Charles Yuan — Research Statement

I build programming systems that enable developers to program a quantum computer to practically realize quantum algorithms. Quantum algorithms offer asymptotic speedups over classical algorithms by exploiting physical properties of quantum information such as *superposition*, *interference*, and *entanglement*. These unique properties, however, also force us to rethink the foundational programming abstractions – such as data structures and control flow – that we use to express algorithms. My research reveals how familiar principles of programming can be incorrect or inefficient on a quantum computer, jeopardizing the speedup of an algorithm when implemented. In response, I show how we can build new abstractions that meet the needs of quantum algorithms and pave way to realizing them on hardware.

Motivation and Overview

Quantum computers provide asymptotic speedups over classical computers for tasks such as optimization, search, and simulation. Realizing this power in practice would accelerate operations research, disrupt cryptography in wide use, and unlock discovery of new materials. In short, it would revolutionize computing, science, and engineering.

A quantum computer manipulates the states of *qubits*, quantum analogues of classical bits, by executing *quantum logic gates*. A qubit exists in a *superposition* – a weighted sum – of the states `0` and `1`. Quantum algorithms derive speedup from the ability of data in superposition to exhibit *interference*, which amplifies the weights of desirable states and cancels out other ones, and *entanglement*, which enables states to be correlated across long distances. Thanks to billions of dollars of investment worldwide, researchers have made steady progress toward building useful physical qubits using various technologies. In the last two years, Google [8], IBM [9], and Microsoft [10] have each announced hardware breakthroughs that the companies forecast will lead us to a scalable and reliable quantum computer.

Challenge. Equally important to building the quantum computer, however, is programming it — turning abstract algorithms that offer speedup into software that delivers it in practice. To correctly orchestrate the phenomena of interference and entanglement, programmers need new languages and tools – quantum analogues of C and Python. Here, my research reveals the fundamental challenges imposed by the nature of quantum information and hardware that programmers and programming systems must overcome in order to practically realize quantum speedup:

- ***Realizing Abstractions.*** Many quantum algorithms rely on data structures such as sets and maps, as well as control flow such as branching and iteration. Classical programmers may easily implement a set as a hash table or realize control flow via a conditional jump. But on a quantum computer, such conventional implementation approaches would nullify quantum interference and thus eliminate the speedup of a quantum algorithm.
- ***New Costs and Errors.*** Moreover, quantum programs suffer from asymptotic overheads in performance due to *quantum error correction* as required by realistic hardware, as well as correctness bugs caused by entanglement. If not detected and fixed by the programmer, these costs and errors also jeopardize the advantage of the algorithm.

Put briefly, programming a quantum computer like a classical one can turn the quantum computer into a classical one. We lose the computational advantage and the return on the substantial investment of quantum hardware.

Contributions. In response, I apply research in programming languages to design parts of a new software stack for quantum computation – including abstractions, languages, and compilers – that enable us to practically realize quantum algorithms across hardware platforms. Tower [1, OOPSLA 2022 Distinguished Artifact Award] is a language that enables programmers for the first time to correctly implement fundamental data structures, such as sets and maps, on a quantum computer. The quantum control machine [2, OOPSLA 2024] is an instruction set architecture that enables a developer to – for the first time – express control flow in a quantum algorithm via the familiar concept of a program counter rather than hardware-level logic gates while preserving asymptotic advantage.

My research also provides analyses and optimizations to overcome new costs and sources of error in quantum programs. The Spire compiler [3, PLDI 2024] identifies critical, but overlooked, overheads in program performance due to quantum error correction and automatically eliminates this slowdown. The Twist type system [4, POPL 2022] enables developers to detect unintuitive errors in quantum programs that arise due to entanglement.

Impact. Taken together, my contributions hold out a promise of expressive, efficient, and cross-platform abstractions for quantum programming — tools on the critical path to realizing the speedup of quantum algorithms and vindicating the costs of quantum hardware. In placing this nascent field on the map, my work informs computer architects of opportunities to co-design hardware and software, endows software engineers with more powerful languages and compilers, and gives computational scientists tools to solve problems that are intractable on any computer today.

Data Structures in Quantum Superposition

Quantum algorithms for tasks such as element distinctness, subset sum, and closest pair rely on data structures to achieve computational advantage. They rely on, for example, a set abstraction to efficiently maintain a collection of items, test for membership, and add and remove items. In a quantum algorithm, sets exist in superposition and exhibit interference — just as a qubit is a superposition of `0` and `1`, a quantum set can exist as a superposition of $\{1\}$ and $\{2,3\}$ at the same time. To realize the algorithm, the programmer must concretely implement sets in terms of qubits and express each set operation as a program that compiles to a *quantum circuit*, or sequence of logic gates.

Problem. However, conventional approaches to implement data structures produce incorrect results in a quantum algorithm. For example, if a quantum programmer uses a hash table or binary tree to implement an abstract set within the algorithms above, then the algorithm no longer produces quantum interference and thus loses all speedup.

The insight is that to correctly exhibit quantum interference, the implementation of a data structure must satisfy two conditions. First, it must be *reversible*, meaning that each data structure operation cannot erase or overwrite any information. Second, it must be *history-independent*, meaning that each data structure instance has a unique physical bit representation. For example, a list is not a history-independent implementation of a set, because both `[3,2]` and `[2,3,2]` represent $\{2,3\}$. Other standard implementations of data structures, such as hash tables and binary trees, similarly do not satisfy these properties. Thus, we need new programming tools to develop correct implementations.

Approach. I introduce Tower [1], a programming system that enables a developer to implement pointer-based data structures, such as lists, sets, and maps, that exist in superposition. Tower provides a language in which the developer can express data structure operations using high-level abstractions that satisfy reversibility and history independence. Then, they can compile the Tower program to a circuit so as to integrate the data structure into an algorithm.

A key contribution of Tower is a novel quantum analogue of the pointer data type, which the developer uses to manipulate superpositions of data structures by manipulating superpositions of memory addresses. Whereas dereferencing a pointer in a classical program has irreversible semantics that may overwrite or erase data, Tower provides a pointer access operator that is reversible and thus can be compiled correctly to a quantum circuit. Another key contribution is a novel memory allocator that returns a uniform superposition of all possible allocation sites. This strategy enables an appropriately constructed data structure to be history-independent at minimal runtime cost.

Impact. Using Tower, a developer can for the first time implement data structures that meet the correctness and asymptotic efficiency requirements of quantum algorithms. To demonstrate, I contribute the **first quantum library of data structures** – including lists, queues, strings, and sets – which developers can invoke to practically realize quantum algorithms that use these abstractions. This work received a **Distinguished Artifact Award** at OOPSLA 2022, and has inspired follow-up research on the implicit computational complexity of quantum programs [11].

Control Flow in Quantum Superposition

Quantum algorithms for search and simulation rely on the concept of control flow, such as branching and iteration, that depends on the value of data in superposition. Programming constructs for control flow, in the form of `if`-statements and `while`-loops or lambda expressions and continuations, have been ubiquitous in classical languages since Fortran and Lisp. By contrast, quantum languages typically do not provide high-level abstractions for control flow and instead require the programmer to specify hardware-level logic gates to implement algorithms that utilize control flow.

Problem. The reason why abstractions for control flow are scarce in quantum languages is that control flow is not natively supported by the architecture of a quantum computer as understood to date. A classical computer supports control flow via a program counter that points to the next instruction to execute. A compiler then translates high-level abstractions for control flow – such as a loop, exception, or function call – to explicit updates of the program counter. These updates can depend on the data in the program, enabling languages to provide fully general control flow.

By contrast, the predominant architectural model of a quantum computer can execute only a program presented as a quantum circuit — a sequence of logic gates whose structure is fixed in advance and cannot depend on data in the program. Researchers know how to express basic control flow in circuit form, such as an `if`-statement branching on a qubit. But no work to date has defined a sound mechanism for a program counter or any other means to achieve general data-dependent control flow on a quantum computer, limiting the design of programming languages.

Approach. My work [2] reveals why a solution has eluded the community so far — the classical implementation of control flow via a program counter can destroy the advantage of a quantum algorithm, and data-dependent control flow is not realizable on a quantum computer in general. However, I also show that it is realizable under appropriate restrictions. To demonstrate, I present an instruction set architecture named the quantum control machine that gives the first complete characterization of control flow that can be correctly realized on a quantum computer.

The core argument is that we cannot realize general abstractions for control flow on a quantum computer by simply endowing it with a program counter in superposition and lifting the classical conditional jump instruction to superposition. Fundamentally, this instruction has an *irreversible* semantics that overwrites the previous value of the program counter and thus prevents an algorithm from producing quantum interference. Moreover, general solutions to this problem in classical reversible computing fail in the quantum setting due to the effects of entanglement.

Building on this insight, I formalize *reversibility* and *synchronization*, the necessary and sufficient conditions for control flow to be correctly realizable on a quantum computer such that a program can produce speedup. In turn, the quantum control machine presents a set of jump instructions whose behavior is restricted to satisfy these conditions.

Impact. My impossibility result organizes our understanding of the field. It formally proves that we cannot correctly realize the classical foundations of control flow – including analogues of unbounded `while`-loops and λ-calculus – on a quantum computer. Put directly, **quantum computers cannot follow the footsteps of classical stored-program computers** whose architectures host the traditional paradigms of imperative and functional programming.

Simultaneously, the quantum control machine points to a way forward by enabling programmers for the first time to express **control flow without hardware-level logic gates** in quantum algorithms for factoring and simulation. This design has catalyzed follow-up research on realizing quantum algorithms that use recursive procedure calls [12], and is the first step toward more expressive languages and tools to teach, implement, and analyze algorithms.

Abstraction Costs Under Quantum Error Correction

My work above studied an idealized quantum computer that can faithfully perform any quantum computation in principle. A realistic device, by contrast, must contend with physical noise that corrupts the states of qubits — calling for the use of *quantum error correction* to protect the integrity of a large-scale computation in practice. Error correction is critical to correctly execute a quantum program, but in turn imposes a performance tradeoff by restricting the available logic gates the program may utilize. The danger is that this restriction can lead a quantum algorithm to compile to so many gates that its speedup falls short of that predicted by its idealized theoretical analysis.

Problem. My work [3] reveals that the abstractions for control flow that theorists use to specify many quantum algorithms can make a program cost asymptotically more on an error-corrected quantum computer than an idealized one. This overhead is independent of my work on control flow above [2] and can jeopardize quantum advantage.

More precisely, the quantum analogue of an `if`-statement incurs a polynomial increase in *T-complexity* — the number of T logic gates a program compiles to. This T gate is crucial to realize control flow on prevailing error-correcting codes and also a dominant resource bottleneck that hardware architects use to quantify the cost of an error-corrected computation. The issue is that whereas an idealized theoretical analysis of a quantum program directly counts the number of `if`-statements, nested `if`s actually compile to polynomially more T gates than the number of statements in the program. This asymptotic slowdown impacts quantum algorithms for search and optimization.

Prior methods to analyze and reduce the cost of quantum programs do not precisely identify or eliminate this overhead of control flow. Thus, we need new tools to ensure that algorithms give speedup on error-corrected hardware.

Approach. I introduce Spire, a quantum compiler featuring analyses and optimizations that find and eliminate the overhead of control flow under error correction in a quantum program. Spire features a cost model that enables a developer to analyze the T-complexity of a program and pinpoint sources of slowdown at the level of program syntax. Atop this model, Spire presents program-level optimizations that enable a developer to rewrite a program to reduce its T-complexity, predict the cost of the optimized program, and compile it to a quantum circuit in a simple way.

Impact. The Spire compiler **automatically produces asymptotically efficient programs**, meaning their runtime T-complexity under error correction matches their theoretical time complexity on an idealized quantum computer.

Moreover, Spire's approach can yield **better results in less time** than quantum circuit optimizers found in existing work. For a set of 11 benchmark programs that use control flow, Spire and only 2 of 8 tested circuit optimizers produce asymptotically efficient circuits. Compared to them, Spire uses 54×–2400× less compile time. This result holds out the promise of compilers that quickly optimize quantum algorithms to satisfy hardware requirements.

Future Directions

Going forward, my work opens new opportunities in quantum and other emerging paradigms of computation.

Optimizing Algorithms. First, I envision a future where quantum algorithms exhibit end-to-end advantage over classical algorithms while running on near-term, resource-constrained hardware. This goal requires reducing constants and not relying on asymptotics alone. In theory, an asymptotic advantage can speed up the exact solution of a problem from millennia on a classical computer to perhaps a year on an idealized quantum computer. But to compete in practice with the best classical heuristics and approximations for tasks such as physical simulation, the quantum computation must finish in days or hours, even given hardware with limited qubit quantity, reliability, and connectivity.

To accelerate quantum programs and account for limited hardware resources, we must build quantum compilers that are both effective and efficient at scale — at the large problem sizes required for quantum advantage. For a sense of these sizes, quantum algorithms for physical simulation require at least 10^8 logic gates. But prior methods to optimize quantum circuits have typically been evaluated on smaller benchmarks of 10^3 gates, and my results above [3] indicate that these prior methods become slower and less effective at large scales. My research introduces an alternative of building program optimizers rather than circuit optimizers — enabling powerful analyses and optimizations by exploiting higher-level program structure. For example, a compiler can safely invoke more efficient subroutines, such as state preparation via *alias sampling* [13] or arithmetic via *conditionally clean ancillas* [14], that are correct only under specific conditions. I will pursue this approach to build a scalable **quantum compiler infrastructure** and continue my industry collaborations with vendors such as Google [5] and IBM to target their hardware roadmaps.

Building Abstractions. In the longer term, I envision a world where programmers can invoke quantum computation as intuitively as parallelism and randomization. Today, even an algorithmic novice can speed up many programs simply by adding threads or sampling the inputs. But even a quantum expert cannot simply tap into superposition to speed up a generic program, since all known quantum algorithms follow templates – such as *phase estimation* and *amplitude amplification* – that are disparate and specialized. We lack a quantum analogue of `#pragma omp parallel` — a broadly accessible programming abstraction that invokes the essential effect of the computational model.

A promising candidate for this abstraction is the *quantum singular value transform* (QSVT) [15], a mathematical framework shown to unify the templates underlying quantum algorithms [16]. The gap is that expressing an algorithm using the QSVT is a tricky process for a programmer, involving approximating polynomials and gates with continuous parameters. Here, my work on the quantum control machine suggests that we can make the QSVT more programmable by augmenting the framework with more traditional abstractions such as control flow, memory, and arithmetic. This approach can transform the QSVT into a **virtual machine for quantum algorithms** — a basis for new languages and tools that empower computational scientists to design and realize quantum algorithms for untapped applications.

Broadening Domains. Finally, I envision the ability to program other emerging models of computation, such as neural networks and analog devices, that offer time, space, and energy advantages over the computers of today. Like quantum computers, they reject the traditional world of discrete data and embrace more exotic forms of information. They also share many of the challenges of quantum programming – continuous parameters, unreliable hardware, probabilistic behavior, uninterpretable states, non-local effects – that make it difficult to express, compose, and verify programs. These obstacles have hindered the broad adoption of these emerging platforms in practice.

To build **new abstractions, languages, and compilers** that make these models of computation easier to program, I will leverage my experience in quantum and probabilistic [6, 7] programming and establish new collaborations with domain experts. This approach benefits fields ranging from machine learning to architecture and devices by enabling us to use the emerging computer systems they have proposed in an accessible, flexible, and safe way.

Referenced Work

[1] **Charles Yuan** and Michael Carbin. Tower: Data Structures in Quantum Superposition. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2022. DOI: 10.1145/3563297. **Distinguished Artifact Award**.

[2] **Charles Yuan**, Agnes Villanyi, and Michael Carbin. Quantum Control Machine: The Limits of Control Flow in Quantum Programming. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2024. DOI: 10.1145/3649811.

[3] **Charles Yuan** and Michael Carbin. The T-Complexity Costs of Error Correction for Control Flow in Quantum Computation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2024. DOI: 10.1145/3656397.

[4] **Charles Yuan**, Christopher McNally, and Michael Carbin. Twist: Sound Reasoning for Purity and Entanglement in Quantum Programs. In *ACM SIGPLAN Symposium on Principles of Programming Languages*, 2022. DOI: 10.1145/3498691.

[5] Matthew Harrigan, Tanuj Khattar, **Charles Yuan**, Anurudh Peduri, Noureldin Yosri, Fionn Malone, Ryan Babbush, and Nicholas Rubin. Expressing and Analyzing Quantum Algorithms with Qualtran, 2024. DOI: 10.48550/arXiv.2409.04643. arXiv: 2409.04643 [quant-ph].

[6] Eric Atkinson, **Charles Yuan**, Guillaume Baudart, Louis Mandel, and Michael Carbin. Semi-Symbolic Inference for Efficient Streaming Probabilistic Programming. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2022. DOI: 10.1145/3563347.

[7] Eric Atkinson, Guillaume Baudart, Louis Mandel, **Charles Yuan**, and Michael Carbin. Statically Bounded-Memory Delayed Sampling for Probabilistic Streams. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2021. DOI: 10.1145/3485492.

External References

[8] Sophia Chen. Google says it's made a quantum computing breakthrough that reduces errors. *MIT Technology Review*, Sept. 11, 2024.

[9] Kenneth Chang. Quantum Computing Advance Begins New Era, IBM Says. *The New York Times*, June 14, 2023.

[10] Stephen Nellis. Microsoft, Quantinuum claim breakthrough in quantum computing. *Reuters*, Apr. 3, 2024.

[11] Emmanuel Hainry, Romain Péchoux, and Mário Silva. A Programming Language Characterizing Quantum Polynomial Time. In *Foundations of Software Science and Computation Structures*, 2023. DOI: 10.1007/978-3-031-30829-1_8.

[12] Zhicheng Zhang and Mingsheng Ying. Quantum Register Machine: Efficient Implementation of Quantum Recursive Programs, 2024. DOI: 10.48550/arXiv.2408.10054. arXiv: 2408.10054 [quant-ph].

[13] Ryan Babbush, Craig Gidney, Dominic W. Berry, Nathan Wiebe, Jarrod McClean, Alexandru Paler, Austin Fowler, and Hartmut Neven. Encoding Electronic Spectra in Quantum Circuits with Linear T Complexity. *Phys. Rev. X*, 8(4), Oct. 2018. DOI: 10.1103/PhysRevX.8.041015.

[14] Tanuj Khattar and Craig Gidney. Rise of conditionally clean ancillae for optimizing quantum circuits, 2024. DOI: 10.48550/arXiv.2407.17966. arXiv: 2407.17966 [quant-ph].

[15] András Gilyén, Yuan Su, Guang Hao Low, and Nathan Wiebe. Quantum singular value transformation and beyond: exponential improvements for quantum matrix arithmetics. In *ACM SIGACT Symposium on Theory of Computing*, 2019. DOI: 10.1145/3313276.3316366.

[16] John M. Martyn, Zane M. Rossi, Andrew K. Tan, and Isaac L. Chuang. Grand Unification of Quantum Algorithms. *PRX Quantum*, 2(4), Dec. 2021. DOI: 10.1103/prxquantum.2.040203.