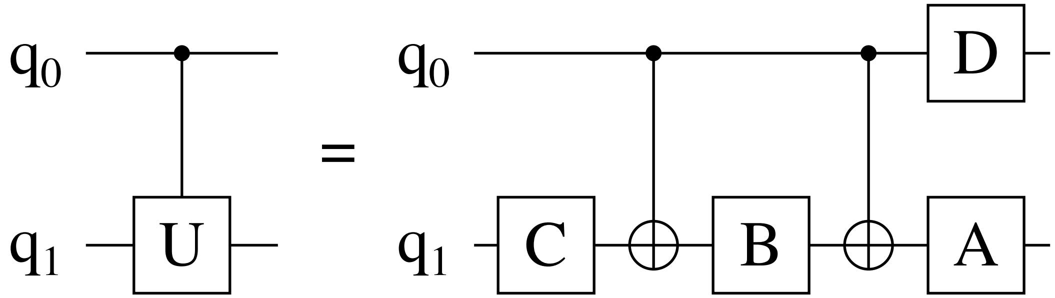
- Did you prove the right thing?
- Do your assumptions match reality?
- Knuth: *"Beware of bugs in the above code; I have only proved it correct, not tried it."*

More "formal": eliminate *with certainty* as many problems as possible.

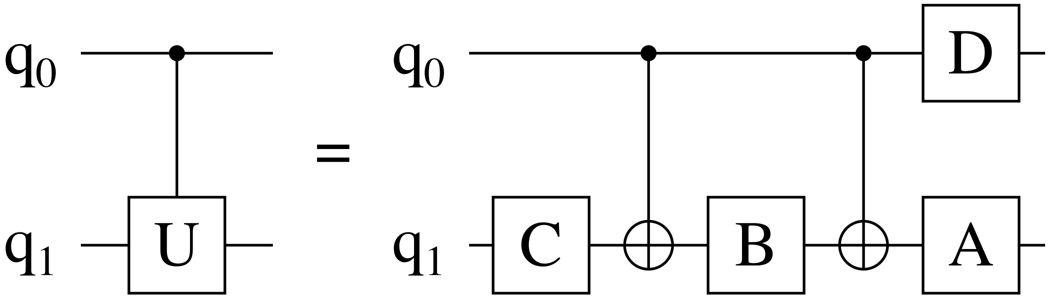
From: https://www.seas.upenn.edu/~cis500/current/lectures/lec01.pdf

- “Exception handling” (i.e., quantum error correction) is costly.
- No printf. No intermediate measurements.
- Can’t set arbitrary breakpoints.
- Few obvious assertions.

Even simple quantum programming bugs lead to non-obvious symptoms

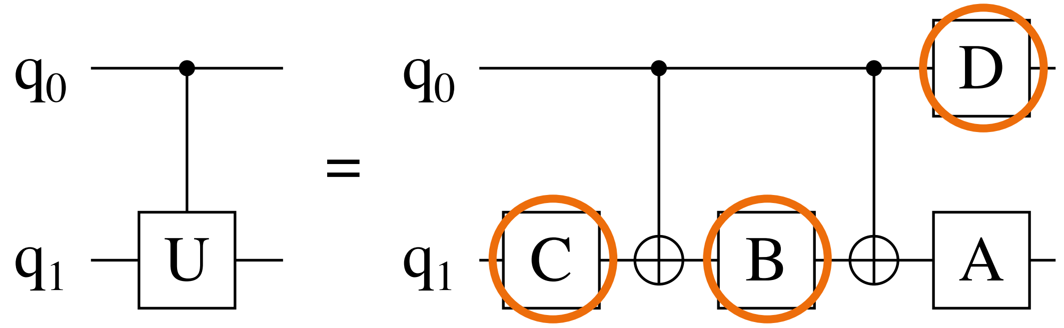


Even simple quantum programming bugs lead to non-obvious symptoms




```
Rz(q1, +angle/2); // C
CNOT(q0, q1);
Rz(q1, -angle/2); // B
CNOT(q0, q1);
Rz(q0, +angle/2); // D
```

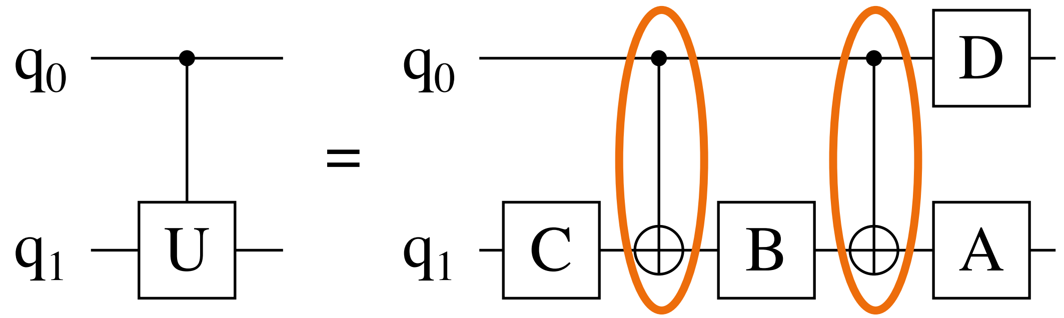
Even simple quantum programming bugs lead to non-obvious symptoms



Elementary single-qubit operations

```
Rz(q1, +angle/2); // C
CNOT(q0, q1);
Rz(q1, -angle/2); // B
CNOT(q0, q1);
Rz(q0, +angle/2); // D
```

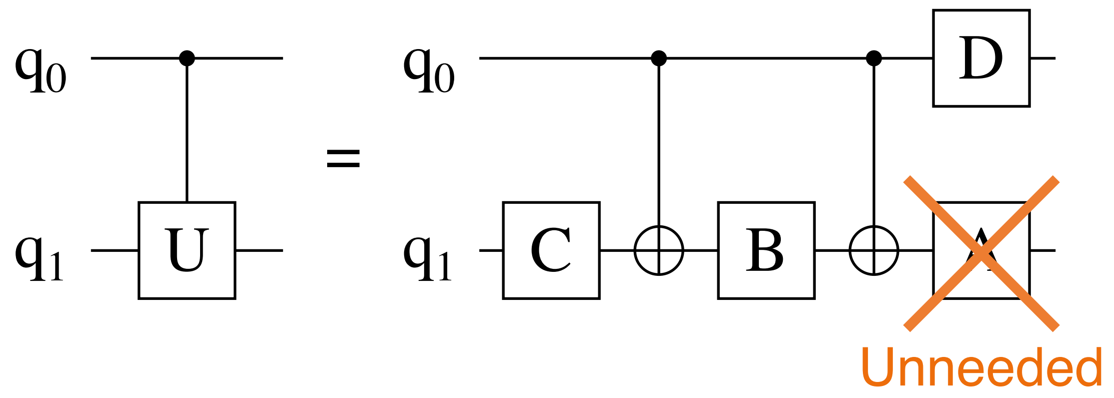
Even simple quantum programming bugs lead to non-obvious symptoms



Elementary two-qubit operations

`Rz(q1, +angle/2); // C`
`CNOT(q0, q1);`
`Rz(q1, -angle/2); // B`
`CNOT(q0, q1);`
`Rz(q0, +angle/2); // D`
 	
Correct,
operation A unneeded	

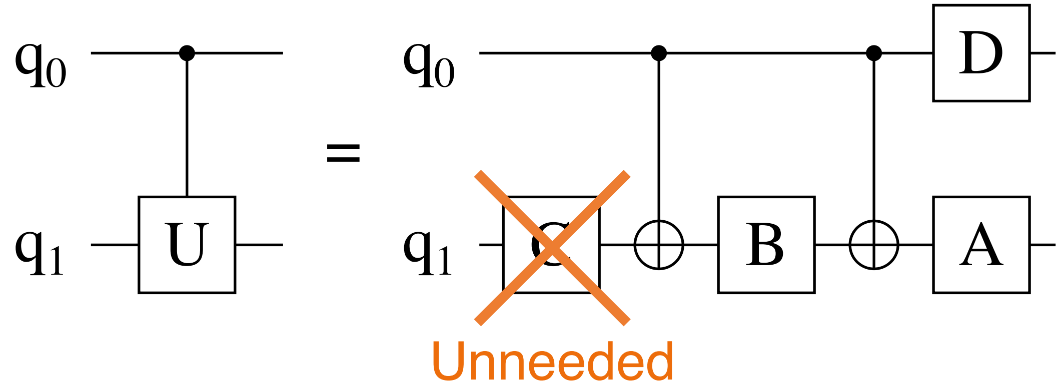
Even simple quantum programming bugs lead to non-obvious symptoms




```
Rz(q1, +angle/2); // C
CNOT(q0, q1);
Rz(q1, -angle/2); // B
CNOT(q0, q1);
Rz(q0, +angle/2); // D
```

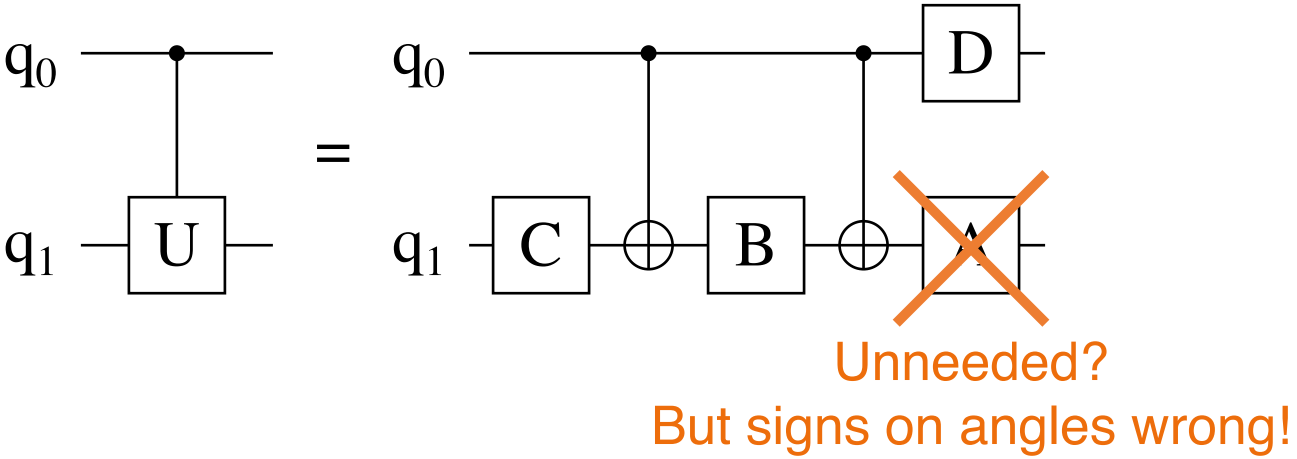
Correct,
operation A unneeded

Even simple quantum programming bugs lead to non-obvious symptoms



`Rz(q1, +angle/2); // C`
`CNOT(q0, q1);`
`Rz(q1, -angle/2); // B`
`CNOT(q0, q1);`
`Rz(q0, +angle/2); // D`
 	
Correct, operation A unneeded
	`CNOT(q0, q1);`
	`Rz(q1, -angle/2); // B`
	`CNOT(q0, q1);`
	`Rz(q1, +angle/2); // A`
	`Rz(q0, +angle/2); // D`
	
	Correct, operation C unneeded	

Even simple quantum programming bugs lead to non-obvious symptoms

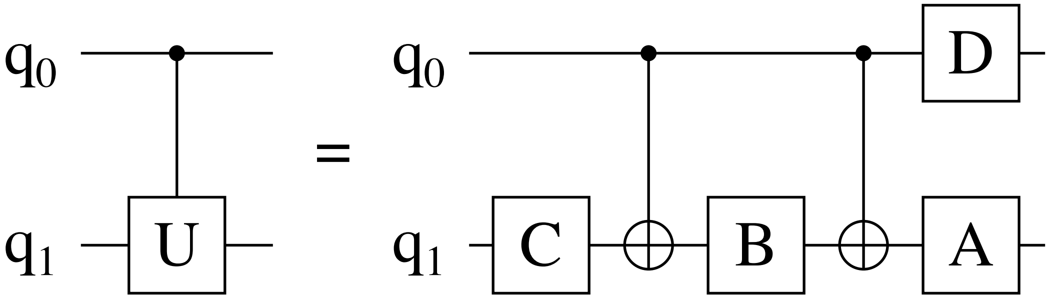


`Rz(q1, +angle/2); // C`
`CNOT(q0, q1);`
`Rz(q1, -angle/2); // B`
`CNOT(q0, q1);`
`Rz(q0, +angle/2); // D`
	`CNOT(q0, q1);`
	`Rz(q1, -angle/2); // B`
	`CNOT(q0, q1);`
	`Rz(q1, +angle/2); // A`
	`Rz(q0, +angle/2); // D`
		`Rz(q1, -angle/2);`
		`CNOT(q0, q1);`
		`Rz(q1, +angle/2);`
		`CNOT(q0, q1);`
		`Rz(q0, +angle/2); // D`

Correct, operation A unneeded	**Correct, operation C unneeded**	**Incorrect, angles flipped**

Many ways to translate basic quantum operations to program code—many details to get right!

Even simple quantum programming bugs lead to non-obvious symptoms



<table>
<tr>
<td>

```
Rz(q1, +angle/2); // C
CNOT(q0, q1);
Rz(q1, -angle/2); // B
CNOT(q0, q1);
Rz(q0, +angle/2); // D
```

**Correct,
operation A unneeded**

$|11\rangle \rightarrow e^{i*angle}|11\rangle$
</td>
<td>

```
CNOT(q0, q1);
Rz(q1, -angle/2); // B
CNOT(q0, q1);
Rz(q1, +angle/2); // A
Rz(q0, +angle/2); // D
```

**Correct,
operation C unneeded**

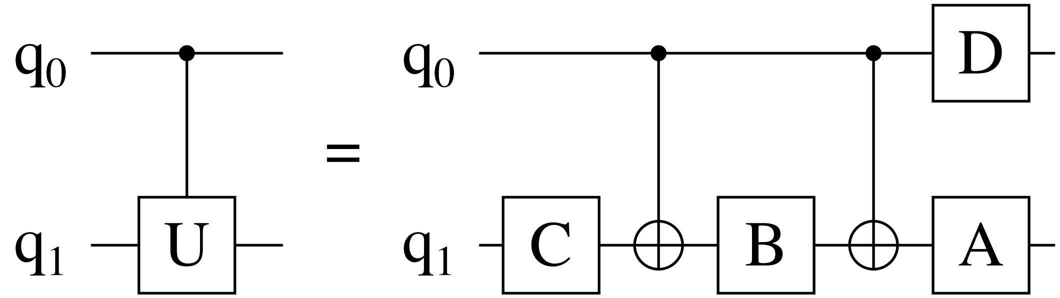
$|11\rangle \rightarrow e^{i*angle}|11\rangle$
</td>
<td>

```
Rz(q1, -angle/2);
CNOT(q0, q1);
Rz(q1, +angle/2);
CNOT(q0, q1);
Rz(q0, +angle/2); // D
```

**Incorrect,
angles flipped**

$|11\rangle \rightarrow e^{-i*angle}|11\rangle$
</td>
</tr>
</table>

Even simple quantum programming bugs lead to non-obvious symptoms




```
Rz(q1, +angle/2); // C
CNOT(q0, q1);
Rz(q1, -angle/2); // B
CNOT(q0, q1);
Rz(q0, +angle/2); // D
```

Correct, operation A unneeded

$|11\rangle \rightarrow e^{i*angle}|11\rangle$

```
CNOT(q0, q1);
Rz(q1, -angle/2); // B
CNOT(q0, q1);
Rz(q1, +angle/2); // A
Rz(q0, +angle/2); // D
```

Correct, operation C unneeded

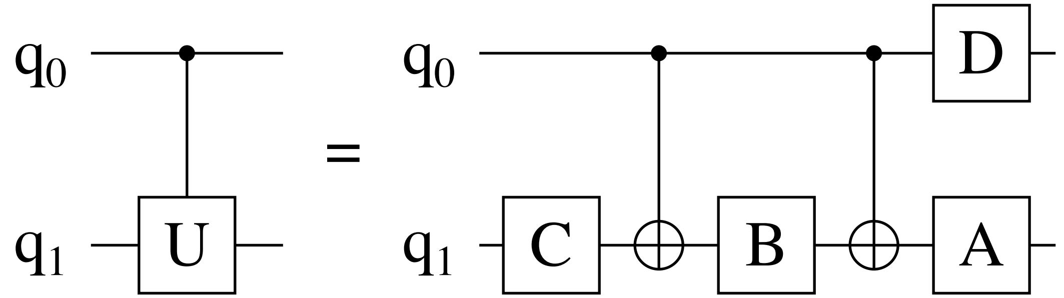
$|11\rangle \rightarrow e^{i*angle}|11\rangle$

```
Rz(q1, -angle/2);
CNOT(q0, q1);
Rz(q1, +angle/2);
CNOT(q0, q1);
Rz(q0, +angle/2); // D
```

Incorrect, angles flipped

$|11\rangle \rightarrow e^{-i*angle}|11\rangle$

Even simple quantum programming bugs lead to non-obvious symptoms



`Rz(q1, +angle/2); // C`
`CNOT(q0, q1);`
`Rz(q1, -angle/2); // B`
`CNOT(q0, q1);`
`Rz(q0, +angle/2); // D`
	`CNOT(q0, q1);`
	`Rz(q1, -angle/2); // B`
	`CNOT(q0, q1);`
	`Rz(q1, +angle/2); // A`
	`Rz(q0, +angle/2); // D`
		`Rz(q1, -angle/2);`
		`CNOT(q0, q1);`
		`Rz(q1, +angle/2);`
		`CNOT(q0, q1);`
		`Rz(q0, +angle/2); // D`

Correct, operation A unneeded	**Correct, operation C unneeded**	**Incorrect, angles flipped**
$\|11\rangle \rightarrow e^{i*angle}\|11\rangle$	$\|11\rangle \rightarrow e^{i*angle}\|11\rangle$	$\|11\rangle \rightarrow e^{-i*angle}\|11\rangle$

- “Exception handling” (i.e., quantum error correction) is costly.
- No printf. No intermediate measurements.
- Can’t set arbitrary breakpoints.
- Few obvious assertions.

I. Correct quantum programs

   A. ***Verification (proofs)***
   B. Validation (debugging & assertions)

Approaches to Software Reliability

- Social
  - Code reviews
  - Extreme/Pair programming
- Methodological
  - Design patterns
  - Test-driven development
  - Version control
  - Bug tracking
- Technological
  - “lint” tools, static analysis
  - Fuzzers, random testing
- Mathematical
  - Sound type systems
  - Formal verification

Less “formal”: Lightweight, inexpensive techniques (that may miss problems)

This isn’t an either/or tradeoff… a spectrum of methods is needed!

Even the most “formal” argument can still have holes:

- Did you prove the right thing?
- Do your assumptions match reality?
- Knuth: *“Beware of bugs in the above code; I have only proved it correct, not tried it.”*

More “formal”: eliminate *with certainty* as many problems as possible.

[“Formal Verification vs. Quantum Uncertainty” by Rand, Hietala, and Hicks]

From: https://www.seas.upenn.edu/~cis500/current/lectures/lec01.pdf

II. Correct quantum programs
   A. Verification (proofs)
   B. ***Validation (debugging & assertions)***

Approaches to Software Reliability

We talk about three papers on these approaches

- Social
  - Code reviews
  - Extreme/Pair programming
- Methodological
  - Design patterns
  - Test-driven development
  - Version control
  - Bug tracking
- Technological
  - “lint” tools, static analysis
  - Fuzzers, random testing
- Mathematical
  - Sound type systems
  - Formal verification

Less “formal”: Lightweight, inexpensive techniques (that may miss problems)

This isn’t an either/or tradeoff… a spectrum of methods is needed!

Even the most “formal” argument can still have holes:

- Did you prove the right thing?
- Do your assumptions match reality?
- Knuth: *“Beware of bugs in the above code; I have only proved it correct, not tried it.”*

More “formal”: eliminate *with certainty* as many problems as possible.

From: https://www.seas.upenn.edu/~cis500/current/lectures/lec01.pdf

Quantum program bug types

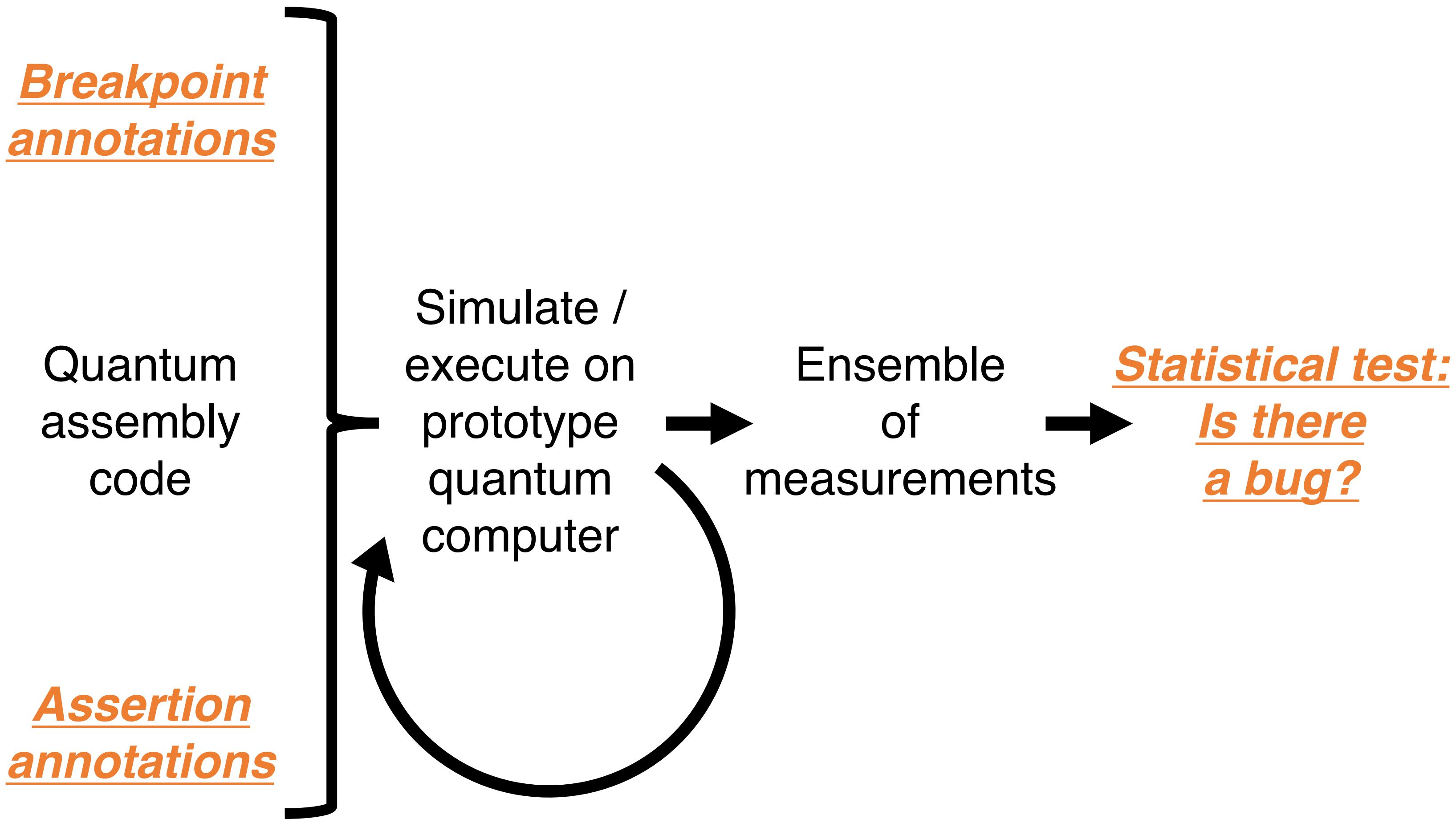
1. Quantum initial values
2. Basic operations
3. Composing operations
   A. Iteration
   B. Mirroring
4. Classical input parameters
5. Garbage collection of qubits

Defenses, debugging, and assertions

1. Preconditions
2. Subroutines / unit tests
3. Quantum specific language support
   A. Numeric data types
   B. Reversible computation
4. Algorithm progress assertions
5. Postconditions

A first taxonomy of quantum program bugs and defenses.

Toolchain for debugging programs with tests on measurements



Detailed debugging of Shor's factorization algorithm

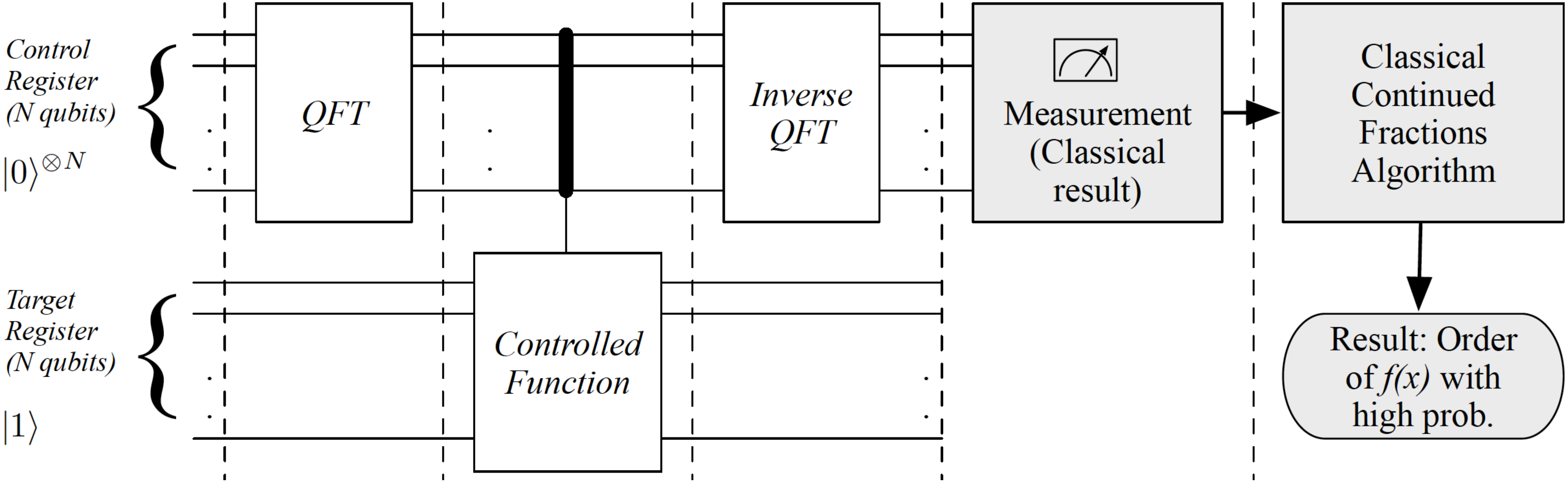


Image credit: Metodi, Faruque, and Chong, Quantum Computing for Computer Architects, 2nd Ed., p26

Detailed debugging of Shor's factorization algorithm

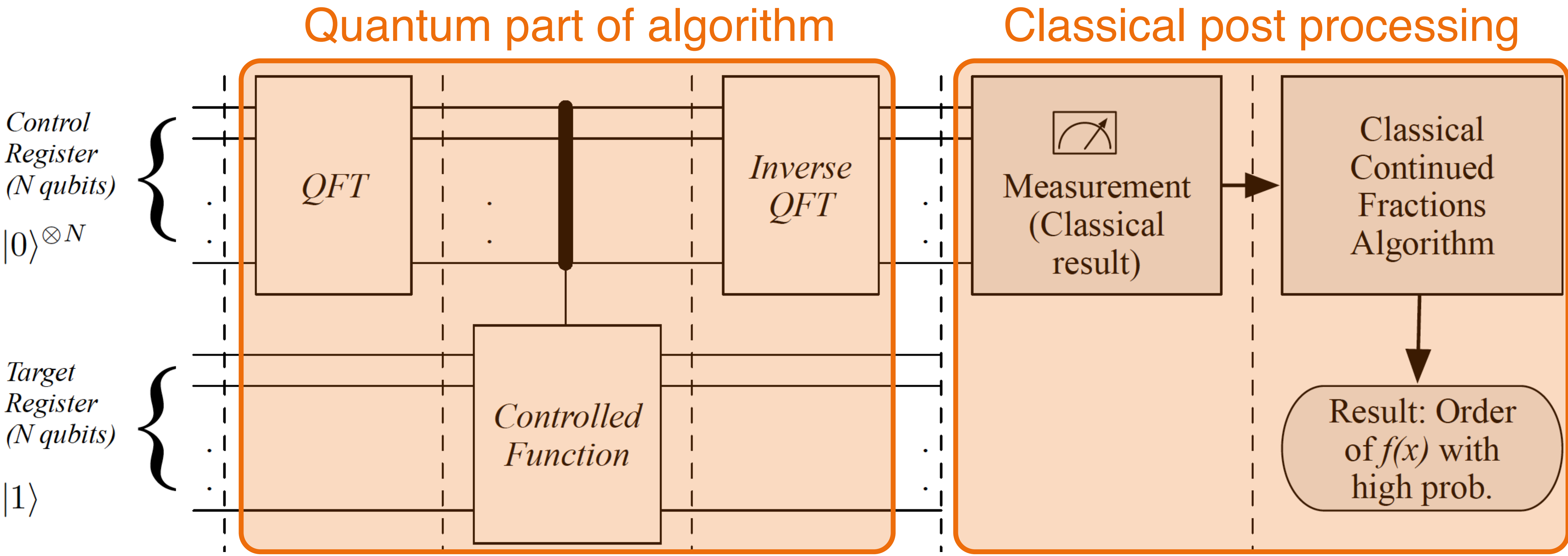
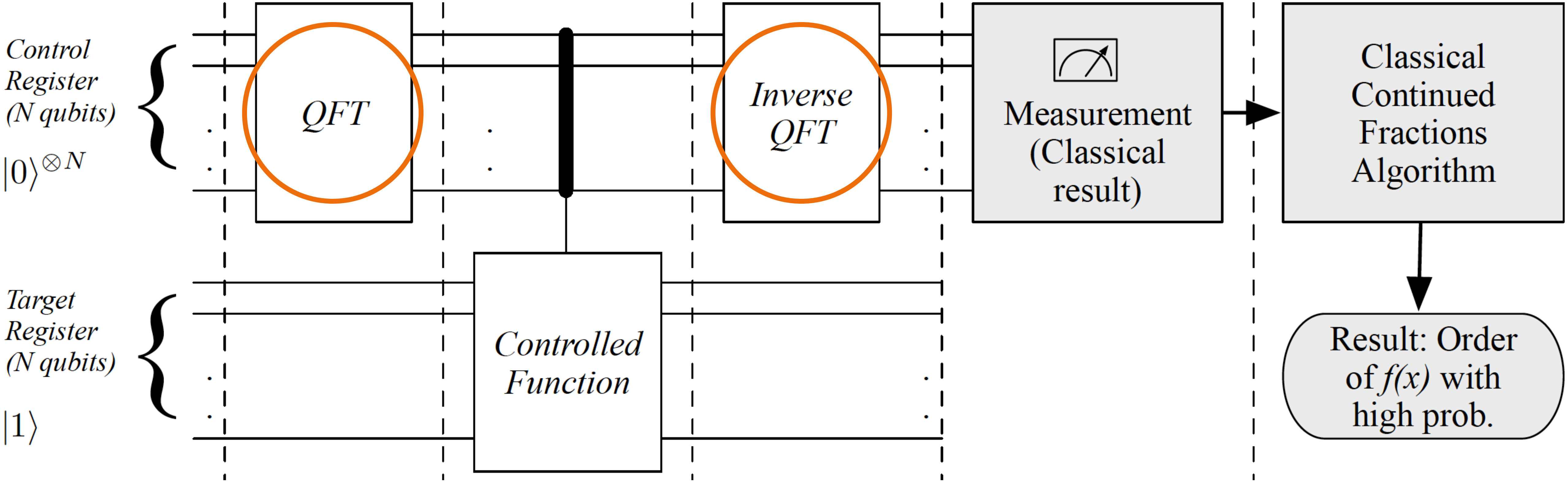


Image credit: Metodi, Faruque, and Chong, Quantum Computing for Computer Architects, 2nd Ed., p26

Bug type 3-B: mistake in composing gates using mirroring

Mirror image submodules

Mirror image submodules



```
#include "QFT.scaffold"
#define width 4 // number of qubits
int main () {

  // initialize quantum variable to 5
  qbit reg[width];
  for ( int i=0; i<width; i++ ) {
    PrepZ ( reg[i], (i+1)%2 ); // 0b0101
  }

  // precondition for QFT:
  assert_classical ( reg, width, 5 );

  QFT ( width, reg );

  // postcondition for QFT &
  // precondition for iQFT:
  assert_superposition ( reg, width );

  iQFT ( width, reg );

  // postcondition for iQFT:
  assert_classical ( reg, width, 5 );
}
```

Listing 1: Test harness for quantum Fourier transform.

Assertions on classical & superposition states help us decide whether programs are correct

Testbench for quantum Fourier transform, consisting of controlled-rotations

```
#include "QFT.scaffold"
#define width 4 // number of qubits
int main () {

  // initialize quantum variable to 5
  qbit reg[width];
  for ( int i=0; i<width; i++ ) {
    PrepZ ( reg[i], (i+1)%2 ); // 0b0101
  }

  // precondition for QFT:
  assert_classical ( reg, width, 5 );

  QFT ( width, reg );

  // postcondition for QFT &
  // precondition for iQFT:
  assert_superposition ( reg, width );

  iQFT ( width, reg );

  // postcondition for iQFT:
  assert_classical ( reg, width, 5 );
}
```

Listing 1: Test harness for quantum Fourier transform.

Assertions on classical & superposition states help us decide whether programs are correct

Testbench for quantum Fourier transform, consisting of controlled-rotations

QFT and iQFT should be inverses, but bug in controlled-rotations would lead to flawed inversion

```
#include "QFT.scaffold"
#define width 4 // number of qubits
int main () {

  // initialize quantum variable to 5
  qbit reg[width];
  for ( int i=0; i<width; i++ ) {
    PrepZ ( reg[i], (i+1)%2 ); // 0b0101
  }

  // precondition for QFT:
  assert_classical ( reg, width, 5 );

  QFT ( width, reg );

  // postcondition for QFT &
  // precondition for iQFT:
  assert_superposition ( reg, width );

  iQFT ( width, reg );

  // postcondition for iQFT:
  assert_classical ( reg, width, 5 );
}
```

Listing 1: Test harness for quantum Fourier transform.

Assertions on classical & superposition states help us decide whether programs are correct

Testbench for quantum Fourier transform, consisting of controlled-rotations

QFT and iQFT should be inverses, but bug in controlled-rotations would lead to flawed inversion

```
#include "QFT.scaffold"
#define width 4 // number of qubits
int main () {

  // initialize quantum variable to 5
  qbit reg[width];
  for ( int i=0; i<width; i++ ) {
    PrepZ ( reg[i], (i+1)%2 ); // 0b0101
  }

  // precondition for QFT:
  assert_classical ( reg, width, 5 );

  QFT ( width, reg );

  // postcondition for QFT &
  // precondition for iQFT:
  assert_superposition ( reg, width );

  iQFT ( width, reg );

  // postcondition for iQFT:
  assert_classical ( reg, width, 5 );
}
```

Listing 1: Test harness for quantum Fourier transform.

Assertions on classical & superposition states help us decide whether programs are correct

Testbench for quantum Fourier transform, consisting of controlled-rotations

QFT and iQFT should be inverses, but bug in controlled-rotations would lead to flawed inversion

Flawed inversion caught in failure of classical assertion based on Chi-squared tests

```
#include "QFT.scaffold"
#define width 4 // number of qubits
int main () {

  // initialize quantum variable to 5
  qbit reg[width];
  for ( int i=0; i<width; i++ ) {
    PrepZ ( reg[i], (i+1)%2 ); // 0b0101
  }

  // precondition for QFT:
  assert_classical ( reg, width, 5 );

  QFT ( width, reg );

  // postcondition for QFT &
  // precondition for iQFT:
  assert_superposition ( reg, width );

  iQFT ( width, reg );

  // postcondition for iQFT:
  assert_classical ( reg, width, 5 );
}
```

Listing 1: Test harness for quantum Fourier transform.

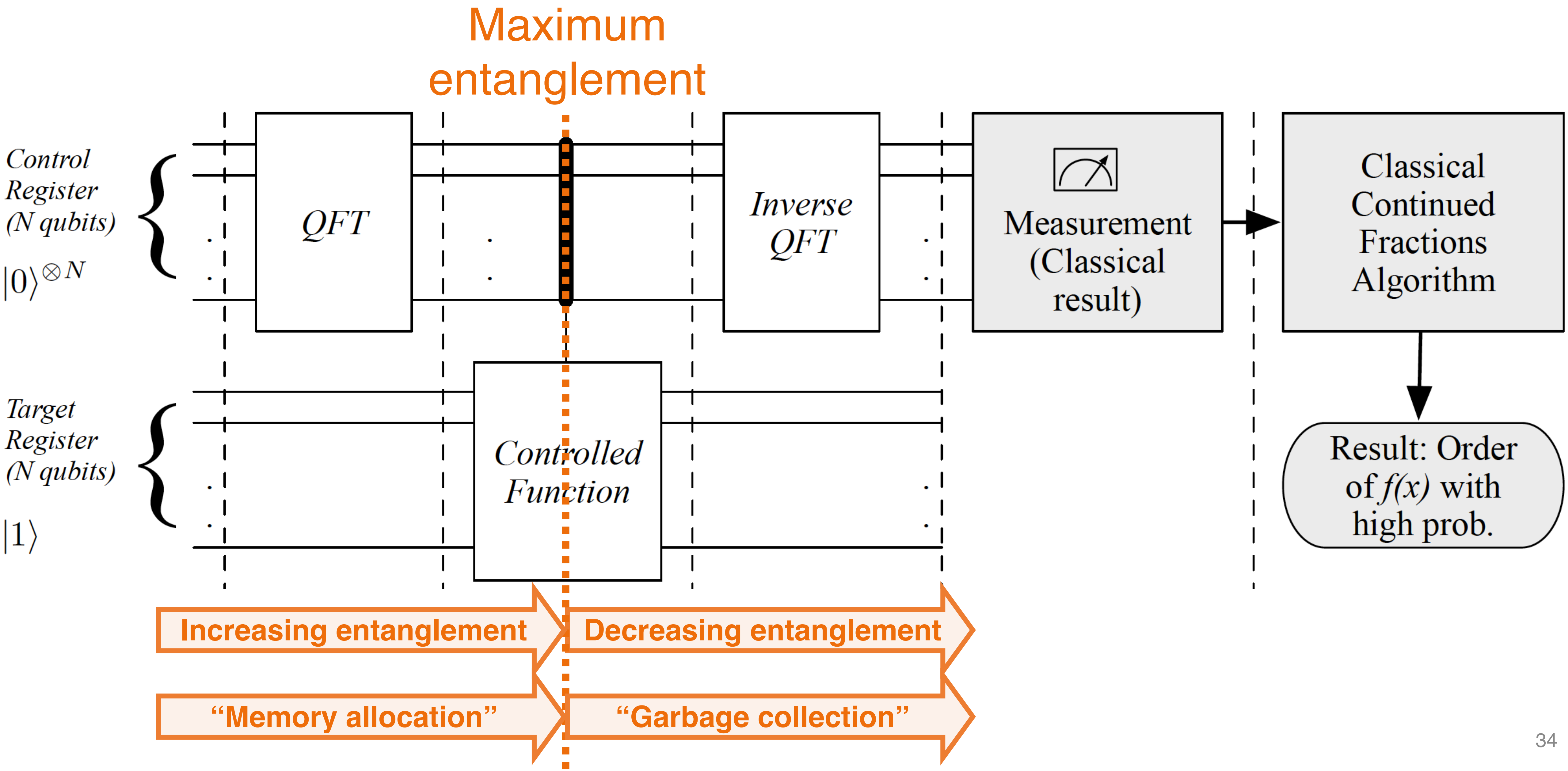
Assertions on classical & superposition states help us decide whether programs are correct

Testbench for quantum Fourier transform, consisting of controlled-rotations

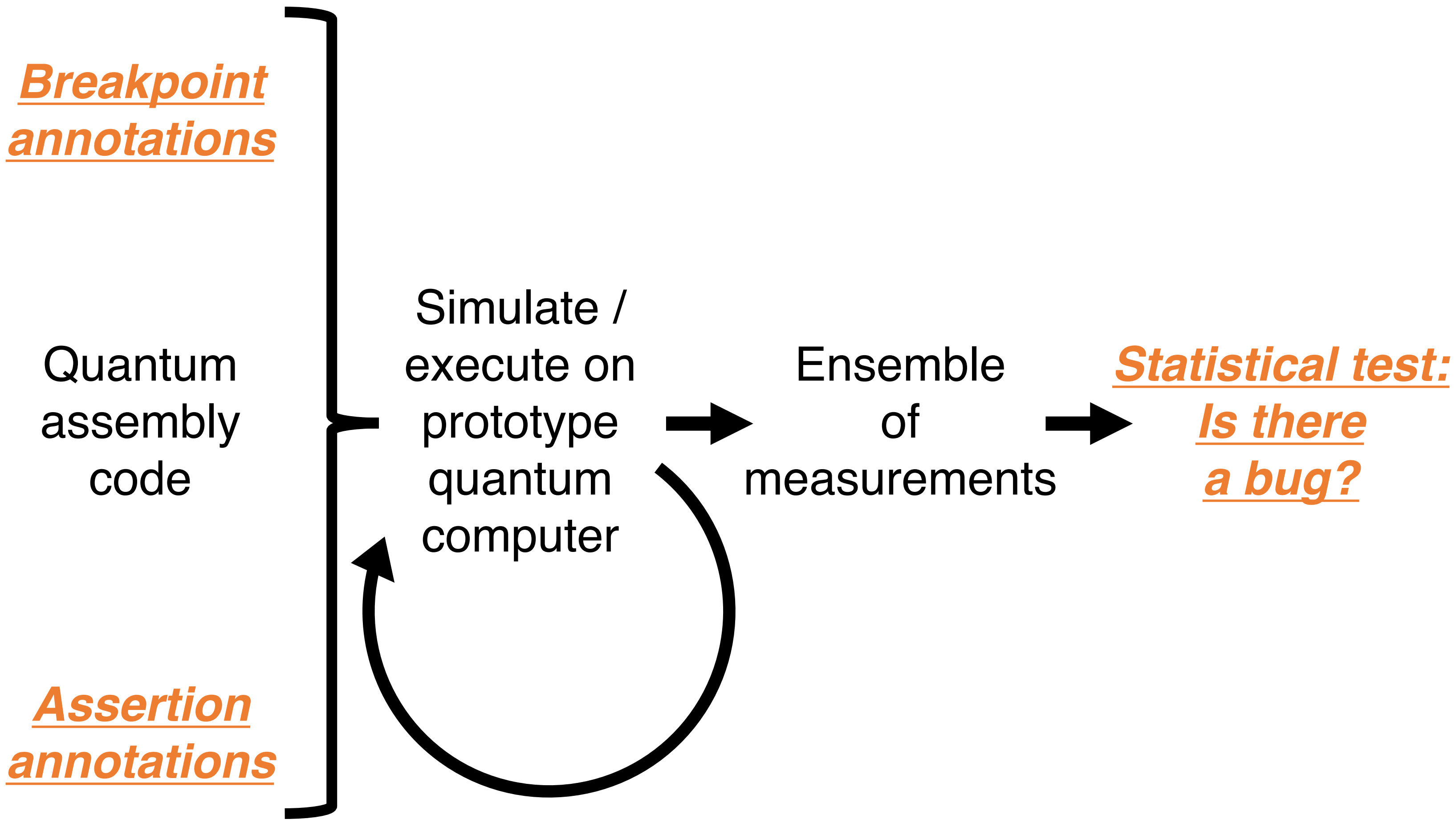
QFT and iQFT should be inverses, but bug in controlled-rotations would lead to flawed inversion

Flawed inversion caught in failure of classical assertion based on Chi-squared tests

Bug type 3-B: mistake in composing gates using mirroring



Toolchain for debugging programs with tests on measurements



Quantum programming and correctness

I. Correct quantum programs
   A. Verification (proofs)
   B. Validation (debugging & assertions)

II. ***Quantum programming: from abstractions to execution***
   A. Creation / discovery of algorithms
   B. Specification of algorithm as a procedure
   C. Compilation to native instructions while maximizing performance
   D. Execution in target machine with facilities for validation

Provide abstractions for easy programming

II. Quantum programming: from abstractions to execution

A. Creation / discovery of algorithms

B. Specification of algorithm as a procedure

C. Compilation to native instructions while maximizing performance

D. Execution in target machine with facilities for validation

Maximize performance and ensure correctness in hardware

II. Quantum programming: from abstractions to execution

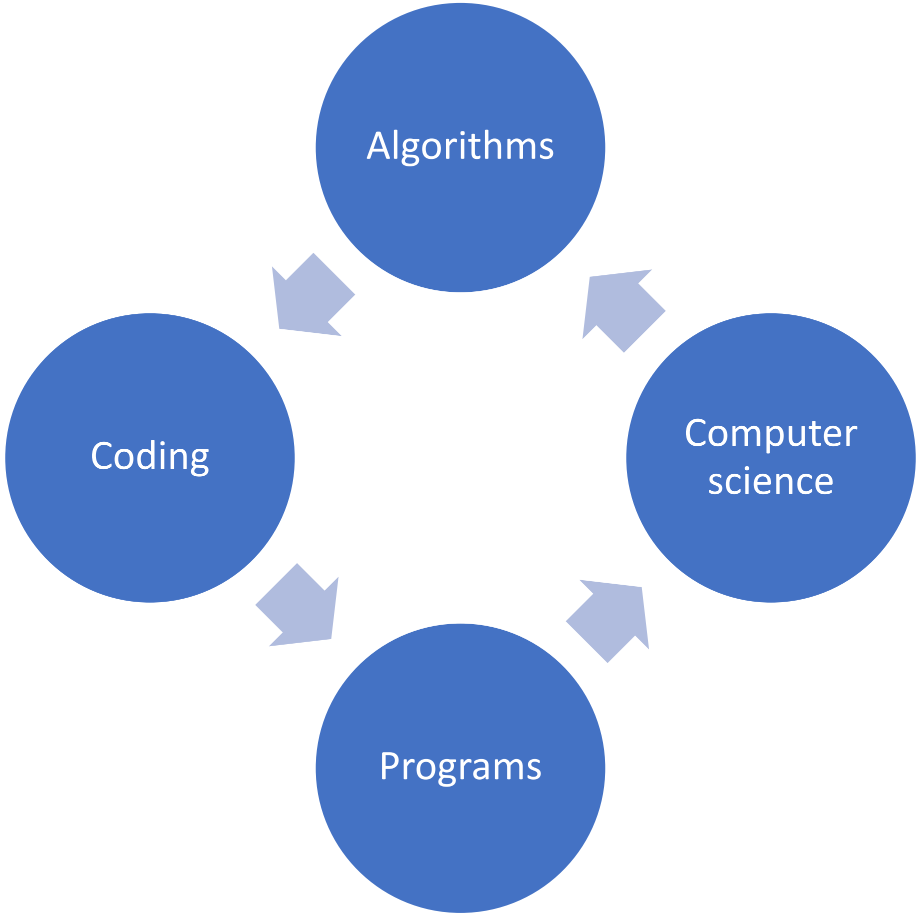
A. Creation / discovery of algorithms

B. Specification of algorithm as a procedure

C. Compilation to native instructions while maximizing performance

D. Execution in target machine with facilities for validation

Underappreciated fact:
Programming languages aid discovery of algorithms



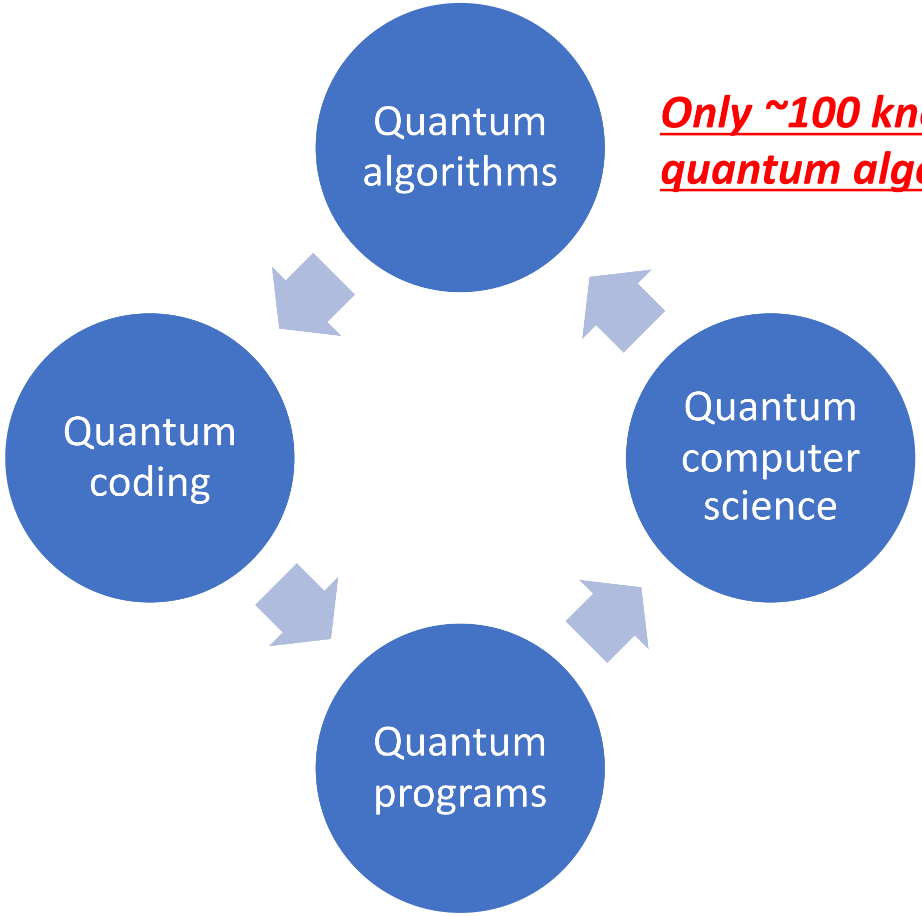
Underappreciated fact:
Programming languages aid discovery of algorithms

https://www.geeksforgeeks.org/random-walk-implementation-python/

Take classical random walks as an example. Notice:

1. Ease of going from 1D example to 2D example (reusable code).
2. Ease of generating a visualization.
3. Code and simulation reveals properties useful for new algorithms.

Programming languages aid discovery of algorithms: Is it currently true for quantum computer science?



Only ~100 known quantum algorithms

I. Quantum programming: from abstractions to execution
   A. Creation / discovery of algorithms
   B. ***Specification of algorithm as a procedure***
   C. Compilation to native instructions while maximizing performance
   D. Execution in target machine with facilities for validation

Specification of algorithm as a procedure

https://www.geeksforgeeks.org/random-walk-implementation-python/

Take classical random walks as an example. Notice:

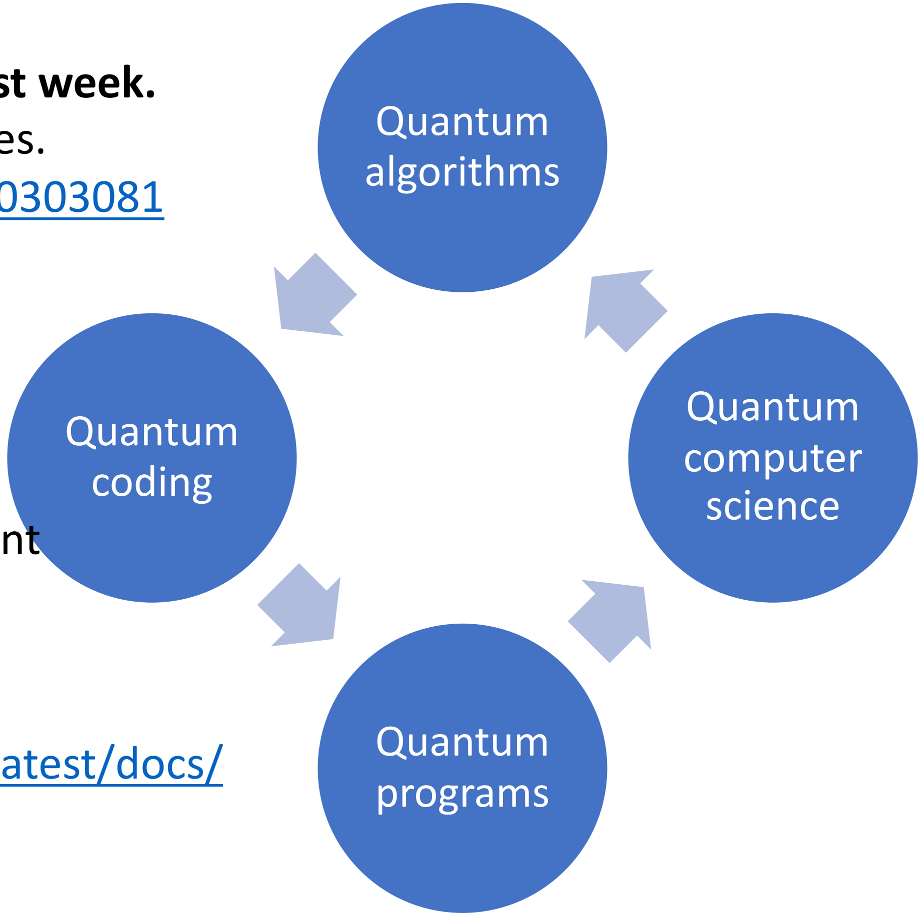
1. Import library for random coin toss
2. Data structures for time series
3. Standard operators for increment and decrement

Specification of quantum algorithm as a quantum procedure

Quantum random walk from last week.
Idea to proof to equation to gates.
https://arxiv.org/abs/quant-ph/0303081

Need quantum equivalent of:
1. Coin toss
2. Position time series
3. Position increment decrement

Gates to code that we can run.
https://cirq.readthedocs.io/en/latest/docs/tutorials/Quantum_Walk.html



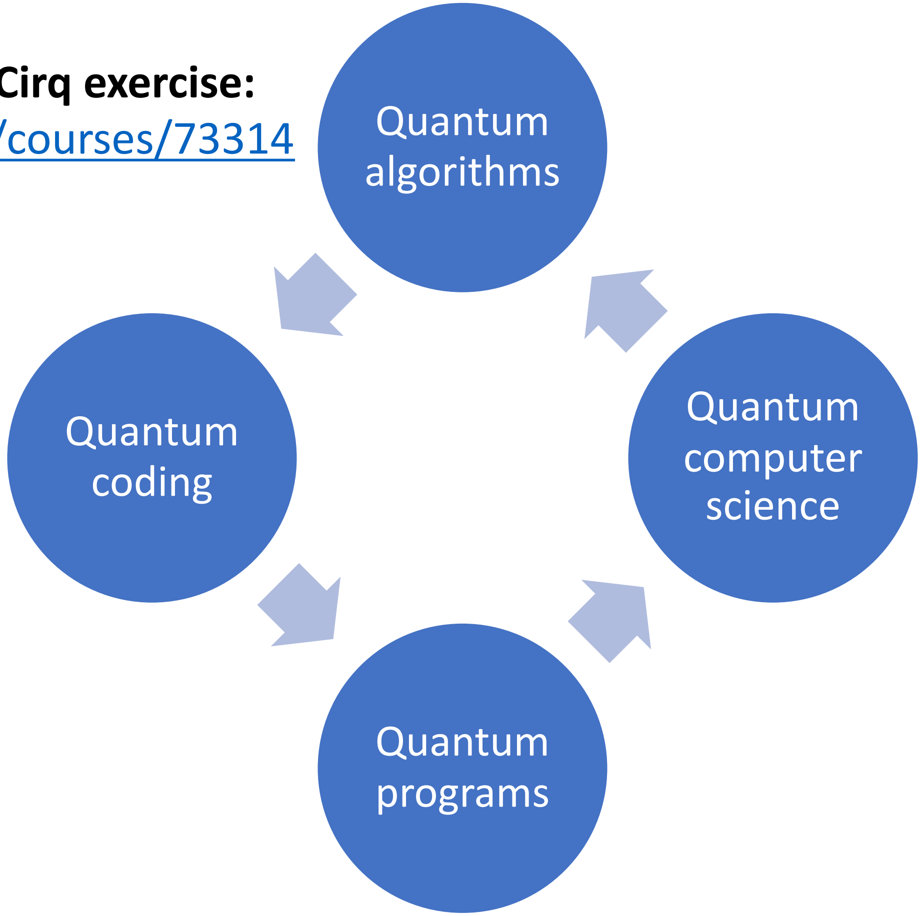
Specification of quantum algorithm as a quantum procedure

As another example, QAOA on Cirq exercise:
https://rutgers.instructure.com/courses/73314/assignments/1017995

Need quantum equivalent of:

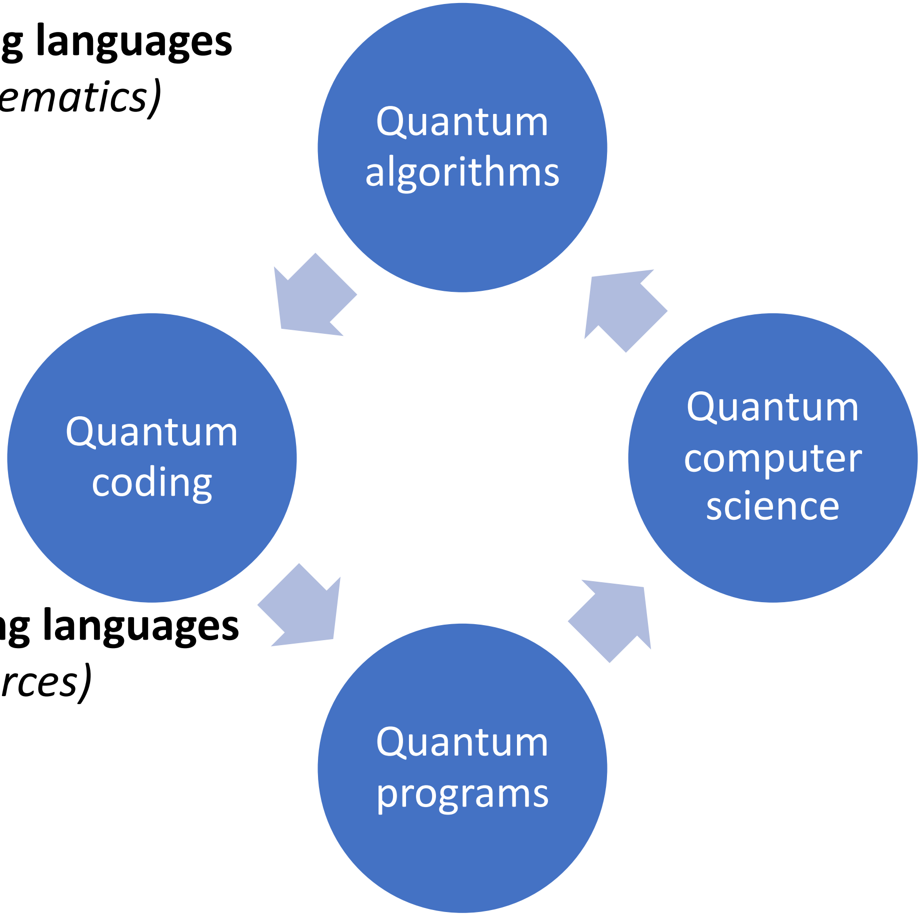
1. Partition representation
2. Edge constraints
3. Way to perturb partitioning

Gates to code that we can run.



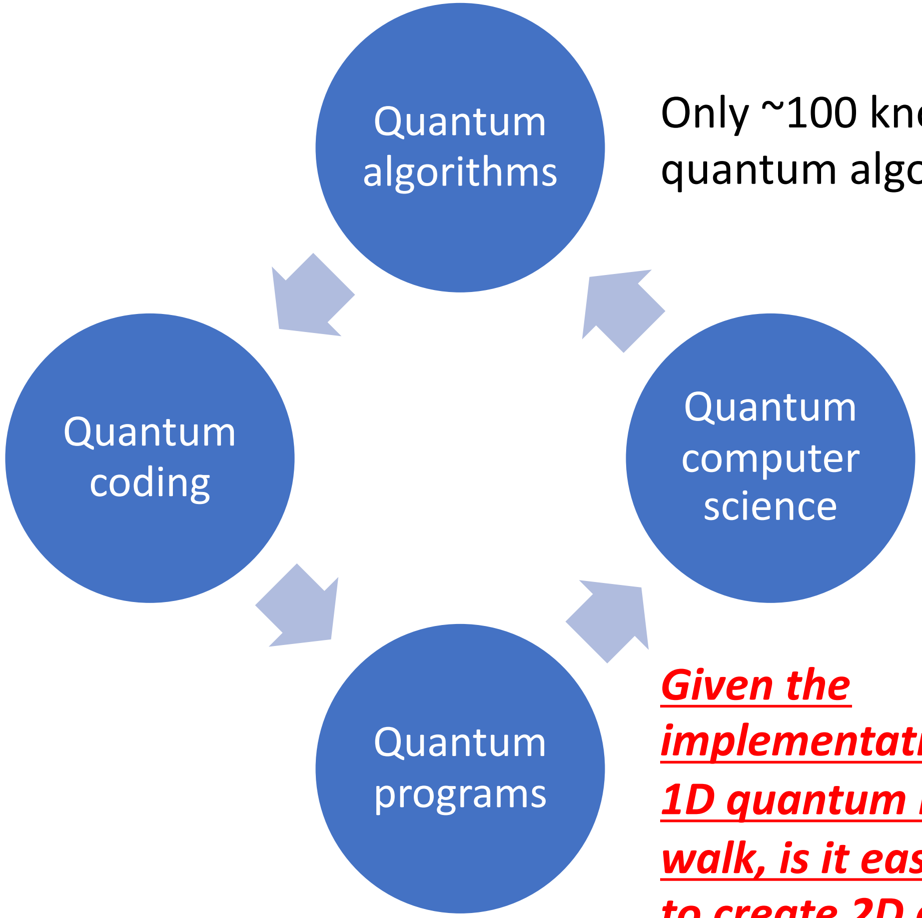
Specification of quantum algorithm as a quantum procedure

Functional quantum programming languages
(emphasis on specifying the mathematics)
Microsoft Liquid
Quipper
QWIRE
Microsoft Q#
Etc.



Imperative quantum programming languages
(emphasis on specifying the resources)
Scaffold
Microsoft ProjectQ
Rigetti PyQuil
IBM Qiskit
Google Cirq
Etc.

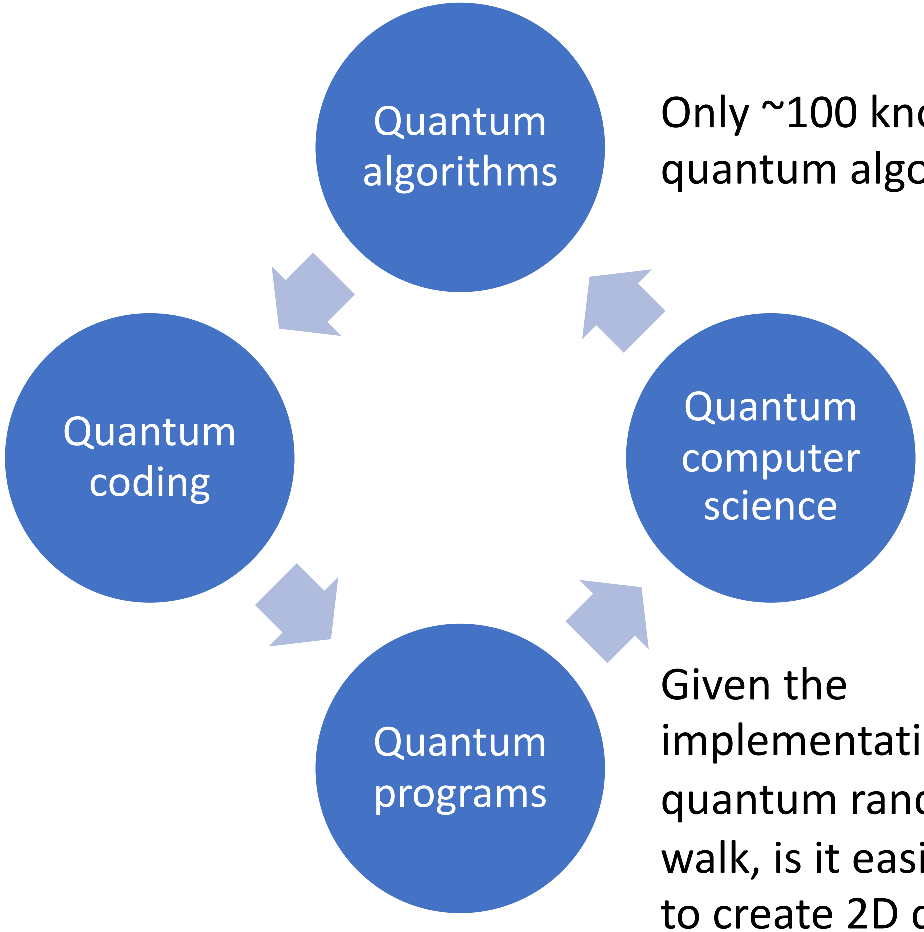
Programming languages aid discovery of algorithms: Is it currently true for quantum computer science?



Only ~100 known quantum algorithms

Given the implementation for 1D quantum random walk, is it easier now to create 2D quantum random walk?

Programming languages aid discovery of algorithms: Is it currently true for quantum computer science?



Only ~100 known quantum algorithms

Some progress in the past two years.
Qiskit Aqua
OpenFermion-Cirq
TensorFlow Quantum

Given the implementation for 1D quantum random walk, is it easier now to create 2D quantum random walk?

II. Quantum programming: from abstractions to execution

A. Creation / discovery of algorithms
B. Specification of algorithm as a procedure
C. ***Compilation to native instructions while maximizing performance***
D. Execution in target machine with facilities for validation

Programmers’ time is scarce.
Classical computing resources are abundant.
Nonetheless, abstractions are expensive.

Table 1. Speedups from performance engineering a program that multiplies two 4096-by-4096 matrices. Each version represents a successive refinement of the original Python code. “Running time” is the running time of the version. “GFLOPS” is the billions of 64-bit floating-point operations per second that the version executes. “Absolute speedup” is time relative to Python, and “relative speedup,” which we show with an additional digit of precision, is time relative to the preceding line. “Fraction of peak” is GFLOPS relative to the computer’s peak 835 GFLOPS. See Methods for more details.

Version	Implementation	Running time (s)	GFLOPS	Absolute speedup	Relative speedup	Fraction of peak (%)
1	Python	25,552.48	0.005	1	—	0.00
2	Java	2,372.68	0.058	11	10.8	0.01
3	C	542.67	0.253	47	4.4	0.03
4	Parallel loops	69.80	1.969	366	7.8	0.24
5	Parallel divide and conquer	3.80	36.180	6,727	18.4	4.33
6	plus vectorization	1.10	124.914	23,224	3.5	14.96
7	plus AVX intrinsics	0.41	337.812	62,806	2.7	40.45

"There’s plenty of room at the Top: What will drive computer performance after Moore’s law?" Leiserson et al. Science. 2020.

Compiling quantum program abstractions to optimal quantum execution

Compilation for maximum correctness, while respecting constraints:

- variable qubit, operation, measurement reliability
- connectivity constraints
- parallelism

Will be topic of two-week chapter on "extracting success."

II. Quantum programming: from abstractions to execution

A. Creation / discovery of algorithms
B. Specification of algorithm as a procedure
C. Compilation to native instructions while maximizing performance
D. ***Execution in target machine with facilities for validation***

Execution in target machine w/ facilities for validation

- Exception handling.
- Printf debugging.
- GDB breakpoints.
- Assertions.
- Etc.