Embedding Quantum Program Verification into Dafny

FEIFEI CHENG, Iowa State University, USA
SUSHEN VANGEEPURAM, Iowa State University, USA
HENRY ALLARD, Iowa State University, USA
SEYED MOHAMMAD REZA JAFARI, Iowa State University, USA
ALEX POTANIN, Australian National University, Australia
LIYI LI, Iowa State University, USA

Despite recent development of quantum program verification, it is still in its early stage, where many quantum programs are hard to verify due to their inherent probabilistic nature and parallelism in quantum superposition. We propose Qafny$^{\mathsf{C}}$, a system that compiles quantum program verification into a well-established classical program verifier Dafny, enabling the formal verification of quantum programs. The key insight behind Qafny$^{\mathsf{C}}$ is the separation of quantum program verification from its execution, leveraging the strength of classical verifiers to ensure correctness before compiling certified quantum programs into executable circuits. Using Qafny$^{\mathsf{C}}$, we have successfully verified 37 diverse quantum programs by compiling their verification into Dafny. To the best of our knowledge, this is the most extensive formally verified set of quantum programs.

CCS Concepts: • **Software and its engineering** → **Formal software verification**; • **Theory of computation** → **Automated reasoning**; **Logic and verification**; **Quantum computation theory**.

Additional Key Words and Phrases: Quantum Computation, Automated Program Verification, Dafny

ACM Reference Format:
Feifei Cheng, Sushen Vangeepuram, Henry Allard, Seyed Mohammad Reza Jafari, Alex Potanin, and Liyi Li. 2025. Embedding Quantum Program Verification into Dafny. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 379 (October 2025), 27 pages. https://doi.org/10.1145/3763157

1 Introduction

Advanced quantum algorithms often involve intricate entanglement and interference, and their quantum program development faces the challenges of lacking facilities to assure the program correctness [Gill et al. 2024; Swayne 2023]. Because quantum systems are inherently probabilistic and must obey quantum physics laws, traditional validation techniques based on run-time testing are virtually impossible to develop for large quantum algorithms. *Formal verification* might be the only viable option for certifying programs, typically requiring a great effort. Many frameworks have been proposed to verify quantum algorithms [Chareton et al. 2021; Hietala et al. 2021b; Le et al. 2022; Liu et al. 2019; Ying 2012; Zhou et al. 2021] using interactive theorem provers, such as Isabelle, Rocq, and Why3, by building quantum semantic interpretations and libraries, with some attempts towards proof automation by creating tactics in these theorem provers, but verifying quantum algorithms is still time-consuming and requires great human effort.

Authors' Contact Information: Feifei Cheng, Iowa State University, USA, fch777@iastate.edu; Sushen Vangeepuram, Iowa State University, USA, sushenv@iastate.edu; Henry Allard, Iowa State University, USA, hallard@iastate.edu; Seyed Mohammad Reza Jafari, Iowa State University, USA, rjafari@iastate.edu; Alex Potanin, Australian National University, Australia, alex.Potanin@anu.edu.au; Liyi Li, Iowa State University, USA, liyili2@iastate.edu.

Qafny [Li et al. 2024] develops a proof system for utilizing classical automated verification techniques [Cohen et al. 2009; Hoare 1969; Itzhaky et al. 2021; Leino 2010; Löding et al. 2017; Martí-Oliet and Meseguer 2000; Pek et al. 2014; Qiu et al. 2013; Reynolds 2002; Roşu and Ştefănescu 2011; Roşu et al. 2013; Ta et al. 2016] to verify quantum programs. While Qafny demonstrates the theoretical feasibility, it remains a prototype, requiring additional manual effort to apply to new programs. This work moves beyond theory and introduces the Qafny$^{\text{c}}$ compiler: a robust, end-to-end system that automates this translation process. By systematically compiling high-level quantum programs and their specifications into the Dafny program verifier, our work transforms a manual proof-of-concept into a practical tool for automated verification.

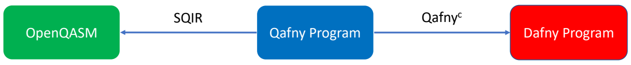

Fig. 1. The Qafny Compiler Pipeline

The Qafny$^{\text{c}}$ design principle is the separation between program verification from execution, as in Figure 1. For execution, Qafny programs are compiled into runnable quantum circuits via SQIR [Hietala et al. 2021a,b] and OpenQASM [Cross et al. 2017]. For verification, they are translated into the Dafny verifier. This separation indicates that our generated Dafny code is never executed, so we are free to encode the quantum states using simple, classical data structures, such as immutable sequences, to model quantum behavior in a way that is optimized for proof, rather than physical simulation. We treat quantum program specifications as logical predicates, and compile quantum program operations into verifiable Dafny functions that manipulate these structures, leveraging the full power of Dafny's SMT-based solver to prove correctness without confronting the complexity of quantum simulation. Such handling benefits two aspects for verifying quantum programs:

- Represent quantum states and operations respectively as classical data structures (e.g., arrays) and transformations on the structures, leveraging existing methods for effective verifications.
- Large upper bounds cannot be a problem. Using a classical array structure to represent a quantum state might contain exponentially many elements, which should not be an issue if we properly encode the program verification in a classical SMT-solver-based program verifier, i.e., whether or not we verify a 2 or 2^n element array does not affect the efficiency.

1.1 A Motivating Example

In Figure 2, f adds a number m to a uniform superposition state ($\varphi = \sum_{j=0}^{2^n-1} \frac{1}{\sqrt{2^n}} |j\rangle$), which is prepared by applying n Hadamard gates to the initial n $|0\rangle$ states. Assuming that the n qubits are named as x, we can describe the qubit array as $x[0, n)$, a range fragment $[0, n)$ in the qubit array x. The verification of applying the function f to the superposition state can be described in Qafny as the following triple.

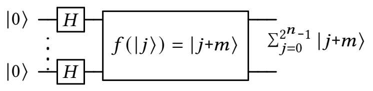
$|0\rangle$
H
$|0\rangle$
H
$f(|j\rangle) = |j{+}m\rangle$
$\sum_{j=0}^{2^n-1} |j{+}m\rangle$

Fig. 2. n qubit addition circuit. $|j{+}m\rangle$ means $(j{+}m)\%2^n$.

$$\left\{ x[0, n) \mapsto \sum_{j=0}^{2^n-1} \frac{1}{\sqrt{2^n}} |j\rangle \right\} x[0, n) \leftarrow \lambda x. f(|x\rangle) \left\{ x[0, n) \mapsto \sum_{j=0}^{2^n-1} \frac{1}{\sqrt{2^n}} |j{+}m\rangle \right\}$$

A key component of the Qafny$^{\text{c}}$ design is to encode quantum states as array-like structures and quantum operations as aggregate array (high-order map) operations over the states. In Figure 2, the φ state after the Hadamard operations can be viewed as an array containing 2^n elements, one for each indexed basis element in φ. Each element is a pair of real and natural numbers (computational basis vector, essentially a bitstring), e.g., $\frac{1}{\sqrt{2^n}}|j\rangle$. The operation $(\lambda x. f(|x\rangle))$ applies f to each basis vector $|j\rangle$ in an element (basis-ket) of the array. The Qafny$^{\text{c}}$ compiler embeds the Qafny triple to the Dafny program adder in Figure 3, by compiling the program procedure with the generation of pre-

```
1  method adder(amp : seq<real>, x : seq<seq<bv1>>, n : nat, m : nat)
2    returns (amp_f : seq<real> , x_f: seq<seq<bv1>>)
3    requires |amp| == |x| == pow2(n)                   //pow2(n) is 2^n.
4    requires ∀j : : 0 ≤ j < |amp| ⟹ amp[j] == 1.0 / sqrt(pow2(n))
5    requires ∀j : : 0 ≤ j < |x| ⟹ b2n(x[j], n) == j //b2n converts bitstrings to nat.
6    ensures |amp_f|== |x_f|== pow2(n)
7    ensures ∀j : : 0 ≤ j < |amp_f| ⟹ amp_f[j] == 1.0 / sqrt(pow2(n))
8    ensures ∀j : : 0 ≤ j < |x_f|⟹  b2n(x_f[j], n) == (j + m) % pow2(n)
9  {
10   amp_f := amp;
11   var i := 0;
12   x_f = []
13   while (i < pow2(n))
14     invariant 0 ≤ i ≤ pow2(n)
15     invariant i == |x_f|
16     invariant ∀j : : 0 ≤ j < i ⟹ b2n(x_f[j], n) == (j + m) % pow2(n)
17   {
18     x_f := x_f + [f(x[i], m, n)]; //the f function adds m to n-bit bitstring x[i].
19     i := i + 1;
20   }
21 }
```

Fig. 3. The Generated Dafny code for the adder example

and post-conditions as well as invariant constraints. The superposition state in the precondition ($\sum_{j=0}^{2^n-1} \frac{1}{\sqrt{2^n}} |j\rangle$), defined over the range $x[0, n)$, is unzipped as two Dafny sequences (type seq), amp (type: seq<real>) and x (type: seq<seq<bv1>>, where bv1 is length-1 bit vector). We utilize the requires in lines 3-5 to describe the precondition. Line 3 equates the lengths of two sequences amp and x with 2^n, as there are 2^n sum-terms. Line 4 constrains that an element in amp is $\frac{1}{\sqrt{2^n}}$, while line 5 sugggests a basis-vector value in x is $|j\rangle$.

We automatically generate variables amp_f and x_f to represent the post-state $\sum_{j=0}^{2^n-1} \frac{1}{\sqrt{2^n}} |j+m\rangle$ for the same range, and generate three ensures: line 6 ensures that they have the same length as in the pre-state (x and amp). The f application does not modify amp, so the program body assigns it to amp_f and results in the post-condition (ensures) in line 7. Finally, we utilize a while loop to apply the f function to each element in the x sequence and concatenate the result to x_f, resulting in the post-condition in line 8. Note that the invariants in the while loop are automatically generated.

The compiled Dafny code is intended for verification purposes. Dafny verifies the aforementioned program by embedding the program verification into an SMT solver via a classical Hoare-logic-based verification procedure, thereby avoiding costly program executions. In the above case, the 2^n (pow2(n)) bound is not a bottleneck for the verification of the while loop because the Dafny verification compilation does not depend on loop bounds but on the program structure complexity. Embedding the quantum program verification into Dafny is effective as long as program structures are simple, which is handled by the Qafny$^{\text{c}}$ compiler via the generation of layouts of function calls, e.g., the f function could contain another layout of while loop structure, and we wrap the verification for the additional layout in a separate function. On the other hand, it can be time-consuming and probably error-prone for programmers to directly verify quantum programs in Dafny, as many additional constraints require manual inputs, e.g., the one-line Qafny triple above is compiled to a 21-line Dafny program above with insertions of additional constraints and program operations such as while loops. The Qafny$^{\text{c}}$ compiler automatically finishes the task for programmers.

1.2 Contributions, Roadmap, and Comparison to Qafny

Our main contribution is the design, implementation, and evaluation of Qafny^c, a compiler-based framework that automates practical quantum program verification. The primary contributions are:

- *An Automated Verification Compiler.* We provide the first end-to-end compiler that automates the translation of quantum program verifications into a classical verifier, turning a manual proof process into a push-button tool.
- *A Type-Directed Verification Strategy.* We introduced a type system that syntactically guides compilation, enabling scalable verification that reasons about the program's structure without exploring an exponential quantum state space.
- *An Extensive Evaluation of Quantum Programs.* We demonstrated the power of our approach by verifying the largest and most diverse set of quantum programs to date, establishing the practical viability of automated verification at scale.
- *Practical Verification for Developers.* We make formal methods a useful tool for quantum programmers by automatically translating low-level verifier output into clear, high-level feedback. This helps developers identify and fix bugs within a typical iterative workflow.

Qafny^c versus Qafny. Previously, Qafny [Li et al. 2024] introduced a quantum proof system, framing quantum program verification within the established paradigm of classical program verification, based on separation logic. It included a prototype compiler to demonstrate the feasibility of this approach; however, its automation was partial and unsystematic. The generated Dafny code often required significant manual post-processing, such as rewriting loop invariants or adding assertions, to become verifiable. Consequently, extending its application to new quantum programs required considerable human effort and expert knowledge, limiting the practical utility for diverse quantum algorithms. In particular, handling mutable data structures like arrays for quantum states in Dafny often demanded explicit manual proof steps to ensure correctness after updates.

Building upon these insights, we developed Qafny^c, a compiler that systematically and automatically verifies quantum programs, shifting the foundation from separation logic to Hoare logic. This design enables Qafny^c to handle array operations by utilizing Dafny's immutable sequences, where updates result in new sequence variables, thereby streamlining reasoning about quantum state transformations. Furthermore, Qafny^c extends the original type system with additional constructs, such as numbering flags in `EN` to facilitate compilation, and introduces a new quantum state type `AA` to reduce the verification burden in Dafny while faithfully representing the structure of quantum algorithms. Qafny^c also enhances flexibility in specifying program constraints through `requires` and `ensures` clauses, permitting quantum states to be described via variable terms.

These developments allow us to verify significantly more programs, increasing from 5 in Qafny, which relied on human intervention, to 37 quantum programs in Qafny^c with full automation. An experimental comparison between Qafny and Qafny^c is provided in Section 5 to demonstrate the different quantum program proofs that Qafny^c can handle, while Qafny is incapable of handling.

Roadmap. Section 2 provides the necessary preliminaries, covering the fundamentals of quantum computation and introducing Qafny^c's typed state predicates. Section 3 then describes the Qafny^c compiler pipeline in detail. Section 4 provides a focused case study on amplitude amplification algorithms. Finally, Section 5 presents a detailed evaluation of our framework.

2 Foundations of Qafny^c: Language and Verification Model

Here, we show the technical groundwork, by first reviewing the quantum computation preliminaries, and then introducting our typed state predicates used in our program pre- and post-conditions with a step-by-step controlled-addition proof for their use in reasoning about state transformations.

```
$\{x[0] : \mathsf{NOR} \mapsto |0\rangle * y[0,n) : \mathsf{NOR} \mapsto |\overline{0}\rangle\}$
  $x[0] \leftarrow \mathsf{H}$;
$\{x[0] : \mathsf{HAD} \mapsto \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) * y[0,n) : \mathsf{NOR} \mapsto |\overline{0}\rangle\}$
$\equiv \{x[0] \uplus y[0,n) : \mathsf{EN}(1) \mapsto \sum_{j=0}^{1} \frac{1}{\sqrt{2}} |j\rangle |\overline{0}\rangle\}$
  if $(x[0])$ $\{y[0,n) \leftarrow y[0,n) + m\}$;
$\{x[0] \uplus y[0,n) : \mathsf{AA}(1) \mapsto \alpha(x[0] = |0\rangle, \frac{1}{\sqrt{2}}) |\overline{0}\rangle + \alpha(x[0] = |1\rangle, \frac{1}{\sqrt{2}}) |m\rangle\}$
$\equiv \{x[0] \uplus y[0,n) : \mathsf{EN}(1) \mapsto \sum_{j=0}^{1} \frac{1}{\sqrt{2}} |j\rangle |j \cdot m\rangle\}$
```

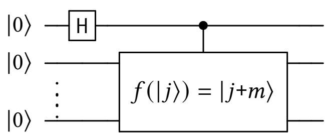

Fig. 4. Controlled-addition program verification (left) and circuit (right); result: $\frac{1}{\sqrt{2}}(|0\rangle |\overline{0}\rangle + |1\rangle |m\rangle)$.

2.1 Quantum Data and Quantum Computation

A quantum datum (often called a quantum state) consists of one or more qubits. A single qubit can be represented as a two-dimensional vector $\binom{z_1}{z_2}$, where z_1 and z_2 are complex amplitudes with $|z_1|^2 + |z_2|^2 = 1$. Using Dirac notation, this is written as $z_1 |0\rangle + z_2 |1\rangle$, with $|0\rangle$ and $|1\rangle$ as the computational basis-vectors ($z_1 |0\rangle$ is a basis-ket). When both z_1 and z_2 are non-zero, the qubit is in a superposition of $|0\rangle$ and $|1\rangle$. Multi-qubit data is constructed via the tensor product, e.g., $|0\rangle \otimes |1\rangle = |01\rangle$. However, not all multi-qubit states can be separated into tensor products; some are entangled states, such as the Bell pair $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$.

Quantum computation applies unitary gates (e.g., Hadamard, controlled-not) to evolve the state of qubits. A gate is represented by a unitary matrix U that acts on the qubit state vector $|\varphi\rangle$, producing a new state $U|\varphi\rangle$. For example, in Figure 4, the Hadamard gate (of the circuit) $\mathsf{H} = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$ transforms computational basis states into superpositions: applying H to $|0\rangle$ yields $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$, and applying H to $|1\rangle$ yields $\frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$. Applying a quantum operation to a superposition state results in a linear sum of applying the operation to each basis-ket state. These operations are composed in quantum circuits as in Figure 2 to implement computations, where each wire denotes a qubit and each gate denotes a transformation applied at a specific time.

Measurement collapses the quantum state to a classical outcome with a probability determined by the amplitudes. For instance, measuring $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ yields 0 or 1, each with probability $\frac{1}{2}$. After measurement, the state irreversibly collapses to the observed basis state.

2.2 Quantum State Predicates and Qafny Triples

A program verification in Qafny$^{\mathsf{C}}$ can be expressed as a separation logic triple $\{\varphi\}\ e\ \{\varphi'\}$, where e is a Qafny program statement in Figure 8, and φ and φ' are typed quantum state predicates in Figure 5, representing pre- and post-conditions, respectively. Qafny verifies such a triple by deriving a proof tree in Li et al. [2024]. On the other hand, Qafny$^{\mathsf{C}}$ embeds the triple as Dafny programs and assertions for verification. We explain below how Qafny$^{\mathsf{C}}$ state predicates are organized, and Section 3 discusses the program syntax. Figure 2 provides a running example performing controlled addition (circuit on the right), controlling on a qubit $x[0]$ and applying f to an qubit array $y[0,n)$, after preparing a superposition qubit $x[0]$ via a Hadamard operation. States in Qafny triples are syntactic entities, and we design the state syntax based on the compiled Dafny state representations. A state φ describes program variable behavior, where variables are partitioned as two different *kinds*: C for classical scalar values, such as natural numbers or reals, and $\mathsf{Q}(n)$ for a physical n-length qubit array conceptually residing in a quantum heap. We permit users to write scalar types, e.g., `nat` and `real`, in a Qafny program, recognized as a C-kind. Qafny$^{\mathsf{C}}$ handles scalar values simply as same-typed Dafny values, as we focus on the quantum data handling below.

In Qafny$^{\mathsf{C}}$, we abstract a quantum state as a heap structure with qubits representing physical heap locations. A state is expressed in the form of $\overline{\kappa : \tau \mapsto q}$ ($\overline{R}$ denotes a multiset of type R),

Basic Terms:

Nat. Num	$m, n \in \mathbb{N}$	Complex Amplitude	$z \in \mathbb{C}$	Real	$p, r \in \mathbb{R}$	Phase	$\omega(r) ::= e^{2\pi i r}$
Variable	x, y	Function Variable	f	Bit/Bool	$d ::= 0 \mid 1$	Bitstring	$c \in d^{+}$

Modes, Kinds, Types, and Classical/Quantum Values:

Classical Scalar Value	v	$::=$	n				
Kind	η	$::=$	$\mathtt{C}$	$\mid\ \mathtt{Q}(n)$			
Basis Vector	β	$::=$	$(\lvert c\rangle)^{*}$				
Basic Ket	w	$::=$	$z\beta$				
EN Typed Data	ρ	$::=$	$\sum_{j_0=0}^{m_0} \dots \sum_{j_{n-1}=0}^{m_{n-1}} w_{j_0,\dots,j_{n-1}}$				
Quantum Type	τ	$::=$	$\mathtt{HAD}$	$\mid\ \mathtt{NOR}$	$\mid\ \mathtt{EN}(n)$	$\mid\ \mathtt{AA}(n)$	
Quantum Data	q	$::=$	$\frac{z}{\sqrt{2^n}} \bigotimes_{j=0}^{n-1} (\lvert 0\rangle + \omega(r_j)\lvert 1\rangle)$	$\mid\ w$	$\mid\ \rho$	$\mid\ \rho\alpha(b,z) + \rho\alpha(b,z)$	

Quantum Loci and States

Qubit Array Range	s	$::=$	$x[n,m)$		
Locus	κ	$::=$	$\overline{s}$	concatenated op	$\uplus$
Typed Quantum State (Heap)	φ	$::=$	$\overline{\kappa : \tau \mapsto q}$	concatenated op	$*$

Syntax Abbreviations and Basis/Locus Equations

$\Sigma_{j=0}^{0} w_j \simeq w_0 \quad 1\beta \simeq \beta \quad \lvert c_1\rangle \lvert c_2\rangle \simeq \lvert c_1 c_2\rangle \quad x[n,n) \simeq \emptyset \quad \emptyset \uplus \kappa \simeq \kappa \quad x[n,m) \simeq x[n,j) \uplus x[j,m) \ \mathtt{if}\ n \le j \le m$

Fig. 5. Qafny state syntax. A range $x[n,m)$ in a locus is the number range $[n,m)$ in a qubit array x. Loci are finite lists; states are finite sets. The ops after "concatenated op" are concatenations.

referring to that a group of *possibly entangled* qubits κ is now mapped to a quantum datum q having the type τ. Elements in a quantum state are separated via the separation conjunction $*$, used to enable modular reasoning about the quantum state, meaning that different qubit groups are disjoint without entanglement. Before we explain the types and loci, we show an example of expressing a Qafny triple from Figure 4 (left). Line 1 is the pre-state predicate before any operation is executed, with NOR typed data $\lvert 0\rangle$ and $\lvert\overline{0}\rangle$ for the two separable qubit groups $x[0]$ and $y[0,n)$. Applying the Hadmard operation to $x[0]$ in line 2 turns $x[0]$ to a HAD typed datum in line 3. The pre- and post-conditions and the operation form a Qafny triple as follows.

$$\{x[0] : \mathtt{NOR} \mapsto \lvert 0\rangle * y[0,n) : \mathtt{NOR} \mapsto \lvert\overline{0}\rangle\}\ x[0] \leftarrow \mathtt{H}\ \{x[0] : \mathtt{HAD} \mapsto \tfrac{1}{\sqrt{2}}(\lvert 0\rangle + \lvert 1\rangle) * y[0,n) : \mathtt{NOR} \mapsto \lvert\overline{0}\rangle\}$$

Qafny$^{\mathsf{c}}$ utilizes types to classify quantum data, allowing quantum program verification managed in different views of a quantum state with different Dafny compilation strategies, detailed in Section 3. A NOR typed datum (w in Figure 5) has a singleton basis-ket, representing a non-superposition state, while qubits in HAD typed data ($\frac{z}{\sqrt{2^n}} \bigotimes_{j=0}^{n-1} (\lvert 0\rangle + \omega(r_j)\lvert 1\rangle)$) are superposition but separable from each other, with $\omega(r_j)$ being a local phase for j-th qubit, e.g., $x[0]$ is a single qubit superposition after the Hadamard application above. The two types provide a qubit view of a quantum state, so program verification is established upon individual qubits.

As mentioned above, many entangled quantum data are not expressible as qubits, where we provide a view of basis-kets, i.e., the quantum datum can be viewed as a linear sum (array) of basis-kets (ρ), each representing a possible outcome for measuring the qubits. For example, we merge the two qubit groups in line 3 into the entangled datum ($\mathtt{EN}(1)$) in line 4. Here, the whole datum has two possible basis-ket states, one is all 0 and the other starts with 1 followed by 0.

In the quantum heap view, physical qubit locations might be hard to track, e.g., the entangled qubit information in line 4 is spread into different basis-kets. Therefore, we develop a *locus* structure $x[0] \uplus y[0,n)$ to indicate that the two *ranges* $x[0]$ and $y[0,n)$ are entangled. It is also an information location indicator, i.e., qubit $x[0]$ points to the first bit in each basis-ket and $y[0,n)$ points to the following bits. Qafny$^{\mathsf{c}}$ enables Dafny axioms to automatically convert the NOR and HAD typed data in line 3 to the EN-typed datum, derived by equivlanece relation $\equiv$; discussed shortly below. The flag 1 in $\mathtt{EN}(1)$ indicates a special structure in EN-typed data, discussed in Section 3.

Many quantum algorithms (Section 4) are described based on probabilistic properties of a group of basis-kets, i.e., for a quantum datum, they partition all basis-kets into different groups and represent the group probabilities by a geometric sum of all basis-ket amplitudes within a group. Qafny^{c} extends the EN entanglement type to $\mathtt{AA}(n)\ (\rho\alpha(b,z) + \rho\alpha(b,z))$ to capture this representation. Essentially, AA typed state is a partition on basis-kets, separated by +, where a construct $\alpha(b,z)$ is used to indicate the condition b and relative amplitude ratio z for the partition. The controlled addition operation, a.k.a. quantum conditional with addition, in line 5 partitions basis-kets into two groups, separated by the bits pointed to by $x[0]$. The addition operation is applied to the group $(x[0] = |1\rangle)$, not the group $(x[0] = |0\rangle)$, resulting in the line 6 state. In the end, we reduce the $\mathtt{AA}(1)$-typed state back to the EN typed one in line 7.

2.3 Compiling Qafny Program Verification to Dafny

Qafny^{c} automates program verification by embedding Qafny triples into Dafny programs. A key Qafny design component is to encode quantum data as immutable array structures (a seq in Dafny) and quantum operations as aggregate array (high-order map) operations over the states, where different seq data structures are used to represent different quantum data types. The aggregate operations are implemented as loop programs, and the Dafny program verification utilizes the operation implementations to derive the quantum postconditions from the preconditions.

An example is to transform the Qafny triple above for the Hadamard operation (Figure 4 lines 1 - 3), modeled as a Dafny method applyHadNor. Given a NOR typed datum, modeled as a seq of bit-vectors (bv1), applyHadNor transitions it to a HAD typed datum, modeled as a seq of real number r; each element represents $\frac{1}{\sqrt{2}}(|0\rangle + \omega(r)\,|1\rangle)$ $(\omega(r) = e^{2\pi i r})$. The Qafny triple is transformed as the following assertions surrounding the method applyHadNor.

```
1 assert ∀k : : 0 ≤ k < 1 ⟹ x[k] = 0;
2 assert ∀k : : 0 ≤ k < n ⟹ y[k] = 0;
3 x_h := applyHadNor(x, 1);
4 assert ∀k : : 0 ≤ k < 1 ⟹ x_h[k] = 0.0;
5 assert ∀k : : 0 ≤ k < n ⟹ y[k] = 0;
```

The precondition is transformed into the two assertions in lines 1 and 2, as two bv1 sequences, x and y. Line 4 shows the postcondition of applyHadNor, where we turn x to be a real sequence x_h, each element being 0.0. Line 5 is unnecessary in the compiled code since y is not modified. Qafny proofs might require quantum state form changes, such as lines 4 and 7 in Figure 4, .e.g, in Figure 4 line 4, we merge the HAD-typed (x_h) and NOR-typed (y) data to the EN(1)-typed datum. Qafny^{c} uses a pre-defined type conversion library function (Figure 11), to convert the state as follows.

```
1 ∀ j : : 0 ≤ j < 2 ⟹ amp_c[j] == 1.0 / sqrt(2.0)
2 ∀ j : : 0 ≤ j < 2 ⟹ b2n(x_c[j], 1) == j
3 ∀ j : : 0 ≤ j < 2 ⟹ b2n(y_c[j], n) == 0
```

To represent a the EN(1)-typed datum (stored in locus $x[0] \uplus y[0,n)$), we use three Dafny sequences, a seq<real> typed sequence (amp_c) and two seq<seq<bv1>> typed sequences (x_c and y_c), to store the two basis-ket information – amp_c stores the amplitudes and x_c and y_c store the basis-vector fragments for $x[0]$ and $y[0,n)$. For $\sum_{j=0}^{1} \frac{1}{\sqrt{2}}\,|j\rangle\,|\overline{0}\rangle$, amp_c[j] represents the amplitude $\frac{1}{\sqrt{2}}$, and x_c[j] and y_c[j] represent the basis-vectors $|j\rangle$ and $|\overline{0}\rangle$.

We show the compilation of another operation in Figure 4 line 5, demonstrating two key Qafny^{c} compiler features: 1) we deal with different components in a quantum state separately, and 2) the quantum conditional compilation is compositional, building upon the subterm compilation. Given the compiled EN(1)-typed datum sequences (amp_c, x_c, and y_c), the compilation is only related to

```
1  method IfAdder(x_c: seq<seq<bv1>>, t: nat, k: nat, y_c: seq<seq<bv1>>, n: nat, m: nat)
2    returns (y_f:  seq<seq<bv1>>)
3    requires |x_c| == |y_c|
4    requires ∀j : : 0 ≤ j < |x_c| ⟹ |x_c[j]| == k
5    requires ∀j : : 0 ≤ j < |y_c| ⟹ |y_c[j]| == n
6    requires ∀j : : 0 ≤ j < |x_c| ⟹ b2n(x_c[j], k) == j
7    requires t < k
8    ensures |y_f|== |x_c| == |y_c|
9    ensures ∀j : : 0 ≤ j < |y_f| ⟹ |y_f[j]| == n
10   ensures ∀j : : 0 ≤ j < |y_f| ⟹ if x_c[j][t] == 1 then b2n(y_f[j], n) ==
11                b2n(y_c[j], n) + m % pow2(n) else b2n(y_f[j], n) == b2n(y_c[j], n)
12 {
13   var i := 0;
14   y_f = []
15   while (i < |x_c|)
16     invariant 0 ≤ i ≤ |x_c|
17     invariant i == |y_f|
18     invariant ∀j : : 0 ≤ j < |x_c| ⟹ |x_c[j]| == k
19     invariant ∀j : : 0 ≤ j < i ⟹ |y_f[j]| == n
20     invariant ∀j : : 0 ≤ j < i ⟹ if x_c[j][t] == 1 then b2n(y_f[j],n) ==
21                b2n(y_c[j], n) + m % pow2(n) else b2n(y_f[j], n) == b2n(y_c[j], n)
22   {
23     if (x_c[i][t] == 1)
24       { y_f := y_f + [add(y_c[i], m, n)]; } //adds m to n-bit bitstring y_c[i].
25     else
26       { y_f := y_f + [y_c[i]]; }
27     i := i + 1;
28   }
29 }
```

Fig. 6. Dafny program and proof for quantum conditional of addition; variables are locally scoped.

the basis-vectors and output amplitudes have no change (amp_c). We then compile the operation related only to x_c and y_c, as IfAdder in Figure 6, which builds upon the compilation of an addition operation, having a similar compiled structure as shown in Figure 3. The IfAdder method is more general than the Qafny code in Figure 4. It takes in a $\mathsf{EN}(1)$-typed locus $x[0,k) \uplus y[0,n)$ and controls on the t-th qubit of the $x[0,k)$'s range to add m to $y[0,n)$. Lines 3 - 5 arrange the constraints for the $\mathsf{EN}(1)$-typed datum, where line 3 equates the basis-ket length in the two ranges $x[0,k)$ and $y[0,n)$, and lines 4 - 5 enforce the qubits in the two ranges to be k and n. In the process, we generate a new sequence y_f as an update to y_c, while x_c is not modified. Lines 8 - 9 ensure that the generated y_f sequence has the same basis-ket number as y_c, and some qubit number in each basis-ket.

The main body mainly builds on top of the addition structure in Figure 3, with the insertion of a classical conditional structure (**if** x_c[j][t] == 1 **then** ... **else** ...) inside the loop, representing checking the controlled qubit (x_c[j][t]) to decide whether we apply the addition or not. Correspondingly, we also modify the generated invariant in line 20 and ensures in line 10.

To verify the pre- and post-conditions of Figure 4 line 5, we instantiate IfAdder with t = 0 and k=1, as well as the x_c and y_c sequences generated in the casting step in Figure 4 line 4. Such an instantiation results in an $\mathsf{AA}(1)$-typed datum, in Figure 4 line 7, with two basis-kets, i.e., y_f has two elements, having values 0 and m. Such a datum is represented by the predicate in line 10 in Figure 6 with a classical conditional structure. In Qafny$^{\mathsf{C}}$, we enable library functions to cast a

AA(m) typed state predicate to a EN(m) typed one, for some simple structures. The predicate in line 10 is one example simple structure that can be cast to a predicate describing a EN(1) typed quantum datum. The casting result in Figure 6 line 7 is listed as follows:

```
    ensures ∀j : : 0 ≤ j < |y_f| ⟹ b2n(y_f[j], n) == (x_c[j][t] as nat * m) % pow2(n)
```

3 The Qafny Compilation Pipeline

This section details the Qafny^c compilation pipeline that translates Qafny program into verifiable Dafny code. We first present the Qafny language syntax in Figure 8. The compilation process itself is organized into several stages, as illustrated in Figure 7. We use the Hamming Weight state preparation algorithm from Figure 9 as a running example throughout the subsequent subsections.

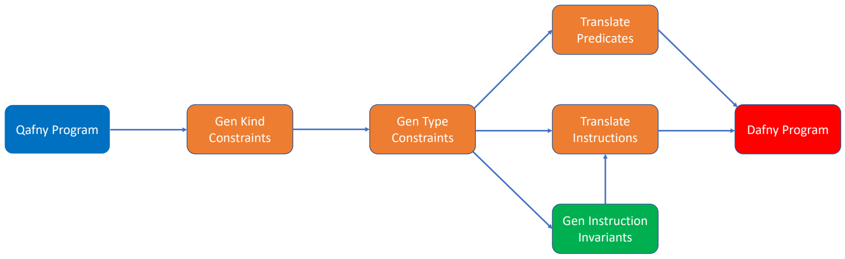

Fig. 7. Architectural Overview of the Qafny^c Compiler

3.1 Qafny Syntax and A Running Example

OQASM Expr	μ		
Arith Expr	a	::=	$x \mid v \mid a + a \mid a \cdot a \mid \ldots$
Bool Expr	b	::=	$x[a] \mid (a_1 = a_2)\ \texttt{@}\ x[a] \mid (a_1 < a_2)\ \texttt{@}\ x[a] \mid \ldots$
Quantum Expr	op	::=	$\texttt{H} \mid \texttt{QFT}^{[-1]} \mid \texttt{DFF}(op, f) \mid \mu$
Statement	e	::=	$\{\} \mid x := a \mid p, x := \texttt{measure}(\kappa) \mid \kappa \leftarrow op$
		$\mid$	$e\ ;\ e \mid \texttt{if}\ (b)\ e \mid \texttt{for}\ \ j\ \in\ [a_1, a_2)\ (\texttt{invariant}\ P)^*\ \{e\}$
Method	$\mathcal{F}$	::=	$\texttt{method}\ f((x : \eta)^*)\ (\texttt{returns}\ (x : \eta)^+)?\ (\texttt{requires}\ P)^*\ (\texttt{ensures}\ Q)^*\ \{e^*\}$

Fig. 8. Qafny Syntax

Figure 8 shows the Qafny program syntax, which is imperative and designed with high-level quantum programming operations. A Qafny program is a multiset of methods ($\mathcal{F}$), each of which contains an optional `returns` clause for returned classical scalar values (quantum data do not need to return), `requires` and `ensures` clauses that specify the method's pre- and post-conditions via predicates. The method body consists of statements (e), including control flow constructs (SKIP ({}), for-loops, and quantum conditionals), classical assignments ($x := a$), computational basis quantum measurement ($p, x := \texttt{measure}(\kappa)$), and quantum *heap mutations* ($\kappa \leftarrow op$). A measurement operation measures the locus κ and stores the result classical value in x in a probability p. A heap mutation applies a quantum operation – such as state preparation operations (Hadamard gate H and quantum Fourier transformation gate QFT), a quantum arithemtic oracle operation (μ) [1], or a diffusion operator (DFF(op, f))–to a locus κ; such behavior is analogous to mutating a qubit heap array by applying the operation to each basis-ket element in the array.

A quantum diffusion operator (DFF(op, f)) [Brassard et al. 2002] motivates the AA type development, where it partitions all basis-kets in a quantum datum into two sets based on checking the Boolean function on the basis-kets, e.g., $f(|j\rangle) = \texttt{true}$, and increases the probability of the

[1] μ can define all quantum arithmetics, e.g., +1 (Figure 4). See [Li et al. 2022].

```
1  method HammingWeight($n$ : nat, $x$ : Q($n$), $y$ : Q($n$))
2  returns($v$ : nat, $r$ : real)
3   requires { $x[0,n)$ : NOR $\mapsto |\overline{0}\rangle$ }
4   requires { $y[0,n)$ : NOR $\mapsto |\overline{0}\rangle$ }
5   ensures { $x[0,n)$ : EN(1) $\mapsto \Sigma_{j=0}^{C(n,v)-1} \sqrt{\frac{1}{C(n,v)}}\,|j\rangle \wedge h(j) = v$ }
6   ensures $r = \frac{C(n,v)}{2^n}$
7  {
8    { $x[0,n)$ : NOR $\mapsto |\overline{0}\rangle * y[0,n)$ : NOR $\mapsto |\overline{0}\rangle$ }
9    $x[0,n) \leftarrow$ H;
10   { $x[0,n)$ : HAD $\mapsto \bigotimes^n \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) * y[0,n)$ : NOR $\mapsto |\overline{0}\rangle$ }
11   for $i \in [0,n)$
12     invariant { $x[0,i), y[0,n)$ : EN(1) $\mapsto \Sigma_{j=0}^{2^i-1} \frac{1}{\sqrt{2^i}} |j\rangle\,|h(j)\rangle$ }
13     invariant { $x[i,n)$ : HAD $\mapsto \bigotimes^n \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ }
14   {
15     if ($x[i]$) {$y[0,n) \leftarrow y[0,n) + 1$;}
16   }
17   { $x[0,n), y[0,n)$ : EN(1) $\mapsto \Sigma_{j=0}^{2^n-1} \frac{1}{\sqrt{2^n}} |j\rangle\,|h(j)\rangle$ }
18   $r, v \leftarrow$ measure($y[0,n)$);
19   { $x[0,n)$ : EN(1) $\mapsto \Sigma_{j=0}^{C(n,v)-1} \sqrt{\frac{1}{C(n,v)}}\,|j\rangle \wedge h(j) = v$ }
20 }
```

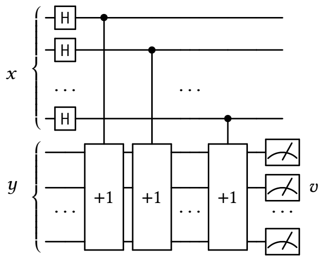

Fig. 9. Qafny$^{\mathsf{C}}$ proof for one-step Hamming weight state preparation (repeat-until-success); circuit on the right; $|h(j)\rangle$ is the basis-vector for the Hamming weight of bitstring k; $C(n,v)$ is the binomial coefficient. The grayed-out state predicates are automatically inferred.

basis-kets answering true; The *op* operation acts as an additional probability adjustment. More details in Section 4. The operation can be defined as a combination of state preparation and oracle operations. It is defined as a primary operation in Qafny due to its common uses in many amplitude amplification algorithms. Quantum reversible Boolean guards b are implemented as OQASM oracle operations, expressed by one of $(a_1 = a_2)\,@\,x[a]$, $(a_1 < a_2)\,@\,x[a]$, and $x[a]$, which intuitively amounts to computing $a_1 = a_2$, $a_1 < a_2$ and `false` respectively as b_0 and storing the result of $b_0 \oplus x[a]$ as a binary in qubit $x[a]$.[2] In both conditionals and loops, guards b are used to represent the qubits being controlled. We assume additional Dafny definable classical operations are available beyond the Qafny syntax in Figure 8, such as classical conditionals.

The example in Figure 9 prepares a quantum state with basis-kets being v-th hamming weight, i.e., the number of ones in each basis-vector is v. Here, we utilize two length n qubit arrays x and y, where $x[0,n)$ are in a uniform superposition by applying n H gates in line 9. We then apply n controlled-additions in lines 11 - 17 – a for-loop with each step i applying a quantum conditional in line 16 to $x[i]$ and $y[0,n)$ in the locus $x[0,i+1) \uplus y[0,n)$. The measurement operation in line 18 outputs the value $v - v = k$ means a successful preparation of a k-th Hamming weight state.

The `requires` clauses in lines 3 and 4 specify that both x and y start in the NOR-typed $|\overline{0}\rangle$ state, as the precondition. The `ensures` clauses specify the postconditions to be verified: that the final entangled state correctly maps each basis state $|j\rangle$ to its hamming weight $|h(j)\rangle$ (line 5), and that the probability of measuring a weight v follows a binomial distribution (line 6).

The core logic is a for-loop, which iterates classically with a C-kind iterator i. Inside the loop is a quantum *conditional* (`if` (b) $\{e\}$) controlled by a quantum Boolean guard (b). Here, the guard is

[2] a_1 and a_2 can possibly apply to a range, like $y[0,n)$, in an entangled locus.

$x[i]$, a quantum reversible Boolean guard that checks the state of the i-th qubit for each basis ket in the suposition. If the guard is met (i.e., if $x[i]$ is in the $|1\rangle$ state), another heap mutation applies an adder oracle (μ) to the y locus. Lines 12 and 13 describe the invariants capturing the quantum conditional behavior. All qubits are arranged as two disjoint loci, $x[0,i) \uplus y[0,n)$ and $x[i,n)$, where the former represents the result of the i-th looping of the quantum conditional in line 15, and $x[i,n)$ represents the remaining qubits have not been dealt with. The quantum conditional in line 15 moves a qubit $x[i]$ to turn the first locus to be $x[0,i{+}1) \uplus y[0,n)$. we apply the addition only if $x[i]$'s basis-vector fragment is $|1\rangle$. As in Figure 4, such an operation creates an AA typed state with two different cases. Clearly, both of them can be rewritten to the invariant state in line 12, because $y[0,n)$ always stores the Hamming weights for every basis-vector fragments in $x[0,i{+}1)$. The invariant in line 13 is also maintained, because every loop step reduces a qubit $x[i]$ in $x[i,n)$.

3.2 Static Constraint Generation

The first Qafny^{c} compilation stage is to perform a static analysis, based on our type system, for an input Qafny program. Such a step generates static type constraints for variables and loci, for our type-based translation steps in the later stage.

Generate Kind Constraints. We first traverse the Qafny program to learn the kinds of variables. For example, the variables n, v and r are recognized as C-kind (`nat` and `real`) in Figure 9, while x and y are Q(n) kind. This information is used to determine the corresponding Dafny variable types in the generated code.

Generate Type Constraints. Next, the compiler derives the quantum types for the different loci at all points in the program. This process, which is crucial for enabling type-based compilation of operations, involves two main subroutines.

First, the compiler collects the function-level input and output types for each locus based on the `requires` and `ensures` in the method header. In Figure 9 lines 2-3, we detect the input type for loci $x[0,n)$ and $y[0,n)$ as NOR, and the output type to be EN(1) for the locus $x[0,n)$ in line 4. Then, we model the constraint for the locus $x[0,n)$ as a transition NOR $\longrightarrow$ EN(1), and $y[0,n)$'s constraint as a transition NOR $\longrightarrow \emptyset$, i.e., the locus is destroyed after the method execution.

Second, the compiler tracks instruction-level locus-type changes, where we track the locus and type changes during instruction executions. For example, the operation in Figure 9 line 9 modifies the locus $x[0,n)$ from NOR to HAD. Furthermore, the compiler uses the `invariant` clauses (lines 12-13) to determine the locus-type change in a loop step, to extend the locus $x[0,i) \uplus y[0,n)$ to be $x[0,i{+}1) \uplus y[0,n)$ by incrementing 1 in the x part, as well as shrink the locus $x[i,n)$ to $x[i{+}1,n)$.

This instruction-level type inference is critical for compiling Qafny statements correctly. For each statement, the compiler infers its locus-type constraints generated via the above procedures, which are used to determine if a type cast is needed to ensure the type consistency. For example, if a statement produces a HAD typed datum but the subsequent operation expects an input of type EN(1), the compiler must automatically insert a cast operation to bridge the mismatch. This mechanism ensures the generated Dafny code respects Qafny's type semantics and preserves the intended quantum behavior.

3.3 Type-Driven Data Representation and Translation to Dafny

With the generated constraints, we first translate the Qafny program's predicates, such as the `requires`, `ensures`, and `invariant` clauses, into corresponding Dafny entities. The translation is *type-based*, adopting different strategies and using different templates depending on the generated Qafny type constraints of the loci being processed. This approach ensures that the semantics of quantum operations are preserved in the compiled Dafny operations based on Dafny *immutable*

Table 1. Qafny state to Dafny data structure mapping on a single range locus $x[j,k)$.

Qafny Quantum Datum	Dafny Encoding	Bound
$x[j,k) : \mathsf{NOR} \mapsto z\beta$	`amp : real, x : seq<bv1>`	`|x| = k-j`
$x[j,k) : \mathsf{HAD} \mapsto \frac{z}{\sqrt{2^{k-j}}} \bigotimes_{i=j}^{k} (\|0\rangle + \omega(r_i)\|1\rangle)$	`amp : real, x : seq<real>`	`|x| = k-j`
$x[j,k) : \mathsf{EN}(n) \mapsto \sum_{j_0=0}^{m_1} \dots \sum_{j_{n-1}=0}^{m_{n-1}} w_{j_0,\dots,j_{n-1}}$	$\texttt{amp:}\underbrace{\texttt{seq <.. seq<}}_{n}\texttt{real >..> , x:}\underbrace{\texttt{seq <.. seq<}}_{n+1}\texttt{bv1 >..>}$	See Definition 3.1
$x[j,k) : \mathsf{AA}(1) \mapsto \alpha(f(x[j,k)), z_1) + \alpha(\neg f(x[j,k)), z_2)$	`amp : seq<real>, x:seq<seq<bv1>>`	`|x| = |amp| = 2`

sequence data structures. Qafny's classical values are directly mapped to their Dafny counterparts, as Figure 9 line 6 is translated to Figure 10 line 15. We primarily introduce quantum data representations in Dafny, where the translation of quantum predicates becomes trivial.

Table 1 demonstrates the translation of different types of Qafny's quantum data, for a single range, to classical Dafny data structures, unveiling the general quantum data compilation strategy. Mainly, quantum data are encoded primarily using Dafny's real number (`real`) type for amplitudes and nested sequences (`seq`) of bitvectors (`bv1`) for basis-vectors. The use of immutable sequences is a key design choice in the Qafny$^{\mathsf{C}}$ compiler, which is a direct improvement over the original Qafny, as it avoids the complex analysis required to determine if mutable arrays are modified by the insertion of `modifies` clauses, often necessitating manual insertions.

For a NOR typed datum, which represents a classical basis-ket such as $|\overline{0}\rangle$, the compiler emits two Dafny variables: a `real` amplitude `amp` and a basis vector `x`, represented as a length $k - j$ bitvector sequence (`seq<bv1>`) corresponding to the locus range $x[j,k)$. Each bitvector in x encodes one qubit's basis-vector $|0\rangle$ or $|1\rangle$. For example, the single range locus $x[0,n)$ (or $y[0,n)$) in Figure 9 is unzipped into a pair `amp1` and `x` (`amp2` and `y`) in Figure 10. The amplitude part of $x[0,n)$'s state predicate of Figure 9 line 3 is translated to Figure 10 line 3, and $x[0,n)$'s basis-vector ($|\overline{0}\rangle$) is translated to Figure 10 line 8. We also encode $x[0,n)$'s bound predicate to Figure 10 line 5 as `|x| == n`. Locus $y[0,n)$ is translated similarly. The complex amplitude encoding is discussed in Section 3.5.

A HAD-typed datum models a quantum state in which each qubit is in an independent superposition, following the product form $x[j,k) : \mathsf{HAD} \mapsto \frac{z}{\sqrt{2^{k-j}}} \bigotimes_{i=j}^{k} (|0\rangle + \omega(r_i)|1\rangle)$, where z is an amplitude and $\omega(r_i)$ encodes a local phase parameterized by r_i. When compiling to Dafny, we generate a `real` variable `amp` for z, and a sequence `x: seq<real>` where each element $x[i]$ holds the real parts (r_i) of local phases $\omega(r_i)$, where we enforce `x`'s length to be $k-j$, same as the qubit array length. This representation permits the reasoning of separable superposition data without explicitly encoding complex phases.

In an EN(n)-typed datum, the flag n represents the number of sum symbols, as in Table 1, i.e., an $\mathsf{EN}(n)$-typed quantum datum has n summation symbols. We compile the amplitudes to an n-dimensional sequence of real numbers `amp`, and the corresponding basis vectors are represented as a $(n+1)$-dimensional sequence of bitvectors `x`. In Figure 10, the variables `amp1_f` and `x_f` represent the transformed Dafny data structures for $x[0,n)$'s post-state in Figure 9 line 5. We also generate their bound information for each dimension, where we create a predicate in line 10 to suggest that both `amp_f` and `x_f` have length `C(n,v)`, corresponding to $x[0,n)$'s basi-ket bound (sum-term bound), and line 11 suggest that each element in `x_f` has a bound of n, corresponding to $x[0,n)$'s qubit bound. In translating the predicates describing a $\mathsf{EN}(n)$-typed datum, e.g., translating Figure 9 line 5, we generate Dafny predicates with an n nested universal quantified structures, e.g., `forall k :: 0 ≤ k < bound ==> ...`, with the proper bound setup. In the above example, we generate Figure 10 lines 12 and 13, for describing $x[0,n)$'s amplitudes and basis-vectors, i.e., the basis-vector of each basis-ket is the Hamming weight `ham(x_f[i], n)` is `v_f`, equal to `v` in line 14. `ham` is our library

```
method HammingWeight(amp1: real, x: seq<bv1>, amp2: real, y: seq<bv1>, n: nat, v_f: nat)
    returns (x_f : seq<seq<bv1>>, amp1_f : seq<real>, v : nat, r : real)
  requires amp1 == 1.0
  requires amp2 == 1.0
  requires |x| == |y| == n
  requires n > 0
  requires 0 ≤ v_f ≤ n
  requires ∀k : : 0 ≤ k < |x| ⟹ x[k] == 0
  requires ∀k : : 0 ≤ k < |y| ⟹ y[k] == 0
  ensures |x_f| == |amp1_f| == C(n,v_f)
  ensures ∀k : : 0 ≤ k < C(n, v_f) ⟹ |x_f[k]| == n
  ensures ∀k : : 0 ≤ k < C(n, v_f) ⟹ amp1_f[k] = 1.0 / sqrt(C(n,v_f) as real)
  ensures ∀k : : 0 ≤ k < C(n, v_f) ⟹ ham(x_f[k],n) == v_f
  ensures v == v_f
  ensures r == C(n, v_f) as real / pow2(n) as real
{
  ...
}
```
Fig. 10. Compiled Dafny method header code for HammingWeight.

function computing the number of ones in a length n bitstring. The reason to include an additional v_f input is for our measurement handling, discussed shortly below.

If a locus has more than two ranges, e.g., locus $x[j, k) \uplus y[0, t)$, we use two different variables to represent basis-vector fragments for each range in the locus. Assume that the locus type is EN(2), we then generate two variables x and y, having Dafny type seq<seq<seq<bv1>>>, to represent the basis vector fragments in $x[j, k)$ and $y[0, t)$, respectively.

Below, we define the length of different dimensions of the compiled EN(n)-typed sequences for representing amplitudes and basis vectors. We only list a single range $x[j, k)$ of the basis-vector for a locus κ, and the situation for the other ranges can be defined in a similar manner. Here, x[i_0 ,..., i_n] refers to x[i_0][i_1]...[i_n].

Definition 3.1 (EN Typed Dimension Length). Given a real number and bitvector sequence amp and x, for representing the amplitudes and basis vector for the range $x[j, k)$ in a locus κ for a EN(n) typed state $\sum_{i_0=0}^{m_0} \dots \sum_{i_{n-1}=0}^{m_{n-1}} w_{i_0,\dots,i_{n-1}}$, the $(t + 1)$-th dimension, when $t \in [0, n)$ length for amp and x, as |amp[i_0 ,..., i_t]| and |x[i_0 ,..., i_t]|, is m_t. The length of the $n + 1$ dimension of x, as |x[i_0 ,..., i_n]|, is $k - j$, i.e., the qubit length of the range $x[j, k)$.

As indicated above, a quantum operation is essentially compiled into nested Dafny loop structures, looping over n-dimensional sequences representing an EN(n)-typed datum basis-kets. Complicated nested loop structures might be ineffective for Dafny to handle. In such a case, Qafny^C flattens nested loop structures as Dafny methods, containing simple loops, to permit the automated verification of such operations for EN(n)-typed datum. To generate the flattened method for a nested loop, observe that the inner loop invariants can be automatically inferred, similar to the generation of loop invariants in Figure 6 lines 16-20. We then replace the inner loop with a generated Dafny method holding the inner loop structure with automatically generated requires and ensures clauses inferred based on the inner loop invariants.

The AA(1) typed quantum data transformation is defined in the last row of Table 1 for a single range locus, where the other cases are defined similarly. An AA typed quantum datum,

```
method MergeTypeHadEn(amp : seq<real> , x: seq<seq<bv1>>, y : seq<seq<bv1>>, amp_xj
    : real, x_j : real, j: nat, n: nat, m: nat)
  returns (amp_c : seq<real> x_c : seq<seq<bv1>>, y_c : seq<seq<bv1>>)
  requires |amp| == |x| == |y| == m // m is the number of basis-kets in x, y, amp
  requires ∀k : : 0 ≤ k < m ⟹ |x[k]| == j
  requires ∀k : : 0 ≤ k < m ⟹ |y[k]| == n
  ensures |amp_c| == |x_c| == |y_c| == 2 * m
  ensures ∀k : : 0 ≤ k < 2 * m ⟹ |x_c[k]| == n + 1
  ensures ∀k : : 0 ≤ k < m ⟹ amp_c[k] == amp_xj / sqrt(2) * amp[k]
  ensures ∀k : : m ≤ k < 2 * m ⟹ amp_c[k] == amp_xj / sqrt(2) * x_j * amp[k]
  ensures ∀k : : 0 ≤ k < m ⟹ ∀j : : 0 ≤ j < n ⟹ x_c[k][j] == x[k][j]
  ensures ∀k : : 0 ≤ k < m ⟹ x_c[k][n] == 0
  ensures ∀k : : m ≤ k < 2 * m ⟹ ∀j : : 0 ≤ j < n ⟹ x_c[k][j] == x[k][j]
  ensures ∀k : : m ≤ k < 2 * m ⟹ x_c[k][n] == 1
  ensures ∀k : : 0 ≤ k < m ⟹ y_c[k] == y[k]
  ensures ∀k : : m ≤ k < 2 * m ⟹ y_c[k] == y[k]
{
...
}
```

Fig. 11. The implementation of MergeTypeHadEn

$\alpha(f(x[j,k)) = b_1, z_1) + \alpha(f(x[j,k)) = b_2, z_2)$, is represented by two length 2 sequences in Dafny, a basis-vector `x : seq<seq<bv1>>` and a real number `amp : seq<real>` sequence, with `|x|==|amp|==2`. The former stores a representative basis-vector based on a boolean function f, with `not f(x[0])` and `f(x[1])`, while the latter stores z_1 and z_2 for `x[0]` and `x[1]`, if we set b_1 = `false` and b_2 = `true`. z_1 and z_2 correspond to the sum of amplitudes for basis vectors where f answers `false` and `true` respectively. An AA type is extended upon EN-types and they can be used together; see Section 4.

3.4 Type-Driven Program Statement Translation

To translate program statements, we utilize a library of predefined templates and functions, selecting the appropriate one based on the inferred type of the locus being operated on. During this process, the compiler generates new, fresh Dafny variables to represent the post-state of any modified locus, similar to the use of SSA (Static Single Assignment) form in classical compilers. Some compilations are discussed in Section 2.3, here we show two key statement translations.

Control Flow Constructs Translation. A Qafny for-loop is a classical iteration construct, translated into a standard Dafny **while** loop with a counter, e.g., $i < n$ in Figure 9 line 11. The main task for the compiler is to translate the loop's body and automatically generate the corresponding Dafny `invariant` clauses from the Qafny specifications. The generated `invariants` for line 12 are listed as follows.

```
invariant |x| == |y| == |amp|
invariant ∀k : : 0 ≤ k < |x| ⟹ |x[k]| == i
invariant ∀k : : 0 ≤ k < |x| ⟹ |y[k]| == n
invariant ∀k : : 0 ≤ k < |x| ⟹ amp[k] == 1.0 / sqrt(pow2(n) as real)
invariant ∀k : : 0 ≤ k < |x| ⟹ x[k] == k
invariant ∀k : : 0 ≤ k < |x| ⟹ b2n(y[k], n) == ham(x[k], i)
```

Below is the translation of a quantum conditional (`if` (b) e). Before the conditional can be compiled, we must ensure the control qubit and target are in the same locus. Qafny^C utilizes template

```
Lemma HamUp(x: seq<seq<bv1>>, i: nat, y: seq<seq<bv1>>, y_f: seq<seq<bv1>>, n: nat)
  requires |x| == |y| == |y_f|
  requires ∀j : : 0 ≤ j < |x| ⟹ |x[j]| == i + 1
  requires ∀j : : 0 ≤ j < |x| ⟹ |y[j]| == n
  requires ∀j : : 0 ≤ j < |x| ⟹ |y_f[j]| == n
  requires ∀j : : 0 ≤ j < |x| ⟹ b2n(y[j], n) == ham(x[j], i)
  requires ∀j : : 0 ≤ j < |x| ⟹ b2n(y_f[j],n) == b2n(y[j],n) + x[j][i] as nat * 1
  ensures ∀j : : 0 ≤ j < |x| ⟹ b2n(y_f[j], n) == ham(x[j], i + 1)
{
  ...
}
```

Fig. 12. Lemma for enforcing Hamming weight constraint.

functions to enable several common qubit merging patterns, enough to verify all the programs in Section 5. We show an example template in Figure 9, where the HAD-typed control qubit $q[i]$ must be merged with the EN(1) typed locus $x[0,i) \uplus y[0,n)$, as we extend the locus to be $x[0,i{+}1) \uplus y[0,n)$. The compiler automates this by inserting a call to a library function MergeTypeHadEn in Figure 11. As we can see from the ensures clauses, this function forms a new EN-typed state and merges the loci by doubling the number of basis-kets (line 6), updating the amplitudes (lines 8 - 9), incrementing the basis-vector lengths in each basis-ket (line 7), concatenating the Dafny bitvector representation of $|0\rangle$ and $|1\rangle$ to the corresponding locus's basis-vector (lines 11 and 13), and connecting the new basis-kets with the old ones (lines 10, 12, 14, and 15).

After the merging step, we apply the same controlled addition semantic function IfAdder in Section 2.3 with t=i, k=i+1, and m=1, eventually generating the following constraint.

```
∀ j : : 0 ≤ j < |y_f| ⟹ b2n(y_f[j],n) == b2n(y_c[j],n) + x_c[j][i] as nat * 1
```

Recall that the quantum conditional in Figure 9 is inside a for-loop, with the above invariant constraints, where we apply a post-processing step after the IfAdder computation to restore the invariants via a Dafny lemma HamUp in Figure 12; such a lemma ensures that y_f[j] stores the Hamming weight results for x[j] after we add a bit to x[j] .

Measurement Statements Translation. Quantum measurement ($r, x := \texttt{measure}(\kappa)$) nondeterministically selects a basis-vector c in a quantum datum, $\sum_j z_j\, |c\rangle\, \beta_j + \sum_k z_k \beta_k$ and outputs c with a probablity $r = \sum_j |z_j|^n$, potentially left with a quantum state $\sum_j \frac{z_j}{\sqrt{r}} \beta_j$ if the measurement is a partial one. Verifying a measurement operation typically refers to determining whether a certain probability and the resulting quantum state are correct given a particular measurement outcome. Such an outcome could be symbolic, characterized by a property that users want to attribute to the measurement. In compiling a quantum measurement, Qafny$^{\text{c}}$ automatically generates a ghost input value, e.g., generating v_f for v in Figure 9, as the assumed measurement outcome for constructing the measurement outcome postconditions. In Figure 9, after applying controlled-additions, we measure the range $y[0,n)$ and obtain v_f. Since each basis-vector fragment $y[0,n)$ stores the Hamming weight ($h(k)$) of the fragment in $x[0,n)$, v_f represents the Hamming weight number of the leftover state in $x[0,n)$, i.e., $x[0,n)$ is a superposition of all basis-kets with basis-vectors being v_f Hamming weight. Before the measurement, the quantum datum in $x[0,n) \uplus y[0,n)$ is in unform superposition, each basis-ket with an amplitude $\frac{1}{\sqrt{2^n}}$. Clearly, the measurement probability ($\sqrt{\frac{C(n,v)}{2^n}}$), solely depends on the number of possible v_f Hamming weight basis-vectors in the datum, as $C(n,v)$. In compiled Dafny code, we use a while-loop to collect all basis-kets with basis-vectors being v_f Hamming weight as for the new quantum datum for $x[0,n)$ described in Figure 10 lines 10 - 13, and use a lemma to suggest that the number of v_f Hamming weight basis-vectors is C(n, v_f) to infer the postcondition for the probability in Figure 10 lines 15.

3.5 The Qafny Support Library: Templates and Axioms

The Qafny^c compilation pipline does not operate in isolation; it is supported by a comprehensive library of predefined Dafny functions, templates, and axioms. This library is a critical component of our framework, providing the necessary abstractions to bridge the semantic gap between high-level quantum operations and the classical verification logic of Dafny. The library serves two primary functions: providing structural templates for state manipulation and axiomatizing core mathematical concepts from quantum mechanics.

Structural Templates. Many quantum operations require complex, yet repetitive, transformations of the underlying Dafny sequence representations. To manage this, Qafny^c utilizes library template functions. These include routines for structural changes, such as splitting and merging loci, as well as templates for type casting (e.g., converting a HAD typed state to an EN(1) typed state). A key example is the MergeTypeHadEn in Figure 11, which encapsulates the complex logic required to merge a separable qubit into an entangled locus, a common pattern in algorithms as shown in Figure 9. By providing these as pre-verified library functions, we ensure that these common operations are handled correctly and consistently without cluttering the main compilation logic.

Axiomatization of Quantum Mathematics. Dafny lacks native support for complex numbers and offers only limited theories for trigonometry, both of which are fundamental to quantum operations. While these amplitudes are complex numbers, quantum programs represent them in special forms, e.g., an EN typed datum models an amplitude as a real-valued magnitude paired with a phase ($e^{i\theta}$) where θ is real, and the Grover's algorithm expresses amplitudes directly as trigonometric functions. We capture the amplitudes' operational behaviors by modeling them as axioms and lemmas based on Dafny reals and utilize Dafny real number library to automate equational rewrites. For example, we define the root-of-unity function, omega(n, m) as an uninterpreted Dafny function and provide axioms that specify its core algebraic properties, such as omega(n1, m) * omega(n2, m) == omega(n1+n2, m). This allows Qafny^c compiler to generate verifiable code for phase-based arithmetic by asserting these known mathematical truths, enabling Dafny's SMT solver to reason about quantum phenomena without needing a full theory of complex arithmetic.

4 Case Study: Amplitude Amplification

We demonstrate our verification capabilities through the proofs of the group of amplitude amplification algorithms [Brassard et al. 2002]. Rather than analyzing individual basis-kets, this group employs a statistical approach that partitions all basis-kets into two sets, labeled with "good" (marked) or "bad" (not marked), treats the geometric sums of basis-ket amplitudes in each set as the event probabilities, and systematically increases the probabilities of good events. The effective encapsulation of this statistical perspective coincides with our AA-typed datum (named after amplitude amplification), designed to model the aggregate probabilistic behavior of the system.

4.1 Grover's Algorithm

The amplitude amplification algorithmic group originates from Grover's quantum search algorithm [Grover 1996], which solves the problem of searching for a key in an unstructured database with quadratic speedup over classical methods. Unlike classical search algorithms that check database entries iteratively, Grover's algorithm amplifies the amplitude z of basis-kets β for a Boolean function $f(\beta) = \texttt{true}$. Through repeated iterations, the probability of measuring a correct key approaches 1. Figure 13 demonstrates the Qafny proof of the Grover's search algorithm.

```
1  method Grovers($x$ : Q($n$), $f$ : nat → bool, $n$ : nat, $m$ : nat) returns ($p$ : real, $y$ : nat)
2   requires { $x[0,n)$ : NOR $\mapsto |\overline{0}\rangle$ }
3   ensures $f(y)$ = true
4   ensures $p = \sin^2((2m+1)\theta)$
5  {
6    $x[0,n)$ ← H;
7    { $x[0,n)$ : HAD $\mapsto \frac{1}{\sqrt{2^n}} \bigotimes_{j=0}^{n-1} (|0\rangle + |1\rangle)$ }
8    for $k \in [0,m)$
9      invariant { $x[0,n)$ : AA(1) $\mapsto \alpha(f(x[0,n)), \sin((2k+1)\theta)) + \alpha(\neg f(x[0,n)), \cos((2k+1)\theta))$ }
10     $x[0,n)$ ← DFF(H, $f$);
11     { $x[0,n)$ : AA(1) $\mapsto \alpha(f(x[0,n)), \sin((2k+1)\theta)) + \alpha(\neg f(x[0,n)), \cos((2k+1)\theta))$ }
12   $p, y$ := measure($x[0,n)$);
13 }
```

Fig. 13. Grover's algorithm proof. $\theta = \arcsin(\sqrt{r})$

The `Grovers` program takes in a qubit array x with length n, a Boolean function f that identifies marked events, the initial marked event probability r, and iteration count m. While f alone suffices for the execution of the program, r is included for verification purposes.

We utilize the AA type to verify the program. In line 6, applying the Hadamard operation to the qubit array $x[0,n)$ results in the HAD typed state in line 7, the state is then implicitly cast to the AA-typed state before the diffusion operation, formally represented as:

$$\alpha(f(x[0,n)), \sin(\theta)) + \alpha(\neg f(x[0,n)), \cos(\theta)),$$

where the superposition is partitioned into two basis-ket sets, determined by evaluation of f to each basis-ket. Here $\sin(\theta)$ and $\cos(\theta)$ are the geometric sums of the amplitudes for their respective partitions, ensuring normalization as $\sin^2(\theta) + \cos^2(\theta) = 1$. We demonstrate the typing mechanism through an illustrative example. Consider the superposition state initially presented in line 7, which we rewrite as an EN(1) typed state $\sum_{j=0}^{2^n-1} \frac{1}{\sqrt{2^n}} |j\rangle$: If $|2^n - 1\rangle$ is the sole marked item in the database, f essentially partitions the state into two parts, $\frac{1}{\sqrt{2^n}} |2^n - 1\rangle$ and $\sum_{j=0}^{2^n-2} \frac{1}{\sqrt{2^n}} |j\rangle$, with the former f answers `true` and the latter answers `false`. Here the angle θ is determined by the marked state's amplitude $\arcsin(\frac{1}{\sqrt{2^n}})$, reflecting that only one state satisfies f.

$$\left\{ \begin{array}{l} \alpha(f(x[0,n)), \sin(\theta')) \\ +\alpha(\neg f(x[0,n)), \cos(\theta')) \end{array} \right\} x[0,n) \leftarrow \mathtt{DFF}(\mathtt{H}, f) \left\{ \begin{array}{l} \alpha(f(x[0,n)), \sin(\theta'+2\theta)) \\ +\alpha(\neg f(x[0,n)), \cos(\theta'+2\theta)) \end{array} \right\}$$

The diffusion operation (DFF(H, f)) rotates the state amplitudes by angles of 2θ, resulting in amplitudes $\sin(\theta' + 2\theta)$ and $\cos(\theta' + 2\theta)$, where θ' is a multiple of inital angle θ. This allows us to maintain the AA typed loop invariant in line 9. The translation of the program involves two key steps: (1) formalizing the diffusion operation DFF(H, f) and its associated AA-typed loop invariant in line 9 as a Dafny predicate; and (2) implementing the measurement operation via probabilistic assertions on the abstracted state representation. Additionally, the AA-typed state predicate for locus $x[0,n)$ is transformed into two length-2 sequences (amp and x) of real numbers and bitvectors as in Table 1, enabling Dafny to verify the statistical amplitude properties symbolically.

```
amp[0] == cos((2 * k + 1) as real * theta) //for false case of f
amp[1] == sin((2 * k + 1) as real * theta) //for true case of f
```

Before the above predicate translation, we insert a casting operation (CastTypeHadAA) - similar to MergeTypeHadEn in Figure 11 - immediately after the translation of line 7 in Figure 13. This operation explicitly converts the type of qubits $x[0, n)$ from HAD to AA. The translation further requires two axiomatic declarations of the basis-vector sequence x_c, shown in lines 2 and 3 below. These axioms are essential for preserving the correspondence between the AA-type state and the underlying quantum basis-vectors. x_c[0] and x_c[1] serve as canonical representatives for the basis-kets partitioned by f, where f(b2n(x_c [0], n)) is false and f(b2n(x_c [1], n)) is true.

```
1 amp, x_c :=CastTypeHadAA(x);
2 assume {: axiom} f(b2n(x_c[0], n)) == false;
3 assume {: axiom} f(b2n(x_c[1], n)) == true;
```

A diffusion operation DFF(op, f) is translated to a Dafny function, based on the state predicate transformed from the casting and axioms above. In Dafny, we pre-define a template function diff. With a unitary operation op and Boolean function f, we instantiate the predicates in the template function to a Dafny method diff.

```
method diff(amp : seq<real>, theta : real) returns (amp_f : seq<real>)
  requires |amp| == 2
  requires amp[0] == cos(theta)
  requires amp[1] == sin(theta)
  ensures |amp_f| == 2
  ensures amp_f[0] == cos(theta + 2.0 * theta)
  ensures amp_f[1] == sin(theta + 2.0 * theta)
{
  ...
}
```

The measurement operation in Qafny^c is determinized for verification purposes during the translation to Dafny. Given that amp_f[1] stores the amplitude value for the marked states satisfying f, and x_c[1] serves as representative of such states, the measurements is simply: (1) assigning pow2(amp_f[1]) to p and (2) yielding x_c[1] as the deterministic y. In verification, we do not need to output a particular basis-vector, but preserve the property that f(b2n(y, n))==true.

```
p := amp_f[1] * amp_f[1]
y := x_c[1]
```

4.2 Amplitude Estimation

This section discusses another algorithm in the amplitude amplification group, amplitude estimation [Brassard et al. 2002], which estimates the probability of marked items in a database. The algorithm combines Grover's diffusion operations with quantum phase estimation, the key step in Shor's algorithm. Figure 14 shows the main part of the Qafny proof. The AEst defined therein has arguments similar to the Grovers program above, but it does not require the m parameter for setting a fixed number of loop steps.

The Amp program verification is a subroutine of the proof in Figure 13, while the procedure of the AEst program verification is similar to the verification of the loop verification in Figure 9, except that we conduct the proof via a AA typed state here.

To understand the loop step proof in Figure 14 line 16, note that $x[k]$'s state type is HAD, and the quantum conditional in line 18 controls the qubit $x[k]$ for the application of the body expression to the locus $x[0, k) \uplus y[0, n)$. Thus, we first combine $x[k]$ and the locus by concatenating the $x[k]$ qubit to the locus, doubling the basis-ket numbers in the locus' datum:

$$\Sigma_{j=0}^{2^k-1} \frac{1}{\sqrt{2^k}}|j\rangle\,|0\rangle\alpha(f(x[0,n)), \sin((2j+1)\theta)) + \Sigma_{j=0}^{2^k-1} \frac{1}{\sqrt{2^k}}|j\rangle\,|0\rangle\alpha(\neg f(x[0,n)), \cos((2j+1)\theta))$$
$$+\Sigma_{j=0}^{2^k-1} \frac{1}{\sqrt{2^k}}|j\rangle\,|1\rangle\alpha(f(x[0,n)), \sin((2j+1)\theta)) + \Sigma_{j=0}^{2^k-1} \frac{1}{\sqrt{2^k}}|j\rangle\,|1\rangle\alpha(\neg f(x[0,n)), \cos((2j+1)\theta))$$

```
1   method Amp($z : \mathtt{Q}(n), f : \mathtt{nat} \rightarrow \mathtt{bool}, n : \mathtt{nat}, m : \mathtt{nat}$)
2    requires { $z[0,n) : \mathtt{AA}(1) \mapsto \alpha(f(z[0,n)), (\theta_0)) + \alpha(\neg f(z[0,n)), \cos(\theta_0))$ }
3    ensures { $z[0,n) : \mathtt{AA}(1) \mapsto \alpha(f(z[0,n)), \sin(2m * \theta + \theta_0)) + \alpha(\neg f(z[0,n)), \cos(2m * \theta + \theta_0))$ }
4   {
5     for $k \in [0,m)$
6       invariant { $z[0,n) : \mathtt{AA}(1) \mapsto \alpha(f(z[0,n)), \sin(2k * \theta + \theta_0)) + \alpha(\neg f(z[0,n)), \cos(2k * \theta + \theta_0))$ }
7       $z[0,n) \leftarrow$ DFF(H, $f$);
8   }
9   method AEst($x : \mathtt{Q}(n), y : \mathtt{Q}(n), f : \mathtt{nat} \rightarrow \mathtt{bool}, n : \mathtt{nat}$)
10   requires { $x[0,n) : \mathtt{NOR} \mapsto |\overline{0}\rangle * y[0,n) : \mathtt{NOR} \mapsto |\overline{0}\rangle$ }
11   ensures { $x[0,n) \uplus y[0,n) : \mathtt{AA}(1) \mapsto E(n)$ }
12  {
13    $x[0,n)$ ← H;
14    $y[0,n)$ ← H;
15    { $x[0,n) : \mathtt{HAD} \mapsto \frac{1}{\sqrt{2^n}} \bigotimes_{j=0}^{n-1} (|0\rangle + |1\rangle) * y[0,n) : \mathtt{HAD} \mapsto \frac{1}{\sqrt{2^n}} \bigotimes_{j=0}^{n-1} (|0\rangle + |1\rangle)$ }
16    for $k \in [0,n)$
17      invariant { $x[0,k) \uplus y[0,n) : \mathtt{AA}(1) \mapsto E(k) * x[k,n) : \mathtt{HAD} \mapsto \frac{1}{\sqrt{2^{n-k}}} \bigotimes_{j=k}^{n-1} (|0\rangle + |1\rangle)$ }
18      if ($x[k]$) {Amp($y, f, n, 2^k$)};
19  }
```

$$E(t) = \Sigma_{j=0}^{2^t-1} \frac{1}{\sqrt{2^t}} |j\rangle \alpha(f(y[0,n)), \sin((2j{+}1)\theta)) + \Sigma_{j=0}^{2^t-1} \frac{1}{\sqrt{2^t}} |j\rangle \alpha(\neg f(y[0,n)), \cos((2j{+}1)\theta))$$

Fig. 14. Core part of the amplitude estimation proof. The angle θ is defined as $\theta = \arcsin(\sqrt{r})$.

Here, the range $x[0,k)$ is extended to $x[0,k{+}1)$, putting bit $|0\rangle$ and $|1\rangle$ to the datum's corresponding $x[k]$ position. We apply the conditional body to the basis-kets with $x[k]$ being $|0\rangle$ but not the ones with $x[k]$ being $|1\rangle$. After the execution, the loop-invariant in line 17 is resumed where k is incremented by 1.

The AA typed state representation in Dafny is crucial in translating Figure 14. As we mentioned in Section 3, AA(1) type is extended on top of EN(1), where one can declare part of an EN(1)-typed datum to be in AA, e.g., the range $y[0,n)$ being AA typed, while the other range in the locus ($x[0,n)$) is EN(1) typed. Translating a partially AA typed datum like AA(1) is an extension of the EN(1) typed datum translation. Specifically, for ranges marked as EN(1), we generate variables for the amplitudes and basis vectors, whereas for each AA typed state, we generate two variables. To translate a locus of the form $x[0,n) \uplus y[0,n)$ with $y[0,n)$ being AA-typed, we generate the variables amp: seq<seq<real>>, x: <seq<seq<seq<bv1>>>, ampy: seq<seq<real>>, and y: seq<seq<seq<bv1>>>. The variables ampy and y capture the behavior of the range $y[0,n)$, similar to the one described in Section 4.1.

```
|amp| == |x| == |ampy| == |y| == pow2(n)
∀ j ::0 ≤ j < pow2(n) ⟹ |amp[j]| == |x[j]| == |ampy[j]| == |y[j]| == 2
∀ j ::0 ≤ j < pow2(n) ⟹ ∀k ::0 ≤ k < 2 ⟹ |x[j][k]| == |y[j][k]| == n
∀ j ::0 ≤ j < pow2(n) ⟹ ∀k ::0 ≤ k < 2 ⟹ amp[j][k] == 1.0 / pow2(n) as real
∀ j ::0 ≤ j < pow2(n) ⟹ ∀k ::0 ≤ k < 2 ⟹ b2n(x[j][k], n) == j
∀ j ::0 ≤ j < pow2(n) ⟹ f(y[j][0]) == false
∀ j ::0 ≤ j < pow2(n) ⟹ f(y[j][1]) == true
∀ j ::0 ≤ j < pow2(n) ⟹ ampy[j][0] == cos((2*j+1) as real * theta)
∀ j ::0 ≤ j < pow2(n) ⟹ ampy[j][1] == sin((2*j+1) as real * theta)
```

The above is the translated predicate capturing the $E(n)$ predicate in Figure 14. In these predicates, the outer sequences for the variables refer to the basis-kets in the range $x[0, n)$, so every outer sequence has a length of 2^n, shown as line 1. The inner sequences represent the partitioned two basis-ket sets in the AA typed $y[0, n)$ locus, with length 2 (line 2). Line 3 suggests that the basis-vectors of the x and y ranges both have length n. Line 5 describes that the basis-vector of the range $x[0, n)$ should be $|j\rangle$ for every $j \in [0, 2^n)$, while lines 6 - 7 categorize basis-kets according to the range $y[0, n)$, by partitioning them into two sets (y[j][0] and y[j][1]), where f answers `true` and `false` respectively. Correspondingly, lines 8 - 9 assign the amplitude sums for the two sets.

```
while (j < pow2(n))
  invariant ∀k : : 0 ≤ k < j ⟹ Post(k)
  invariant ∀k : : j ≤ k < pow2(n) ⟹ Pre(k)
{
  Amp(ampy[j], y[j], ...); //pseudocode, real Dafny code might require assignments
  j = j + 1;
}
```

One of the key Qafny^c features is that the program verification of a quantum state is localized to the qubits referenced within a function, independent of any other potentially entangled qubits. Figure 14 line 17 applies Amp to the range $y[0, n)$, which is defined based on the AA typed state and follows the sequence modeling in Section 4.1, i.e., the Amp function have the input arguments for the qubit array z, having the Dafny types ampz: seq<real> and z: seq<seq<bv1>>. To apply Amp to the range $y[0, n)$, Qafny^c generates a **while** loop that iterate over each element in the sequences of ampy and y, invoking Amp on each element ampy[j] and y[j], shown above. During this process, Qafny automatically generates the predicate Post(k) for elements in the range $[0, j)$ and the predicate Pre(k) for elements in $[j, 2^n)$. Post(k) is generated via the invariant $E(k)$ after executing the quantum conditional in line 18, while Pre(k) is generated before executing the conditional.

5 Evaluation

This section presents an experimental evaluation of Qafny^c. We first define our benchmark suite of quantum programs and the baseline of state-of-the-art frameworks for comparison. We then present our results, structured to answer the following research questions:

- RQ1: Applicability and Scalability: How does Qafny^c perform across the benchmark suite in terms of the breadth of the verifiable quantum programs and its scalability?
- RQ2: Qualitative Comparison: How does Qafny^c's verification methodology differ from other state-of-the-art quantum verification works?

5.1 Experimental Setup

Benchmark suite. To evaluate Qafny^c, we use a benchmark of 37 diverse quantum programs. These programs span all major families of quantum computation, from foundational protocols to complex QFT-based algorithms. The full list is available in Table 2.

Baseline Systems. We compare Qafny^c's efficiency against a baseline of four state-of-the-art frameworks: the original Qafny prototype, SQIR, CoqQ and QBricks. We selected these systems because they're prominent, recently developed frameworks that represent two dominant alternative approaches to quantum program verifications. Specifically, SQIR and QBricks represent the circuit-based approach, which focuses on the formal semantics of low-level quantum gates, while CoqQ represents the measurement-based approach, which reasons about the probabilistic outcomes of measurement. Including the original Qafny prototype allows for a direct comparison to highlight the significant improvements in automation and applicability achieved by our compiler.

Table 2. Qafny^{c} Running time (including compilation (starting time 7s) and verification time in Dafny) & LOC in Qafny^{c} (not the LOC for the compiled Dafny code). Programs shown in blue were also verified by the original Qafny prototype. While the source LOCs are comparable for both, the Qafny's runtime is not reported due to its heavy reliance on human effort during verification.

Algorithm	Runtime	LOC	Algorithm	Runtime	LOC
Entanglement State Preparation					
Bell Pair	<10	13	Controlled GHZ	<10	8
GHZ	13	27			
Entanglement-based Network Communication					
Superdense Coding	12	21	Teleportation	13	25
Entanglement Swap	10	39			
State Discrimination & Testing					
State Distinguishing	24	46	Swap Test	25	51
Hadamard Testing Algorithm					
Deutsch-Jozsa	14	19	Simon's	16	26
Bernstein-Vazirani	12	22			
Hidden Subgroup Algorithms					
Hidden Subgroup	23	45	Quantum Phase Estimation (QPE)	22	45
Shor's algorithm	20	29			
Hidden Shift Algorithms					
Hidden Shift	22	43	Boolean Hidden Shift	24	42
Boolean Hidden Shift	24	42			
Amplitude Amplification & Counting					
Grover's search	15	27	Amplitude Amplification	24	39
Amp without Phase Estimation	29	57	Approximate Counting	19	30
Find Next K	27	29	Exact Counting	17	28
Conventional Amp	33	63			
Arithmetic & QFT-based Arithmetic					
QFT mod Q	34	69	QFT Adder	24	46
Modular Multiplication	31	59			
Quantum Walk & Linear Combination					
Quantum Walk	31	49	LCU	16	25
Compositional Algorithms					
Abelian Stabilizer	23	39	Amplitude Estimation	26	54
Fixed-point Oblivious Amp	26	45	Non-Boolean Amp	30	40
Fixed Point Search	28	32			
Repeat-Until-Success					
Hamming Weight	19	23	N-Basis-State Preparation	15	19
Brassard-Hoyer-Tapp	21	52			

By evaluating Qafny^{c} against these distinct methodologies, we can effectively analyze the trade-offs and advantages of our procedural, automated verification strategy.
Hardware and Implementation. The Qafny^{c} compiler is implemented in Python, using a visitor-pattern approach. We performed our evaluation on an Ubuntu computer with an 8-core 13th-gen Intel i9 processor and 16GB of DDR5 memory.

5.2 RQ1: Applicability and Scalibility

To address our first research question, we evaluated its performance on an extensive suite of 37 quantum algorithms and subroutines.

Applicability. The primary goal of the applicability study was to demonstrate that Qafny^{c} is expressive enough to formally specify and verify a large range of quantum programs common in the literature. Our benchmark was curated to include a diverse set of algorithms, successfully verifying programs from several key domains, including well-known quantum algorithms such as Shor's algorithm, Grover's search, and quantum phase estimation, as well as foundational primitives like Bell pairs or Teleportation. The successful verification of all 37 programs in Table 2, to our knowledge, represents the most extensive set of formally verified quantum programs to date and confirms the broad applicability of the Qafny^{c} framework.

Scalability. To assess scalability, we measured the verification time required for each program in our benchmark suite. Our Qafny^{c} compiler handles both parameterized and non-parameterized quantum programs, with 35 of the 37 evaluated algorithms parameterized by the number of qubits (n) to effectively demonstrate scalability; only Bell pair and superdense coding have a fixed size of 2. The verification time includes the entire Qafny^{c} compilation process: parsing the Qafny program, compiling it to Dafny, and the execution of the Dafny verifier. As we can see in Table 2, all 37 programs were implemented with fewer than 70 lines of Qafny code and were verified in under 35 seconds. The majority were verified in less than 23 seconds. This demonstrates the efficiency of the Qafny^{c} compiler and the effectiveness of leveraging Dafny's SMT-based solver, which avoids code execution and is not bottlenecked by the exponential size of quantum data. The performance scales well with the logical complexity of the program rather than the number of qubits, showcasing a scalable approach to quantum program verification.

Compared to Previous Qafny. The primary difference between our work and the previous Qafny proof system is the shift from a theoretical prototype to a fully automated compiler, as we discussed in Section 1.2. This automation is enabled by the key architectural changes, most notably the use of Dafny's immutable `sequence` to model quantum data (demonstrated in Section 3.3), which avoids the complex manual reasoning required by the original prototype's reliance on mutable `arrays`. The most direct measure of this improvement is the expansion of tractably verifiable programs. While the original system could handle 5 algorithms with significant manual intervention, our new compiler provides fully automated, "push-button" verification for all 37 diverse algorithms in our benchmark suite (Table 2). This addresses the practical bottlenecks of the previous approach. For complex quantum algorithms, the manual effort of writing correct state-update assertions and loop invariants becomes so error-prone and time-consuming that it forms a practical barrier to successful verification. Our two key technical advancements—the automated sequencing and generalized $\mathsf{EN}(n)$ type system—remove this barrier.

A decisive factor in this improvement is automated sequencing verification. While manual adjustment of the generated Dafny code is theoretically possible, in practice, the lack of automation for sequencing operations becomes a critical bottleneck. In Qafny^{c}, the compiling of $x[0,n) \leftarrow op_1(x[0,n))\ ;\ x[0,n) \leftarrow op_2(x[0,n))$ generates three arrays `x`, `x_i`, and `x_f`, to represent the states before and after op_1's application, as well as after op_2's application, where the state predicates are described via universal quantifiers over the array incides. Dafny fails to accumulate the transitive state transitions, where `x_f` is described by `x_i`, but not `x`. Qafny explicitly inserts assertions to build such a connection. In many cases, the assertions require manual adjustment for a successful connection establishment. In contrast, Qafny^{c} designs a data structure to track the state predicate transitions and generates additional predicates during the compilation of Dafny operations. In the above example, when compiling op_2 as a method, instead of generating a predicate to connect `x_f` and `x_i`, we generate the following two predicates. In the compiled Dafny loop for op_2, we insert the two predicates as parts of the loop invariants by replacing `bound` with the corresponding loop counter, and we replace `bound` with `|x|` in the `ensures` clauses in the method.

```
∀ j ::0 ≤ j < bound ⟹ x_i[j] == op1(x[j])
∀ j ::0 ≤ j < bound ⟹ x_f[j] == op2(op1(x[j]))
```

A second enhancement is the generalized $\mathsf{EN}(n)$-typed system, which are crucial for automating verification of complex algorithm like Shor's and QPE, where the inverse QFT operation turns an $\mathsf{EN}(1)$-typed datum to a $\mathsf{EN}(2)$-type. In our approach, each locus fragment is modeled as Dafny

Table 3. Qualitative Comparison of Verification Frameworks

Feature	**Qafny^c**	**SQIR/ QBricks**	**CoqQ**
Philosophy	Procedural	Foundational	Foundational
Reasoning Style	Syntactic (Type-based)	Semantic (Unitary Matrix / Path-Sum)	Semantic (Density Operator)

Table 4. Running time (include theory loading) & LOC for other platforms; -: no data.

(a) CoqQ data.

Algorithm	Runtime (sec)	LOC
Hidden Subgroup	137	280 + theories
QPE	121	210 + theories
Grover's	116	180 + theories

(b) QBricks and SQIR data.

Algorithm	QBricks		SQIR	
	Runtime (sec)	LOC	Runtime (sec)	LOC
GHZ	-	-	141	119
Deutsch-Jozsa	74	108	163	408
QPE	963	809	657	2141
Grover's search	253	233	148	1018
Shor's algorithm	1328	1163	1244	8464

type seq<seq<seq<bv1>>>. A measurement picks an element (seq<seq<bv1>>) in the outer sequence and computes the probability based on the sum of amplitudes associated with the element. Qafny adopts only EN(1) types, modeled as simple arrays, where measurement applications must search the array to collect the amplitudes associated with the measurement output basis-vector, making verification challenging.

5.3 RQ2: A Qualitative Comparison of Verification Approaches

A direct quantitative comparison of metrics like LOC and runtime with the baseline frameworks is challenging because they follow fundamentally different verification philosophies. As summarized in table 3, baseline frameworks such as SQIR, QBricks, and CoqQ adopt a fundational philosophy–emphasizing *depth* by reasoning semantically over formal mathematical models for high assurance proof. Qafny instead follows a procedural philosophy-emphasizing *breadth* by applying a syntactic, type-based reasoning to a wide range of program patterns with high automation. This section addresses our second research question qualitively, highlighting the trade-offs in automation and verification scope between these approaches.

Procedural vs Foundational Verification. Qafny^c takes a procedural approach to prove that a given program is a correct implementation of its specification, which focuses on automation and implementation correctness, enabling us to verify a broad suite of 37 diverse quantum programs.

By contrast, the baseline frameworks takes a foundational approach, typically within interactive theorem-proving systems (ITPs), such as Isabelle, Rocq, and Why3. Here, program properties are expressed as mathematical theorems, and users prove them step-by-step, often developing detailed proofs for the underlying mathematical theories. This depth comes at a cost: published results usually covers a smaller number of foundational algorithms, as reflected in the coarse results of table 4. The proof effort inherent in the foundational approach limits the number of verified examples.

This "verification scope" tradeoff can be illustrated with Shor's algorithm. Qafny^c does prove that a specific implementation of Shor's algorithm correctly builds the required quantum state for any symbolic input n. However, it does not prove the underlying mathematical theorems that guarantee the algorithm's success. Such properties include the period-finding property of the QFT, the number-theoretic link between finding a period and factoring, and the probabilistic guarantees of measuring a useful outcome. These deeper properties are the domain of foundational systems, whereas Qafny^c focuses on the crucial but distinct task of ensuring the program is a correct implementation of the algorithm.

Syntactic vs Semantic Reasoning. The differences in philosophy manifest in a technical distinction: Qafny^{c} performs syntactic reasoning, while the baseline frameworks perform semantic reasoning.

Semantic reasoning, as used in SQIR, QBricks and CoqQ, relies on uniform quantum state representations and inteprets programs through mathmetical semantics. For example, SQIR interprets every unitary quantum operation as a unitary matrix and uses tactics like `solve_matrix` and `gridify` to match the result matrix entry with the specification. QBricks utilizes the equational properties of path-sum states to perform state canonicalization, while CoqQ constructs a proof system based on the density matrix state representation, enabling users to connect measurement probabilities with program applications. All the above frameworks perform semantics-based program reasoning, requiring users to verify a program by building step transitions on special forms of quantum states.

In contrast, Qafny^{c} identifies different quantum state syntactic forms based on their use in quantum computation and classifies these forms as types. This allows users to choose different quantum state representations when developing their programs. Qafny^{c} also captures different program patterns and develops proof tactics based on their utility. Eventually, all the program patterns and state representations are formulated as Qafny^{c} syntactic components, which are embedded into Dafny for automated verification.

6 Discussion

Our Qafny^{c} compiler development is *test-driven*. We maintain a sample program set as the validation set for our compiler. For compiling each operation, we first wrote a simple Qafny program containing the operation. We then manually developed the Dafny programs for these simple programs and updated our Qafny^{c} compiler to compile them correctly. Finally, we validate the updated Qafny compiler against the entire validation set to ensure that all programs remain valid and verifiable in Dafny. This iterative process, which eventually included 28 simple programs in our validation set, ensured the correctness and robustness of the compiler before its application to the main benchmark suite.

A key goal of any verification framework is to be useful for developers. While a formal user study is beyond the scope of this work, we can analyze the design features of Qafny^{c} that support its practical utility in bug finding and providing useful feedback.

Bug Detection Metrics. We developed a benchmark suite of 37 quantum programs from scratch in Qafny. The verification process also demonstrates our bug detection mechanism, exposing two distinct categories of errors. The first, **frontend bugs**, includes implementation errors introduced while writing the Qafny code. We utilize the Qafny type system to catch these issues, which ranged from incorrect mappings of quantum gates to Qafny^{c}'s IR to other logical flaws in the program's construction. The second category, **algorithmic bugs**, comprises more subtle faults originating from the high-level description of an algorithm itself. In several cases, Qafny^{c} identifies issues such as missing preconditions or implicit assumptions that were not explicitly stated in the source literature, demonstrating its value beyond checking simple implementation mistakes.

Error Reporting and User Feedback. A core component of Qafny^{c}'s utility is its ability to translate verification results from Dafny into actionable feedback for the programmer. Without this, a user would receive cryptic errors referring to generated code rather than their own source.

Here, Qafny^{c} embeds location information from the source code file directly into its Abstract Syntax Tree (AST) during parsing. This metadata is propagated through the entire compilation pipeline. When the Dafny verifier reports an error, Qafny^{c} intercepts the message and uses the propagated location map to trace the error back to the line in the original Qafny source code.

Our system handles two primary types of verification failures. When Dafny finds a provable error, such as a violated postcondition, the raw error message is captured and re-contextualized with the corresponding Qafny line number, pointing the user directly to the source of the problem. For proof that stalls, Qafny^c uses a configurable timeout to terminate the process. It then informs the user of the timeout and reports the location in the Qafny code where the verifier was working, helping to isolate overly complex or potentially incorrect assertions.

Currently, Qafny^c operates in a batch mode, using Dafny's command-line interface to perform a single pass. It captures all outputs and formats them with the appropriate source mapping, rather than supporting the real-time interactive verification available in some IDEs.

7 Related Work

This section gives related work beyond the discussion in Section 5.3.

Quantum Verification Frameworks and Proof Systems. The circuit-based verification framework includes Qwire [Rand et al. 2017], SQIR and QBricks, with their difference with respect to Qafny^c given in Section 5.3. Many other works are measurement-based, including quantum Hoare logic [Feng and Ying 2021; Liu et al. 2019; Ying 2012, 2019], quantum separation logic (QSL) [Le et al. 2022; Zhou et al. 2021], quantum relational logic [Li and Unruh 2021; Unruh 2019], and probabilistic Hoare logic for quantum programs [Kakutani 2009], informing the Qafny^c development. The key difference is that Qafny^c is a compiler that compiles quantum program verification to Dafny. We compare a key measurement-based work, CoqQ, with Qafny^c in Section 5.3.

Qafny^c is built on top of the Qafny [Li et al. 2024], which provides the theoretical foundation for a proof system to verify quantum programs. Qafny^c realizes the proof system by constructing a robust compiler to translate quantum program verification to Dafny for automated verification.

Classical Proof Systems. We are inspired by separation logic, as articulated in Reynolds [2002], and others [Itzhaky et al. 2021; Löding et al. 2017; Neider et al. 2018; Sammler et al. 2021; Ta et al. 2016; Zhan 2018]. A sound and complete subset of the separation logic system was studied by Faisal Al Ameen and Tatsuta [2016]; Tatsuta et al. [2019]. The Qafny implementation is compiled to Dafny [Leino 2010], a language designed to simplify writing correct code. The natural proof methodology [Löding et al. 2017; Madhusudan et al. 2012; Pek et al. 2014] informs the Qafny development, where it embeds the proofs of data structures to a recursive search problem.

8 Conclusion and Future Work

We present Qafny^c, a system for expressing quantum programs based on Qafny and compiling the verification of the programs into Dafny. With the Qafny^c compiler, implemented in Python, we have successfully compiled 37 Qafny quantum programs for real-world quantum algorithms along with their specifications into Dafny for verification. All of these programs have been successfully verified in Dafny. We believe that Qafny offers a valuable tool for quantum program development and automated verification, enabling programmers to both create and verify quantum programs efficiently. In the future, we plan to utilize Qafny^c to develop many different kinds of quantum programs with verification, so that most on-paper quantum algorithms can be turned into programs, which can be correctly compiled to quantum circuits via our circuit compiler.

9 Data-Availability Statement

The software mentioned in Section 3 and supporting Section 5 is available at https://github.com/qafny/qafny_impl. The artifact is produced in the described experimental setting.

Acknowledgments

We thank Mingwei Zhu for his helpful comments and contributions during the work's development.

References

Gilles Brassard, Peter Høyer, Michele Mosca, and Alain Tapp. 2002. Quantum amplitude amplification and estimation. 53–74 pages. doi:10.1090/conm/305/05215

Christophe Chareton, Sébastien Bardin, François Bobot, Valentin Perrelle, and Benoît Valiron. 2021. An Automated Deductive Verification Framework for Circuit-building Quantum Programs. In *Programming Languages and Systems - 30th European Symposium on Programming, ESOP 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 12648)*, Nobuko Yoshida (Ed.). Springer, 148–177. doi:10.1007/978-3-030-72019-3_6

Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. 2009. VCC: A Practical System for Verifying Concurrent C. In *Theorem Proving in Higher Order Logics*, Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 23–42.

Andrew W. Cross, Lev S. Bishop, John A. Smolin, and Jay M. Gambetta. 2017. Open quantum assembly language. *arXiv e-prints* (Jul 2017). arXiv:1707.03429 [quant-ph]

Mahmudul Faisal Al Ameen and Makoto Tatsuta. 2016. Completeness for recursive procedures in separation logic. *Theoretical Computer Science* 631 (2016), 73–96. doi:10.1016/j.tcs.2016.04.004

Yuan Feng and Mingsheng Ying. 2021. Quantum Hoare Logic with Classical Variables. *ACM Transactions on Quantum Computing* 2, 4, Article 16 (dec 2021), 43 pages. doi:10.1145/3456877

Sukhpal Singh Gill, Oktay Cetinkaya, Stefano Marrone, Daniel Claudino, David Haunschild, Leon Schlote, Huaming Wu, Carlo Ottaviani, Xiaoyuan Liu, Sree Pragna Machupalli, Kamalpreet Kaur, Priyansh Arora, Ji Liu, Ahmed Farouk, Houbing Herbert Song, Steve Uhlig, and Kotagiri Ramamohanarao. 2024. Quantum Computing: Vision and Challenges. arXiv:2403.02240 [cs.DC] https://arxiv.org/abs/2403.02240

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* (Philadelphia, Pennsylvania, USA) *(STOC '96)*. Association for Computing Machinery, New York, NY, USA, 212–219. arXiv:quant-ph/9605043 doi:10.1145/237814.237866

Kesha Hietala, Robert Rand, Shih-Han Hung, Liyi Li, and Michael Hicks. 2021a. Proving Quantum Programs Correct. In *Proceedings of the Conference on Interative Theorem Proving (ITP)*.

Kesha Hietala, Robert Rand, Shih-Han Hung, Xiaodi Wu, and Michael Hicks. 2021b. A Verified Optimizer for Quantum Circuits. In *Proceedings of the ACM Conference on Principles of Programming Languages (POPL)*.

C. A. R. Hoare. 1969. An axiomatic basis for computer programming. *Commun. ACM* 12, 10 (Oct. 1969), 576–580. doi:10.1145/363235.363259

Shachar Itzhaky, Hila Peleg, Nadia Polikarpova, Reuben N. S. Rowe, and Ilya Sergey. 2021. Cyclic Program Synthesis. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 944–959. doi:10.1145/3453483.3454087

Yoshihiko Kakutani. 2009. A Logic for Formal Verification of Quantum Programs. In *Advances in Computer Science - ASIAN 2009. Information Security and Privacy*, Anupam Datta (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 79–93.

Xuan-Bach Le, Shang-Wei Lin, Jun Sun, and David Sanan. 2022. A Quantum Interpretation of Separating Conjunction for Local Reasoning of Quantum Programs Based on Separation Logic. *Proc. ACM Program. Lang.* 6, POPL, Article 36 (jan 2022), 27 pages. doi:10.1145/3498697

K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning*, Edmund M. Clarke and Andrei Voronkov (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 348–370.

Liyi Li, Finn Voichick, Kesha Hietala, Yuxiang Peng, Xiaodi Wu, and Michael Hicks. 2022. Verified Compilation of Quantum Oracles. In *OOPSLA 2022*. doi:10.48550/ARXIV.2112.06700

Liyi Li, Mingwei Zhu, Rance Cleaveland, Alexander Nicolellis, Yi Lee, Le Chang, and Xiaodi Wu. 2024. Qafny: A Quantum-Program Verifier. In *38th European Conference on Object-Oriented Programming (ECOOP 2024) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 313)*, Jonathan Aldrich and Guido Salvaneschi (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 24:1–24:31. doi:10.4230/LIPIcs.ECOOP.2024.24

Yangjia Li and Dominique Unruh. 2021. Quantum Relational Hoare Logic with Expectations. In *48th International Colloquium on Automata, Languages, and Programming (ICALP 2021) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 198)*, Nikhil Bansal, Emanuela Merelli, and James Worrell (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 136:1–136:20. doi:10.4230/LIPIcs.ICALP.2021.136

Junyi Liu, Bohua Zhan, Shuling Wang, Shenggang Ying, Tao Liu, Yangjia Li, Mingsheng Ying, and Naijun Zhan. 2019. Formal Verification of Quantum Algorithms Using Quantum Hoare Logic. In *Computer Aided Verification*, Isil Dillig and Serdar Tasiran (Eds.). Springer International Publishing, Cham, 187–207.

Christof Löding, P. Madhusudan, and Lucas Peña. 2017. Foundations for Natural Proofs and Quantifier Instantiation. *Proc. ACM Program. Lang.* 2, POPL, Article 10 (dec 2017), 30 pages. doi:10.1145/3158098

Parthasarathy Madhusudan, Xiaokang Qiu, and Andrei Stefanescu. 2012. Recursive Proofs for Inductive Tree Data-Structures. *SIGPLAN Not.* 47, 1 (jan 2012), 123–136. doi:10.1145/2103621.2103673

Narciso Martí-Oliet and José Meseguer. 2000. Rewriting logic as a logical and semantic framework. In *Electronic Notes in Theoretical Computer Science*, J. Meseguer (Ed.), Vol. 4. Elsevier Science Publishers.

Daniel Neider, Pranav Garg, P. Madhusudan, Shambwaditya Saha, and Daejun Park. 2018. Invariant Synthesis for Incomplete Verification Engines. In *Tools and Algorithms for the Construction and Analysis of Systems*, Dirk Beyer and Marieke Huisman (Eds.). Springer International Publishing, Cham, 232–250.

Edgar Pek, Xiaokang Qiu, and P. Madhusudan. 2014. Natural Proofs for Data Structure Manipulation in C Using Separation Logic. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. Association for Computing Machinery, New York, NY, USA, 440–451. doi:10.1145/2594291.2594325

Xiaokang Qiu, Pranav Garg, Andrei Ştefănescu, and Parthasarathy Madhusudan. 2013. Natural Proofs for Structure, Data, and Separation. *SIGPLAN Not.* 48, 6 (jun 2013), 231–242. doi:10.1145/2499370.2462169

Robert Rand, Jennifer Paykin, and Steve Zdancewic. 2017. QWIRE practice: Formal verification of quantum circuits in Coq. In *Proceedings 14th International Conference on Quantum Physics and Logic, QPL 2017, Nijmegen, The Netherlands, 3-7 July 2017.* 119–132. doi:10.4204/EPTCS.266.8

J.C. Reynolds. 2002. Separation logic: a logic for shared mutable data structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science.* 55–74. doi:10.1109/LICS.2002.1029817

Grigore Roşu and Andrei Ştefănescu. 2011. Matching Logic: A New Program Verification Approach (NIER Track). In *ICSE'11: Proceedings of the 30th International Conference on Software Engineering.* ACM, 868–871. doi:doi:10.1145/1985793.1985928

Grigore Roşu, Andrei Ştefănescu, Ştefan Ciobâcă, and Brandon M. Moore. 2013. One-Path Reachability Logic. In *Proceedings of the 28th Symposium on Logic in Computer Science (LICS'13).* IEEE, 358–367.

Michael Sammler, Rodolphe Lepigre, Robbert Krebbers, Kayvan Memarian, Derek Dreyer, and Deepak Garg. 2021. RefinedC: Automating the Foundational Verification of C Code with Refined Ownership Types. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021).* Association for Computing Machinery, New York, NY, USA, 158–174. doi:10.1145/3453483.3454036

Matt Swayne. 2023. *What Are The Remaining Challenges Of Quantum Computing?* https://thequantuminsider.com/2023/03/24/quantum-computing-challenges/

Quang-Trung Ta, Ton Chanh Le, Siau-Cheng Khoo, and Wei-Ngan Chin. 2016. Automated Mutual Explicit Induction Proof in Separation Logic. doi:10.48550/ARXIV.1609.00919

Makoto Tatsuta, Wei-Ngan Chin, and Mahmudul Faisal Al Ameen. 2019. Completeness and expressiveness of pointer program verification by separation logic. *Information and Computation* 267 (2019), 1–27. doi:10.1016/j.ic.2019.03.002

Dominique Unruh. 2019. Quantum Relational Hoare Logic. *Proc. ACM Program. Lang.* 3, POPL, Article 33 (jan 2019), 31 pages. doi:10.1145/3290346

Mingsheng Ying. 2012. Floyd–Hoare Logic for Quantum Programs. *ACM Trans. Program. Lang. Syst.* 33, 6, Article 19 (Jan. 2012), 49 pages. doi:10.1145/2049706.2049708

Mingsheng Ying. 2019. Toward Automatic Verification of Quantum Programs. *Form. Asp. Comput.* 31, 1 (feb 2019), 3–25. doi:10.1007/s00165-018-0465-3

Bohua Zhan. 2018. Efficient Verification of Imperative Programs Using Auto2. In *Tools and Algorithms for the Construction and Analysis of Systems*, Dirk Beyer and Marieke Huisman (Eds.). Springer International Publishing, Cham, 23–40.

Li Zhou, Gilles Barthe, Justin Hsu, Mingsheng Ying, and Nengkun Yu. 2021. A Quantum Interpretation of Bunched Logic and Quantum Separation Logic. In *Proceedings of the 36th Annual ACM/IEEE Symposium on Logic in Computer Science* (Rome, Italy) *(LICS '21).* Association for Computing Machinery, New York, NY, USA, Article 75, 14 pages. doi:10.1109/LICS52264.2021.9470673

Received 2025-03-25; accepted 2025-08-12