Syntax-guided Synthesis of Quantum Unitary Programs

ANONYMOUS AUTHOR(S)

Quantum programs are notoriously difficult to code and verify due to unintuitive quantum knowledge associated with quantum programming. Automated tools relieving the tedium and errors associated with low-level quantum details would hence be highly desirable. In this paper, we initiate the study of *syntax-guided program synthesis* for quantum unitary programs that recursively define a family of unitary circuits for different input sizes, which are widely used in existing quantum programming languages. Specifically, we present QSynth, a quantum program synthesis framework, including a new inductive quantum programming language, its specification, a sound logic for reasoning, and an encoding of the reasoning procedure into SMT instances. By leveraging existing SMT solvers, QSynth successfully synthesizes quantum unitary programs for state preparation, quantum adder circuits, and quantum Fourier transformation, which can be readily transpiled to executable programs on major quantum platforms, e.g., Q#, IBM Qiskit, and AWS Braket.

Additional Key Words and Phrases: Quantum Programs, Syntax-guided Program Synthesis, SMT solvers

1 INTRODUCTION

Automated program synthesis has been a dream target for computer scientists since the beginning of programming. The ability to mechanically construct programs would significantly ease the task of programming in dealing with the low-level details, and hence leave the human programmers to focus on the high-level design of the system.

A significant progress on classical program synthesis has been made in the recent decades [25, 39], with many diverse and exciting developments such as syntax-guided synthesis [1, 2, 31, 33], example-guided synthesis [19, 22, 23, 32, 35, 50], semantics-guided synthesis [16, 36], type-driven synthesis [26, 27, 32, 49], and resource-guided synthesis [30, 40]. The modern approaches to solve these problems make use of sophisticated search algorithms, such as enumerative search with pruning [24, 48], constraint solver like satisfiability modulo theory (SMT) solver [18, 33, 55], and machine learning [42, 44].

Quantum programming is one specific type of programming that would benefit significantly from the study of program synthesis. Dealing with the low-level details in quantum programming requires a lot of domain knowledge in quantum computing, and could be a tedious and erroneous procedure even for quantum experts. Quantum program synthesis would help quantum programmers stay with the more intuitive high-level design of quantum applications, and hence reduce the barriers for newcomers to quantum computing. Technically, quantum program synthesis naturally requires the development of high-level specifications of quantum programs, which is an important direction to explore given most existing quantum programming languages are still using the low-level circuit abstraction.

A simplified case of quantum program synthesis, without involving any loop or recursive program structures, is conventionally known as the circuit synthesis in the literature of quantum computing. The task of generating a quantum circuit given a target unitary has been intensively studied in the past [6, 15, 38, 52, 53, 63], either through algorithms induced by analytical formulas like the Solovay-Kitaev theorem [38] or through clever enumerative search with pruning like QFAST [63]. These synthesis tasks typically deal with a fixed

2022. XXXX-XXXX/2022/11-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

(and usually small) number of qubits and cannot handle arbitrary input size, e.g., QFAST only synthesizes QFT circuits up to 5 qubits, and adders up to 4 qubits.

The size of quantum circuits associated with realistic quantum applications usually varies with different inputs. It is hence desirable to represent such a family of circuits by a succinct quantum program that produces the desired circuit when given a specific input. Such programs are ubiquitous in the code base of all major commercial quantum platforms, and are typically written and maintained manually by human programmers.

We deem these programs as more natural and important targets for quantum program synthesis and focus on synthesizing them in this paper. To that end, we first formulate these instances as *quantum unitary programs that recursively defines a family of quantum circuits without measurements (i.e., unitaries)*, which is modelled after the recursive structure used in Q# and Python. Precisely, we define an Inductive extension of the Simple Quantum Intermediate Representation (SQIR) [28] called ISQIR, which is the target language of synthesis. Any ISQIR program can be easily converted to Q# or Qiskit/Braket in Python.

Technical Challenges. Although our synthesis approach shares a lot of similarities with classical ones, especially on the general methodology and the search procedure of candidate programs, we want to highlight a few unique features and challenges in the quantum setting. First, note that our target ISQIR program involves a varying number of qubits that depends on the input, which already corresponds to more challenging synthesis cases in the classical setting, e.g., programs involving arrays [56]. An immediate challenge is the need that specifications for ISQIR program should also depend on the input size, which renders the natural candidate for specifications on quantum programs, i.e., quantum Hoare triple [62], inapplicable as they are defined on fixed-dimension quantum systems. Instead, we develop an arguably more flexible and intuitive specification called the *hypothesis-amplitude* specification, which could vary on different inputs and could be in general as expressive as quantum Hoare logic on any fixed dimension.

A bigger challenge comes from the scalability of these specifications since the underlying Hilbert space associated with quantum programs is exponentially large in terms of the system size. The full expressiveness of the hypothesis-amplitude specification will hence significantly restrict its scalability, especially when matching the expressive power of quantum Hoare logic. Thus, *successful quantum program synthesis would need to balance the expressivenss and the scalability of the underlying specification.*

The third challenge arises when one automates the reasoning based on the hypothesis-amplitude specification and its associated logic by using SMT solvers. One major difficulty is the lack of support to verify the equivalence between two general *complex* functions, which is however typical in the quantum setting. Another major difficulty is the existence of many cases where naturally encoded SMT instances from the specification would contain an *unbounded* number of terms. To obtain efficient SMT encodings, we restrict ourselves to sparse *parameterized path-sum amplitudes* (defined in Section 5), which, however, turn out to be expressive enough to specify and synthesize many quantum programs.

Contributions. We present QSynth, a quantum program synthesis framework, an overview and case illustration of which is in Section 2. To that end, our contributions are multi-folded:

- We develop the syntax and the semantics of the inductive SQIR (ISQIR) in Section 4.1.

- We design the hypothesis-amplitude specification and its associated logic (called the unitary ISQIR logic) in Section 4.2.
- We identify sufficient conditions for which efficient encoding of the verification process into SMT instances exists in Section 5.
- We showcase the synthesis of more practical quantum programs like quantum adder circuits [13, 20] and quantum Fourier transformation [12] by QSynth in Section 6.

Related Work. We refer curious readers to surveys [25, 39] for a comprehensive picture on the development of classical program synthesis techniques. Many synthesis frameworks are developed into productive tools. For example, the Sketch [55] framework completes programs with holes by specifications. Rosette [58] builds solvers into the language to automatically fill in holes when programming. Synthesis of probabilistic programs [10, 46, 51] are also similar tasks, since probabilistic programs are special cases of quantum programs. However, QSynth needs to deal with unique challenges from quantum programs.

An important procedure in syntax-guided synthesis is to verify any candidate program. Various logic and verification tools for quantum programs are developed in the last decade. For example, quantum Hoare logic [62] uses quantum predicates and Hoare triples to express and derive properties of quantum programs. Quantum abstract interpretation [64] provides efficient tools to test properties of quantum programs. In particular, the path-sum representation [3] of quantum program semantics inspired our representation. The use of SMT solvers to automate the reasoning has also appeared in QBricks [11], Giallar [57] and Quartz [61], although they didn't deal with unique challenges in our case.

2 SYNTHESIS OVERVIEW

2.1 Oracle-Guided Inductive Synthesis

Following Srivastava et al. [56], we formulate the synthesis problem as an *Oracle-Guided Inductive Synthesis* (OGIS) [34] problem. OGIS comprises three key components: a *specification* for the target program, *an inductive learning engine* and a *correctness oracle*. Synthesis is an iterative process: at each step, the learning engine formulates and sends a candidate program to the oracle, and the oracle sends its response including whether the learner has found a program that satisfies the *specification*. This process continues until the learning engine finds a correct program or reaches some predefined maximum time.

2.2 QSynth Overview

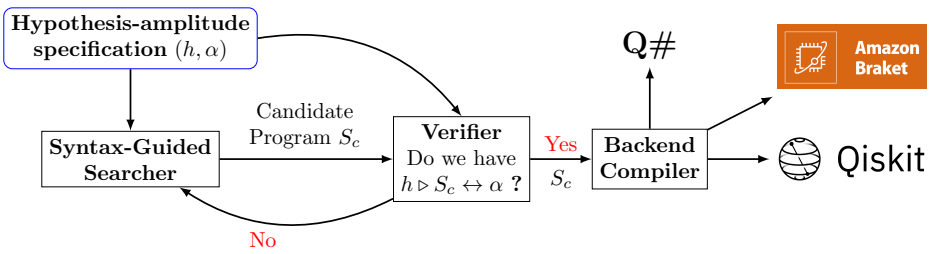
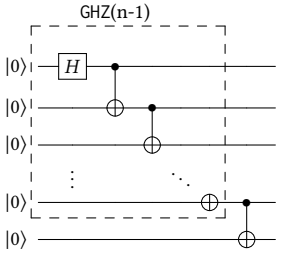


Fig. 1. QSynth Overview.

$$S_{GHZ} := \textbf{fix}_1\ Id$$
$$\quad\ \textbf{const}\ \texttt{H 0}$$
$$\quad\ \textbf{const}\ \texttt{ID}$$
$$\quad\ \textbf{const}\ \texttt{CNOT n-1 n}$$

(a) ISQIR program



(b) $(n+1)$-qubit GHZ circuit

```
def GHZ(N):
    circuit=QuantumCircuit(N+1)
    def S(circ, n):
        if n==0:
            circ.h(0)
        else:
            S(circ, n-1)
            circ.cx(n-1, n)
    S(circuit,N)
    return circuit
```

(c) Qiskit program

```
Operation GHZ(q : Qubit[], n : Int) :
    Unit{
    if (n==0){
        H(q[0]);
        return ();
    }
    GHZ(q,n-1);
    CNOT(q[n-1],q[n]);
}
```

(d) Q# program

Fig. 2. GHZ state preparation programs in different quantum programming languages. (a) ISQIR program generating a circuit for preparing $(n+1)$-qubit GHZ state. (b) The quantum circuit represented by the ISQIR program S_{GHZ}. (c) Qiskit function `GHZ` compiled from ISQIR program S_{GHZ}. Statement `circ.cx` in Qiskit means appending a `CNOT` gate to the circuit `circ`. (d) Q# GHZ program compiled from ISQIR program S_{GHZ}.

The workflow of QSynth is shown in Fig 1. We have developed the so-called *hypothesis-amplitude specification* (h, α) to specify desired properties of the target ISQIR programs. We employ the standard Syntax-Guided Top-down Tree Search [25] to enumerate candidate ISQIR programs. No specific search modification is employed, although optimizing the search procedure for quantum programs is an interesting direction for future work.

A key component of QSynth is the correctness oracle to verify the candidate program S_c with respect to the desired (h, α) specification, denoted $h \triangleright S_c \leftrightarrow \alpha$. This is made possible based on a newly developed *unitary inductive SQIR logic* as well as an SMT encoding scheme to automatically verify $h \triangleright S_c \leftrightarrow \alpha$ with SMT solvers.

Once a correct target program is returned, our backend compiler could further transpile this ISQIR program into commercial quantum programming languages like Q#, Qiskit, and Braket. For example, Fig 2(c) shows a Qiskit program compiled from the $(n+1)$-qubit GHZ ISQIR program S in Fig 2(a)[1], and similarly a Q# program in Fig 2(d).

2.3 Challenges and Solutions via the GHZ example

We will elaborate on the aforementioned technical challenges and our solutions by walking through the complete synthesis procedure for the GHZ example. We assume some basic quantum notation and refer readers to Section 3 for a more detailed preliminary.

[1]Qiskit's semantic requires the program always use class `QuantumCircuit(n)` to initialize a circuits with fixed qubits count n. So we wrap program S with an outside function `GHZ` which calls `QuantumCircuit` to initialize the circuit. Function `GHZ`(N) returns a Qiskit circuit that generates $(N+1)$-qubit GHZ state.

Target Program. An N-qubit ($N \in Z^+$) Greenberger-Horne-Zeilinger state (GHZ state) is an entangled quantum state given by

$$|GHZ\rangle_N = \frac{1}{\sqrt{2}}(|0\rangle^{\otimes N} + |1\rangle^{\otimes N}),$$

which was first studied by Greenberger et al. [21] and is widely used in quantum information, e.g., [29, 43, 60, 65]. Preparing the N-qubit GHZ state from $|0\rangle^{\otimes N}$ for any N is a natural task for program synthesis. Namely, we hope to synthesis an ISQIR program S such that $[\![\{\!\{S\}\!\}(n)]\!]$ is a unitary that transfers state $|0\rangle^{n+1}$ to state $|GHZ\rangle_{n+1}$ for any $n \geq 0$.

Target Specification. Our first challenge is how to specify our target program, which generates $|GHZ\rangle_{n+1}$ from $|0\rangle^{n+1}$. The difficulty is two-folded: (1) we want a specification for any input n, which excludes any existing specification methods for fixed dimensions (e.g., Quantum Hoare triples); (2) the specification should be as succinct as possible. For the latter, a direct matrix representation that might require $2^{n+1} \times 2^{n+1}$ is less desirable. To that end, we develop the following *hypothesis-amplitude* (h, α) *specification* (Definition 4.4), an instantiation of which to the GHZ target program is given as follows:

$$h := \{(n, x, y) \mid x = 0\}, \qquad \alpha_{GHZ}(n, x, y) := \frac{1}{\sqrt{2}}\delta(y = 0 \vee y = 2^{n+1} - 1), \tag{2.1}$$

where the term $\delta(b)$ in α_{GHZ} is a {0,1}-valued function that returns 1 if the Boolean expression b is True and 0 otherwise.

Intuitively, the hypothesis function h specifies the interesting input to the program, the desired program's behaviour on which is specified by the amplitude function α. Precisely, given input x, the amplitude $\langle y|[\![\{\!\{S\}\!\}(n)]\!]|x\rangle$ of basis y on the input x for a desired program S is given by $\alpha(n, x, y)$, where x, y are bit strings.

In the GHZ example, we are only interested in input $|0\rangle^{n+1}$, which leads to a trivial h containing only $x = 0$ in (2.1). The output state, which is $\frac{1}{\sqrt{2}}(|0\rangle^{n+1} + |1\rangle^{n+1})$, corresponds to an amplitude function $\alpha(n, x, y)$ with only non-zero value $\frac{1}{\sqrt{2}}$ on either $y = 0$ (referring to $|0\rangle^{n+1}$) or $y = 2^{n+1} - 1$ (referring to $|1\rangle^{n+1}$), which explains (2.1).

Our hypothesis-amplitude specification is arguably as natural as the common classical specifications that describe the desired input-output relationship, except that one could have many such input-output pairs (i.e., *superposition*) with potential complex amplitudes, in the quantum setting, which requires an explicit use of our amplitude function $\alpha(n, x, y)$.

Verification of Candidate Programs. Assume the searcher outlined in Section 2.2 has identified a candidate S_{GHZ}, same as Fig 2 (a), on the left of Fig 3. The program S_{GHZ} is constructed by a FIX syntax with subprograms S_0, S_L, S_R, which is a recursive program with one base case S_0 (i.e., $S_{GHZ}(0)$). For the inductive case, $S_{GHZ}(n)$ consists of $S_L(n)$, $S_{GHZ}(n-1), S_R(n)$ sequentially. The FIX syntax, similar to the fixpoint in Coq, is introduced to enable an inductive structure in ISQIR programs.

QSynth verifier leverages the newly develop unitary ISQIR logic (Section 4.2) to verify the goal judgement $h \triangleright S_{GHZ} \leftrightarrow \alpha_{GHZ}$, which basically states that candidate program S_{GHZ} satisfies the (h, α) specification. QSynth verifier recursively applies the logic rules to split the judgement of h, α for larger programs into that of smaller programs. The side

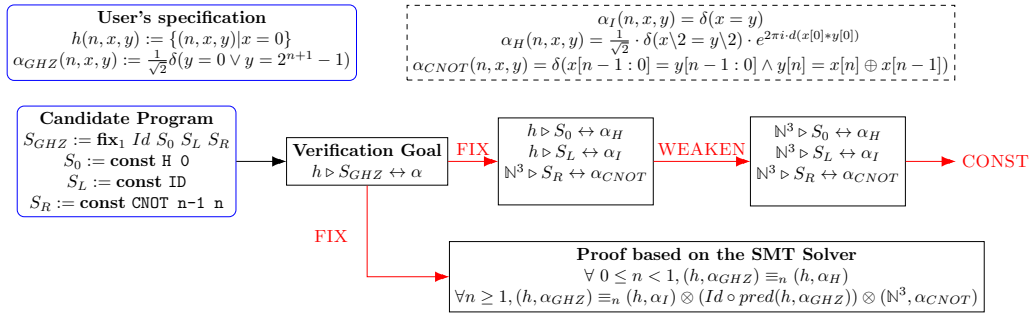


Fig. 3. An example of synthesizing $n+1$-qubit GHZ state preparation program.

conditions are checked by SMT solvers, and the h, α judgement for constant SQIR program for quantum gates that are independent of n are directly computed.

In the GHZ example, QSynth verifier first uses the FIX rule to split the goal judgement into two parts. The first part is two formulas (right bottom of Fig 3) to be checked by the SMT solver, with details elaborated on later. The second part is three judgements for S_L, S_R, S_0 (right hand of the verification goal box in Fig 3). QSynth uses the WEAKEN rule to adjust these judgements into an appropriate format. Since S_0, S_L, S_R are simple programs (formally called const SQIR programs), their corresponding hypothesis-amplitude specifications (i.e. $\alpha_I, \alpha_H, \alpha_{CNOT}$ on the upper right corner of Fig 3)[2] are predefined in QSynth and can be verifed directly using the CONST rule.

After this verification step, the synthesis terminates and QSynth sends S_{GHZ} to the backend compiler, which compiles S_{GHZ} into programs as shown in Fig 2.

Proof based on SMT solvers. We continue with the two formulas in the bottom right corner box in Fig 3 generated by the FIX rule and aim to verify them by SMT solvers.

The first formula indicates the *base* case is correct. The relation $\equiv_n$ indicates the equivalence between two hypothesis-amplitude specifications given specified n. In this GHZ example, the first formula becomes:

$$\forall x\ y \in \mathbb{N}, x = 0 \longrightarrow \alpha_{GHZ}(0,x,y) = \alpha_H(0,x,y).$$

Note that the amplitude functions α_{GHZ} or α_H are generally complex-valued, the automatic equivalence check of which are not generally supported by any existing tool.

Inspired by recent work on automated quantum program verification [3, 11], QSynth restricts α into a succinct path-sum representation yet with rich enough expressiveness (elaborated on in Section 5.1.2), called the parameterized path-sum amplitude (PPSA) in Definition 5.1. In PPSA representation, the non-zero amplitudes over x, y will share the same magnitude, which is a function depending on n, but could have different phases. Thus, instead of representing a general amplitude function $\alpha(n,x,y)$, it suffices to represent

[2] α_I returns 1 if $x = y$ else 0, indicating the identity matrix. α_H represents the matrix of the program **const** H 0, i.e., $\frac{1}{\sqrt{2}}\begin{pmatrix}1 & 1\\ 1 & -1\end{pmatrix} \otimes I^n$. The expression $e^{2\pi i \cdot \frac{x[0]*y[0]}{2}}$ in α_H returns -1 if $x[0] = 1, y[0] = 1$ and returns 1 otherwise, indicating the matrix $\frac{1}{\sqrt{2}}\begin{pmatrix}1 & 1\\ 1 & -1\end{pmatrix}$; the expression $\delta(x\backslash 2 = y\backslash 2)$ returns 1 if x, y only have the lowest bit difference and 0 otherwise, indicating the matrix I^n on qubits $q_1, ..., q_n$. α_{CNOT} will be discussed later.

$\alpha(n, x, y)$ by components. For instance, α_H, α_{GHZ} defined in Fig 3, can be described with three components:

- A fraction expression indicating the amplitude: $\frac{1}{\sqrt{2}}$ for both α_{GHZ}, α_H.
- A Boolean expression indicating the non-zero value: $\delta(y = 0 \vee y = 2^n - 1)$ for α_{GHZ} and $\delta(x\backslash 2 = y\backslash 2)$ for α_H where $x\backslash 2$ and $y\backslash 2$ are integer division (e.g. $10\backslash 3 = 3$).
- An expression indicating the phase: $e^{2\pi i \cdot 0}$ for α_{GHZ} and $e^{2\pi i \cdot \frac{x[0]*y[0]}{2}}$ for α_H, where $x[0]$ means the lowest (i.e. 0th) bit of x's binary representation (e.g. $6[0] = (110)_b = 0$).

A direct substitution of α_H, α_{GHZ} would require QSynth, or SMT solvers, to verify

$$\forall x\ y \in \mathbb{N}, x = 0 \rightarrow \frac{1}{\sqrt{2}}\delta(y = 0 \vee y = 1) = \frac{1}{\sqrt{2}}\delta(x\backslash 2 = y\backslash 2) \cdot e^{2\pi i \cdot \frac{x[0]*y[0]}{2}},$$

which is infeasible. Using PPSA, one can equivalently verify the following by SMT solvers:

$$\forall x\ y \in \mathbb{N}, x = 0 \rightarrow \frac{1}{\sqrt{2}} = \frac{1}{\sqrt{2}} \wedge \delta(y = 0 \vee y = 1) = \delta(x\backslash 2 = y\backslash 2) \wedge \frac{x[0] * y[0]}{2} = 0.$$

The second formula concerns the correctness of the *induction* case. The function $pred(h, \alpha_{GHZ})$ generates the hypothesis-amplitude specification for the recursive call (i.e. $S_{GHZ}(n-1)$), and $(h_1, \alpha_1) \otimes (h_2, \alpha_2)$ is used to calculate the hypothesis-amplitude specifications when composing two ISQIR programs together, both formally in Definition 4.5).

In this GHZ example, the composition with (h, α_I) is trivial since α_I represents an identity matrix, and the second formula becomes

$$\forall n \geq 1, \forall x\ y \in \mathbb{N}, (h, \alpha_{GHZ}) \equiv_n (h, \alpha')$$
$$\alpha'(n, x, y) = \sum_{z \in \mathbb{N}} \alpha_{GHZ}(n-1, x, z)\alpha_{CNOT}(n, z, y) \tag{2.2}$$
$$\alpha_{CNOT}(n, x, y) = \delta(x[n-1:0] = y[n-1:0] \wedge y[n] = x[n] \oplus x[n-1])$$

α_{CNOT} is designed to represents S_R: the term $x[n-1:0] = y[n-1:0]$ indicates the identity map $|q_0 ... q_{n-1}\rangle \mapsto |q_0 ... q_{n-1}\rangle$; the term $y[n] = x[n] \oplus x[n-1]$ indicates the map $|q_{n-1}\rangle|q_n\rangle \mapsto |q_{n-1}\rangle|q_{n-1} \oplus q_n\rangle$. Their combination indicates S_R's map, i.e., $|q_0 ... q_{n-1}\rangle|q_n\rangle \mapsto |q_0 ... q_{n-1}\rangle|q_{n-1} \oplus q_n\rangle$.

There is another major challenge to verify (2.2) by SMT solvers due to the infinite summation over $z \in \mathbb{N}$, which comes from the composition of two amplitude specifications (details in Section 4). As a result, α' could have an unbounded number of terms, which makes it infeasible for any SMT solver.

To circumvent this general difficulty, we introduce a *sparsity* constraint, which restricts the number of non-zero points with any fixed x or y to be constant (Definition 5.2), and prove that the composition of two amplitude functions will have finite terms if *one of* the composed function is sparse. We also prove that all quantum gates applied on a constant number of qubits (e.g. all SQIR programs) have a sparse amplitude function. However, the use of the FIX statement could generate non-sparse amplitude functions.

As a result, we only allow one use of the FIX statement in our synthesis, as otherwise we could risk composing two non-sparse amplitude functions that would lead to an infinite sum. The specification for the target program, however, could be non-sparse, as we won't need to compose the target programs with others.

In the GHZ example, α_{CNOT} is sparse and we have

$$\forall n, y \in \mathbb{N}, \quad \alpha_{CNOT}(n, z, y) \neq 0 \leftrightarrow z = y \oplus (y[n-1] \ll n)$$

Here the expression $y \oplus (y[n-1] \ll n)$ sets the $n+1$th bit of y (i.e. $y[n]$) to $y[n] \oplus y[n-1]$ and keeps $y[n-1:0]$ unchanged. Let $z = y \oplus (y[n-1] \ll n)$. With this sparsity of α_{CNOT}, we can simplify the formula (2.2) to

$$\forall n \geq 1, \forall x\ y \in \mathbb{N}, (h, \alpha_{GHZ}) \equiv_n (h, \alpha'),$$
$$\text{where } \alpha'(n, x, y) = \alpha_{GHZ}(n-1, x, z)\alpha_{CNOT}(n, z, y) = \frac{1}{\sqrt{2}}\delta(z = 0 \vee z = 2^n - 1).$$

With the PPSA representation, it suffices to verify the following SMT instance:

$$\forall n \geq 1, \forall x\ y \in \mathbb{N}, x = 0 \rightarrow \frac{1}{\sqrt{2}} = \frac{1}{\sqrt{2}} \ \wedge \ \delta(y = 0 \vee y = 2^{n+1} - 1) = \delta(z = 0 \vee z = 2^n - 1).$$

2.4 Characterization of QSynth's Search Space

In principle, QSynth would search any ISQIR program allowed by its syntax. The difficulty in automatic verification of candidate programs leads to two additional restrictions: (1) the one use of the FIX syntax; and (2) the target specification needs to follow the PPSA representation. Note that synthesis with nested-loops or fixed-point structures is also challenging for classical programs, due to the involvement of loop variants. We leave relieving these restrictions as an interesting open question for future exploration.

3 QUANTUM PRELIMINARIES

3.1 Quantum States

A quantum state consists of one or more *quantum bits*. A quantum bit (or *qubit*) can be expressed as a two dimensional vector $\binom{\alpha}{\beta}$ such that $|\alpha|^2 + |\beta|^2 = 1$. The α and β are called *amplitudes*. We frequently write this vector as $\alpha|0\rangle + \beta|1\rangle$ where $|0\rangle = \binom{1}{0}$ and $|1\rangle = \binom{0}{1}$ are *basis states*. A state written $|\phi\rangle$ is called a ket, following Dirac's notation. When both α and β are non-zero, we can think of the qubit as being "both 0 and 1 at once," a.k.a. a *superposition*. For example, $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ is an equal superposition of $|0\rangle$ and $|1\rangle$. A qubit is only in superposition until it is *measured*, at which point the outcome will be 0 with probability $|\alpha|^2$ and 1 with probability $|\beta|^2$. Measurement is not passive: it has the effect of collapsing the state to match the measured outcome, i.e., either $|0\rangle$ or $|1\rangle$.

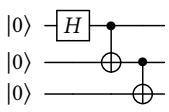
A quantum state with n qubits is represented as vector of length 2^n. We can join multiple qubits together by means of the *tensor product* ($\otimes$) from linear algebra. For convenience, we write $|x_0\rangle \otimes |x_1\rangle \otimes \cdots \otimes |x_m\rangle$ as $|x_0 x_1 ... x_m\rangle$ for $x_i \in \{0, 1\}, 0 \leq i \leq m$; we may also write $|k\rangle$ where $k \in \mathbb{N}$ is the decimal interpretation of bits $|x_0 x_1 ... x_m\rangle$. For example, a 2-qubit state is represented as a $2^2 = 4$ length vector where each component corresponds to (the square root of) the probability of measuring $|00\rangle$, $|01\rangle$, $|10\rangle$, and $|11\rangle$, respectively. We may also write these four kets as $|0\rangle, |1\rangle, |2\rangle, |3\rangle$. Sometimes a multi-qubit state cannot be expressed as the tensor product of individual qubits; such states are called *entangled*. One example is the state $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$, known as a *Bell pair*.

3.2 Quantum Programs

Quantum programs are composed of a series of *quantum operations*, each of which acts on a subset of qubits in the quantum state. Quantum operations can be expressed as matrices, and their application to a state expressed as matrix multiplication. For example, the *Hadamard* operator H on one qubit is expressed as a matrix $\frac{1}{\sqrt{2}}\begin{pmatrix}1 & 1\\ 1 & -1\end{pmatrix}$. Applying H to state $|0\rangle$ yields state $\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$, also written as $|+\rangle$. n-qubit operators are represented as $2^n \times 2^n$ matrices. For example, the *CNOT* operator over two qubits is expressed as the $2^2 \times 2^2$ matrix shown at the right.

$$\begin{pmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 0 & 1\\ 0 & 0 & 1 & 0\end{pmatrix}$$

In the standard presentation, quantum programs are expressed as *circuits*, as shown in Fig 4(a). In these circuits, each horizontal wire represents a qubit and boxes on these wires indicate quantum operations, or *gates*. The circuit in Fig 4(a) has three qubits and three gates: the *Hadamard* (H) gate and two *controlled-not* (CNOT) gates. The semantics of a gate is a *unitary matrix* (a matrix that preserves the unitarity invariant of quantum states); applying a gate to a state is tantamount to multiplying the state vector by the gate's matrix. The matrix corresponding to the circuit in Fig 4(a) is shown in Fig 4(c), where I is the 2×2 identity matrix, $CNOT$ the matrix for the CNOT gate, and H for the Hadamard gate.



(a) Quantum circuit

```
1 H 0;
2 CNOT 0 1;
3 CNOT 1 2;
```

(b) Unitary SQIR

$$(I \otimes CNOT) \times (CNOT \otimes I) \times (H \otimes I \otimes I)$$

(c) Matrix expression

Fig. 4. Example quantum program: 3-qubit GHZ state preparation.

3.3 Unitary SQIR

SQIR [28] is a simple quantum language embedded in the Coq proof assistant. SQIR's *unitary fragment* is a sub-language of full SQIR [28] for expressing programs consisting of unitary gates also with the extension of measurements.

Syntax. A unitary SQIR program P is a sequence of applications of gates G to qubits q:

$$P := P_1; P_2 \mid G\ q \mid G\ q_1\ q_2 \mid G\ q_1\ q_2\ q_3.$$

Qubits are referred to by natural numbers that index into a global register. A SQIR program is parameterized by a unitary gate set g (from which G is drawn) and the size n of the global register (i.e., the number of available qubits).

The Coq function ghz on the left recursively constructs a SQIR program, which prepares the $n+1$-qubit GHZ state. When $n = 0$, the program applies the Hadamard gate H to qubit 0. Otherwise, ghz calls itself recursively with input $n-1$ and then applies $CNOT$ to qubits q_{n-1}, q_n. For example, ghz 2 generates the circuit in Fig 4(a).

$$\llbracket P_1;\ P_2 \rrbracket_d = \llbracket P_2 \rrbracket_d \times \llbracket P_1 \rrbracket_d;$$

$$\llbracket G\ q_1 \cdots q_i \rrbracket_d = \begin{cases} apply_i(G,\ q_1, ...,\ q_i,\ d) & \text{well-typed} \\ 0_{2^d} & \text{otherwise} \end{cases},\ i = 1, 2, 3.$$

Fig. 5. Semantics of unitary SQIR programs, assuming a global register of dimension d. The $apply_k$ function maps a gate name to its corresponding unitary matrix and extends the intended operation to the given dimension by applying an identity operation on every other qubit in the system.

```
Fixpoint ghz (n : nat) : ucom g (n + 1) :=
  match n with
    | 0 ⇒ H 0
    | _ ⇒ ghz n−1; CNOT n−1 n
  end.
```

Semantics. The semantics of unitary SQIR is shown in Fig 5. First, if a program is not *well-typed* its denotation is the zero matrix (of size $2^d \times 2^d$, where d is the size of the global register). A program P is well-typed if every gate application is *valid*, meaning that its index arguments are within the bounds of the global register, and no index is repeated. The latter requirement enforces linearity and hence quantum mechanics' *no-cloning theorem* [59].

The program's denotational semantics follows from the composition of the matrices that correspond to each of the applications of its unitary gates. The only wrinkle is that a full program consists of many gates, each operating on $1 \sim 3$ qubits. Thus, a gate application's matrix needs to apply the identity operation to the qubits not being operated on. This is the purpose of using $apply_1$, $apply_2$ and $apply_3$ [3].

4 INDUCTIVE SQIR AND ITS LOGIC

Our main purpose is to synthesize programs that satisfy a high-level functionality, like generating a type of quantum states or finding the structure to core procedures in algorithms that works for general input size n. To that end, we extend the existing intermediate representation SQIR [28] to depict a family of programs called Inductive SQIR (ISQIR) in Section 4.1. For synthesis purpose, in Section 4.2, we also develop the so-called hypothesis-amplitude specification to describe the desired properties of ISQIR programs, together with a logic for reasoning, whose encoding into SMT instances will be given in Section 5.

4.1 Inductive SQIR (ISQIR)

We extend SQIR with an inductive structure, e.g., `fixpoint` in Coq, to equip the language with the ability to describe a family of quantum circuits for general input n.

Definition 4.1 (Inductive SQIR- Syntax). *An ISQIR program is defined inductively by:*

$$S ::= \textbf{\textit{const}}\ P \mid \textbf{\textit{seq}}\ S_1\ S_2 \mid \textbf{\textit{relabel}}\ \pi\ S \mid \textbf{\textit{fix}}_k\ \pi\ P_0\ P_1\ \cdots\ P_{k-1}\ S_L\ S_R.$$

Here, $P, P_0, ..., P_{k-1}$ are SQIR programs, π is a series of injective natural number mappings.

[3]For example, $apply_1(G_u,\ q_1,\ d) = I_{2^q} \otimes u \otimes I_{2^{(d-q-1)}}$ where u is the matrix interpretation of the gate G_u and I_k is the $k \times k$ identity matrix.

At a high level, any ISQIR program is a succinct way to describe a series of SQIR programs indexed by an integer (or input size) $n = 0, 1, 2, \cdots$. Intuitively, **const** P represents a repeating series of SQIR programs where every entry in the series is the same SQIR program P. **seq** S_1 S_2 concatenates two series S_1 and S_2 by concatenating SQIR programs of each entry. We also use $S_1; S_2$ and **seq** S_1 S_2 interchangeably for notation convenience. **relabel** π S permutes the qubit labels for the nth entry with permutation $\pi(n)$.

$\mathbf{fix}_k$ is the new *inductive* structure introduced to ISQIR. Specifically, $\mathbf{fix}_k$ constructs a series of SQIR programs by recursion, with k base cases $P_0, ..., P_{k-1}$, and the recursive call for the n-th entry is sandwiched by the n-th entries of ISQIR programs S_L and S_R. The choice of $\mathbf{fix}_k$ is inspired by commonly seen quantum programs and serves as a good syntax guide for synthesis purposes for all the case studies in this paper.

We formulate the *semantics* of ISQIR programs as functions from a natural number to a SQIR program, i.e., $\mathbb{N} \to$ SQIR.

Definition 4.2 (ISQIR- Semantics). *An ISQIR program represents a series of unitary SQIR programs $\{P_0, P_1, ...\}$. We define the IR semantics $\{\!|S|\!\}$ of ISQIR program inductively by:*

$$\{\!|\textbf{const } P|\!\}(n) = P; \qquad \{\!|\textbf{seq } S1\ S2|\!\}(n) = \{\!|S1|\!\}(n); \{\!|S2|\!\}(n);$$

$$\{\!|\textbf{relabel } \pi\ S|\!\}(n) = \mathtt{map\_qb}(\pi(n), \{\!|S|\!\}(n));$$

$$\{\!|\textbf{fix}_k|\!\}(n) = \begin{cases} P_n, & n < k \\ \{\!|S_L|\!\}(n); \mathtt{map\_qb}(\pi(n), \{\!|\textbf{fix}_k|\!\}(n-1)); \{\!|S_R|\!\}(n); & else \end{cases}$$

Here ; is the sequential construct in SQIR, map_qb is a function relabeling the indices of qubits in a SQIR program according to injective function $\pi(n)$, and $\mathbf{fix}_k$ is an abbreviation of $\mathbf{fix}_k$ π P_0 ... P_{k-1} S_L S_R. We use a slightly changed denotational semantics of any SQIR program P, denoted $[\![P]\!]$, which is an **infinite-dimensional** matrix (i.e., $\mathbb{N}^2 \to \mathbb{C}$), that returns entries of P's original semantics within P's dimension and 0 otherwise.

Example 4.1. As an example, recall the GHZ program written in ISQIR syntax:

$$\mathbf{fix}_1\ Id\ (\textbf{const H } 0)\ (\textbf{const ID})\ (\textbf{relabel } \pi\ (\textbf{const CNOT } 0\ 1))$$

where Id is an identity map, (**const** H 0) is a Hadamard gate on qubit q_0 which is the P_0, (**const** ID) is an identity unitary (served as S_L), and $\pi(n) = \lambda x.x + n - 1$ for $n \geq 1$. $\pi(n)$ maps the CNOT gate (served as S_R) on qubits q_0, q_1 to a CNOT gate on qubit q_{n-1}, q_n.

For notation convenience, when relabeling a SQIR program with a map π (i.e. **relabel** π **const** P), we usually omit the **relabel** key word. Thus, S_R can also be denoted as **const** CNOT n-1 n.

We also develop the following syntax for general permutation π used in ISQIR programs that is also part of the candidate program search space.

Definition 4.3 (Permutation Syntax).

$$\pi ::= Id \mid w[e_1 \rightleftharpoons e_2] \mid \textbf{shift}\ e_1\ e_2\ m \mid \pi_1 \cdot \pi_2$$
$$e ::= n \mid m \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 * e_2 \qquad m \in \mathbb{N}$$

*Id is an identity permutation; $w[e_1 \rightleftharpoons e_2]$ swaps the mapping e_1 and e_2. **shift** e_1 e_2 m maps any $x \in [e_1, e_2)$ to $[(x - e_1 + m)\ mod\ (e_2 - e_1)] + e_1)$.*

For example, permutation $w[1 \rightleftharpoons n](n)$ maps 1 to n and maps n to 1. Permutation (**shift** 2 5 1)(n) maps 4 to 2 and maps 3 to 4.

4.2 Unitary ISQIR Logic

A critical component of program synthesis is the ability of expressing desired properties of the target programs, usually called specifications. The pre and post conditions of programs in typical Hoare triples provide a natural approach to express these input-output specifications. Quantum Hoare triples [62] are hence a natural candidate for describing input-output specifications for quantum programs. However, contrary to the classical setting where pre/post conditions have a lot of flexibility in description, the conventional pre/post conditions in quantum Hoare triples are described by quantum predicates which are Hermitian matrices of exponential dimensions in terms of the system size. The exponential dimension of quantum predicates incurs both the scalability issue and technical inconvenience in automating the reasoning directly based on quantum Hoare triples.

Moreover, for ISQIR programs, one needs to express the specifications for a family of programs for different sizes, which is like classical program synthesis with a varying-length array of variables. However, existing Hoare triples can only be used to provide specifications for quantum systems of a fixed dimension.

To that end, we develop the so-called *hypothesis-amplitude* specifications for ISQIR programs, where the hypothesis component (denoted h) of the specification describes a certain subset of input states x in the computational basis, and the amplitude component (denoted α) describes the output state of the program on the given input x. Both h, α are functions of the index n so that they can describe a family of SQIR programs.

Definition 4.4. *A* hypothesis-amplitude triple *contains h, S and α. Here h is a set of tuples $(n, x, y) \in \mathbb{N}^3$ (we abuse the notation h to also represent its indicator function of type $\mathbb{N}^3 \to \mathbb{B}$) that specifies the interested entries of S' semantics, and $\alpha(n, x, y)$ is a complex function with natural number inputs.*

A hypothesis-amplitude triple is a valid judgement, denoted $h \triangleright S \leftrightarrow \alpha$, when

$$h \triangleright S \leftrightarrow \alpha \iff \forall (n, x, y) \in h,\ \langle y|[\![\{\!\{S\}\!\}(n)]\!]|x\rangle = \alpha(n, x, y). \tag{4.1}$$

Following the above definition, the hypothesis h is like classical pre-conditions and specifies the set of inputs where the post-conditions are provided. For any such input x, the output state of the program $S(n)$ is given by $[\![\{\!\{S\}\!\}(n)]\!]|x\rangle = \sum_y \alpha(n, x, y)|y\rangle$, which explains why α is called the amplitude. We do not always need to specify the program's semantics for all inputs, so we use set h to filter those unnecessary information. By the linearity of unitary, the input-output specification on a set of inputs in the computational basis can be extended to a specification in the linear space spanned by the given input set.

Compared with quantum Hoare triples, our hypothesis-amplitude specification provides a more flexible and arguably more intuitive way to formalize the desired properties on the target functions. For instance, for state preparation, our specification is almost straightforward to use and avoids the extra efforts of converting specifications into exponential-size

quantum predicates. [4] Moreover, for all unitary programs, our hypothesis-amplitude specification could provide the same expressive power as general quantum Hoare triples at the cost of using potentially complicated h, α. Nevertheless, efficient encoding into SMT instances are only known in restricted cases of h, α as discussed in Section 5.

We develop *unitary ISQIR logic* shown in Fig 6 to reason about ISQIR program's semantics with respect to the hypothesis-amplitude specification with helper functions in Def 4.5. The soundness is formally proven in Theorem 4.2, whose proof is postponed to Appendix A.1.

One can view (h, α) as a series of incomplete matrices: only those entries in the set h are known, whose values can be looked up in α. This gives the intuition behind our rules. The Weaken rule states that any subset of h can also be observed by α. The Const rule lifts SQIR semantics to ISQIR semantics. The Replace rule states that if the observed entries are the same for two amplitude functions, then one can substitute the other one with hypothesis h. The Relabel rule relabels the entries of matrices for both h and α. The Seq rule calculates matrix multiplication for each term in the series. The Fix rule checks observed entries for terms with index $i < k$ (base cases) and computes matrix multiplication for $i \geq k$ (inductive cases).

Definition 4.5. *For a hypothesis set h and an amplitude function α, the **relabeling function** and the **predecessor functions** of h and α are defined by*

$$\pi \circ (h, \alpha) := (\pi \circ h, \pi \circ \alpha) \qquad pred\ (h, \alpha) := (pred\ h, pred\ \alpha)$$
$$\pi \circ h := \{(n, \pi(n, x), \pi(n, y)) \mid (n, x, y) \in h\} \quad (\pi \circ \alpha)(n, x, y) := \alpha(n, \pi(n, x), \pi(n, y))$$
$$(pred\ \alpha)(n, x, y) := \alpha(n - 1, x, y) \qquad pred\ h := \{(n - 1, x, y) \mid (n, x, y) \in h\}$$

The entries of h and α are relabeled according to a series of injective function π. Predecessor functions move the series by one index, and are used for recursive calls in fixed-point programs.

*The **composition function** of h and α are defined by:*

$$comp(h_1, \alpha_1, h_2, \alpha_2) := \Big\{(n, x, y) : \forall z, ((n, x, z) \in h_1 \wedge (n, z, y) \in h_2) \vee ((n, x, z) \in h_1 \wedge \alpha_1(n, x, z) = 0) \vee ((n, z, y) \in h_2 \wedge \alpha_2(n, z, y) = 0)\Big\},$$
$$(\alpha_1 * \alpha_2)(n, x, y) := \sum_{z \in \mathbb{N}} \alpha_1(n, x, z)\alpha_2(n, z, y),$$
$$(h_1, \alpha_1) \otimes (h_2, \alpha_2) := (comp(h_1, \alpha_1, h_2, \alpha_2), \alpha_1 * \alpha_2).$$

*We also define several restricted **equivalence relations**:*

$$h_1 \equiv_n h_2 \Leftrightarrow (\forall x, \forall y, (n, x, y) \in h_1 \leftrightarrow (n, x, y) \in h_2)$$
$$\alpha_1 \equiv_n^h \alpha_2 \Leftrightarrow \forall x, \forall y, (n, x, y) \in h \rightarrow \alpha_1(n, x, y) = \alpha_2(n, x, y)$$
$$\alpha_1 \equiv^h \alpha_2 \Leftrightarrow \forall n, \alpha_1 \equiv_n^h \alpha_2 \qquad (h_1, \alpha_1) \equiv_n (h_2, \alpha_2) \Leftrightarrow h_1 \equiv_n h_2 \ \wedge \ \alpha_1 \equiv_n^{h_1} \alpha_2$$

Theorem 4.2. *The rules of unitary ISQIR logic in Fig 6 are sound.*

[4] Quantum Hoare triples, however, need a post-predicate where the target state spans the subspace with eigenvalue 1, and the rest space has eigenvalue 0. These complication comes from the generality of quantum Hoare triple.

$$\frac{h \triangleright S \leftrightarrow \alpha, \quad h' \subseteq h}{h' \triangleright S \leftrightarrow \alpha}\ \textsc{Weaken} \qquad \frac{\alpha \equiv^{\mathbb{N}^3} \lambda n.[\![P]\!]}{\mathbb{N}^3 \triangleright (\mathbf{const}\ P) \leftrightarrow \alpha}\ \textsc{Const}$$

$$\frac{h \triangleright S \leftrightarrow \alpha, \quad \alpha \equiv^{h} \alpha'}{h \triangleright S \leftrightarrow \alpha'}\ \textsc{Replace} \qquad \frac{h \triangleright S \leftrightarrow \alpha \quad \alpha' \equiv^{\pi \circ h} \pi \circ \alpha}{\pi \circ h \triangleright \mathbf{relabel}\ \pi\ S \leftrightarrow \alpha'}\ \textsc{Relabel}$$

$$\frac{h_i \triangleright S_i \leftrightarrow \alpha_i \quad \forall i{=}1{,}2 \quad (h,\alpha){=}(h_1,\alpha_1) \otimes (h_2,\alpha_2)}{h \triangleright \mathbf{seq}\ S_1\ S_2 \leftrightarrow \alpha}\ \textsc{Seq}$$

$$\frac{\begin{array}{c} h_L \triangleright S_L \leftrightarrow \alpha_L, \quad h_R \triangleright S_R \leftrightarrow \alpha_R \\ \forall i{<}k, \quad h_i \triangleright \mathbf{const}\ P_i \leftrightarrow \alpha_i, \quad (h,\alpha) \equiv_i (h_i,\alpha_i) \\ \forall i{\geq}k, \quad (h,\alpha) \equiv_i (h_L,\alpha_L) \otimes (\pi \circ \mathrm{pred}\ (h,\alpha)) \otimes (h_R,\alpha_R) \end{array}}{h \triangleright \mathbf{fix}_k\ P_0\ \cdots\ P_{k-1}\ S_L\ S_R \leftrightarrow \alpha}\ \textsc{Fix}$$

Fig. 6. The unitary ISQIR logic.

5 ENCODING VERIFICATION PROCESS

In this section we explain how to encode the hypothesis-amplitude triple introduced In Definition 4.4 and the functions and relations in Definition 4.5 into the SMT instance.

5.1 Encoding the hypothesis-amplitude triple

The hypothesis-amplitude triple in Definition 4.4 includes a hypothesis set h and an amplitude function α, which will be encoded separately.

5.1.1 Encoding the hypothesis set. The hypothesis set h refers to a set of natural numbers. Intuitively, we encode the hypothesis h with a Boolean expression B constructed by n, x, y, whose value is true if and only if (n, x, y) is inside the hypothesis set. Namely,

$$(n, x, y) \in h \iff B(n, x, y) = \mathbf{True}. \tag{5.1}$$

5.1.2 Encoding the amplitude function. Encoding the complex function α is challenging since there is currently no automated program verification tool that supports complex numbers. We solve this by restricting the function α in a limited form that can be encoded into SMT instances. This is a trade-off between the expressiveness of our specification and the feasibility of automated verification.

Parameterized path-sum amplitude (PPSA) function. We restrict an amplitude function α to be a *Parameterized Path-Sum Amplitude* function:

Definition 5.1. *A Parameterized Path-Sum Amplitude (PPSA) function $\alpha_p : \mathbb{N}^3 \to \mathbb{C}$ is defined as*

$$\alpha_p(n, x, y) := \frac{1}{\sqrt{\beta(n)}} \sum_{i=0}^{m} \delta(B_i(n, x, y)) \cdot e^{2\pi i \cdot d(V_i(n,x,y))}$$

- *β is a natural number expression constructed only by n and it decides the magnitude of all possible paths.*
- *$m \in \mathbb{N}$ is a constant number. $\{B_i\}_m$ is a group of boolean expressions constructed by (n, x, y) and satisfies $\forall n\ x\ y \in \mathbb{N}, \sum_{i=0}^{m} \delta(B_i(n, x, y)) \leq 1$, where $\delta(B) : Bool \to \{0, 1\}$ is a function that returns 1 if B is True and returns 0 otherwise.*

$$n, x, y : \text{Variables} \in \mathbb{N} \quad k : \text{Fixed number} \in \mathbb{Z}$$

$$\begin{aligned} \text{Boolean Expression } B ::= & \quad \textbf{True} \mid \textbf{False} \mid B_1 \wedge B_2 \mid B_1 \vee B_2 \mid \neg B' \mid V_1 \ \textbf{rop} \ V_2 \\ \text{Binary-}\mathbb{N}\ V ::= & \quad x \mid y \mid n \mid k \mid \delta(B) \mid V_1[V_2] \mid V'[V_1 : V_2] \mid \textbf{uop}\ V' \mid V_1 \ \textbf{bop} \ V_2 \\ \text{Magnitude } \beta ::= & \quad k \mid n \mid \beta_1 \ \textbf{bop} \ \beta_2 \mid 2^n \end{aligned}$$

Fig. 7. Syntax of PPSA function.

- $\{V_i\}_m$ *is a group of natural number expressions constructed by* (n, x, y).
- *For* $x \in \mathbb{N}$, *suppose* x*'s binary representation is* $x_q ... x_1 x_0$ *where* $x_i \in \{0, 1\}$, *we use* $d(x)$ *to denote the fractional binary notation of* x.

$$d(x) = [0.x_0 x_1 ... x_q]_2 = \sum_{i=0}^{q} x_i \cdot 2^{-(i+1)}.$$

Fig. 7 shows the syntax we allow to construct the expressions β, B, V in a PPSA function. $\delta(B)$ is a {0,1}-value function that returns 1 if B is True and returns 0 otherwise. "V_1 **rop** V_2" is a set of common relational operators bewteen V_1, V_2(e.g. $= \neq > < \geq \leq$). "**uop** V" is a set of common unary operator on V(i.e. - ! & | $\oplus$). "V_1 **bop** V_2" is a set of binary operators between V_1 and V_2, including common arithmetic operators (i.e. + - * \ %) and bit-wise operators (i.e. & | $\oplus$ « »). All operators in **rop,uop,bop** have the same meaning as they have in C language. $V_1[V_2]$ means the V_2-th bit of V_1's binary representation (in the order from low to high). $v'[V_1 : V_2]$ is the natural number represented by the binary representation truncated from high bit V'_{V_1} to low bit V'_{V_2}. For example, let $V_1 = (6)_{10} = (110)_2$, we have $V_1[0] = 0, V_1[2 : 1] = (11)_2 = 3$. All these syntaxes are supported by the Z3 SMT solver.

By restricting amplitude function α to a PPSA function, we disassembled the complex number function α into the combination of several integer or boolean expressions, which allows us to represent α with a set of SMT expressions that enable us to encode the calculation in Definition 4.5 into the SMT solver. This will be discussed in Section 5.2.

Our design for the PPSA function is inspired by Richard Feynman's *sum-over-path* formalism of quantum mechanics, which has also inspired many quantum state representations [3–5, 7–9, 11, 14, 41, 41, 45]. However, all of these representations can only express constant size unitary operators and fail to work for any input size.

PPSA inherits the expressibility of the existing sum-over-path representations, which can be used to express most famous quantum algorithms (e.g., [3, 11]), and works for a general input size. Hence, we believe the restriction to PPSA is mild and serves as a good balance between expressiveness and feasibility. Some common unitary operators that can be represented by hypothesis-amplitude triple while restricting the amplitude function to PPSA are listed in Table 1. More examples are provided in Section 6.

5.2 Encoding reasoning based on ISQIR logic

To enable SMT-based automation in reasoning, one needs to encode the relabeling functions, composition functions, predecessor function and the equivalence relations of h, α defined in Definition 4.5 into SMT instances. In the cases of the relabeling functions $(\pi \circ h)$, $(\pi \circ \alpha)$,

Name	Unitary Operator	Hypothesis-Amplitude Specification
Uniform_{n+1}	$\|0\rangle^{n+1} \mapsto \frac{1}{\sqrt{2^{n+1}}} \sum_{0 \le y < 2^{n+1}} \|y\rangle$	$h = \{(n,x,y) \mid x = 0\}$ $\alpha(n,x,y) = \frac{1}{\sqrt{2^{n+1}}} \delta(y < 2^{n+1})$
Toffoli_{n+1}	$\|q_0 q_1 \cdots q_n\rangle \mapsto$ $\|q_0 q_1 \cdots (q_n \oplus \prod_{i=0}^{n-1} q_i))\rangle$	$h = \{(n,x,y) \mid x < 2^{n+1} \wedge y < 2^{n+1}\}$ $\alpha(n,x,y) = \delta(x[n-1:0] = y[n-1:0] \wedge$ $y[n] = x[n] \oplus (\&x[n-1:0]))$
QFT_{n+1}	$\|x\rangle \mapsto \frac{1}{\sqrt{2^n}} \sum_{y=0}^{2^n-1} e^{\frac{2\pi i \cdot xy}{2^n}} \|y\rangle$	$h = \{x < 2^{n+1} \wedge y < 2^{n+1}\}$ $\alpha(n,x,y) = \frac{1}{\sqrt{2^{n+1}}} \cdot e^{2\pi i \cdot d((x \cdot y) \gg (n+1))}$

Table 1. Examples of Unitary Operators represented by the Hypothesis-Amplitude Specification.

the predecessor function pred (h, α) and the composition function $\text{comp}(h_1, \alpha_1, h_2, \alpha_2)$, all used operations are supported by SMT solvers directly and the encoding is trivial.

We hence focus our discussison on the non-trivial encoding of the composition function $\alpha_1 * \alpha_2$ and the equivalence relations.

5.2.1 Encoding the composition function. Recall the $\alpha_1 * \alpha_2$ function from Definition 4.5:

$$(\alpha_1 * \alpha_2)(n,x,y) = \sum_{z \in \mathbb{N}} \alpha_1(n,x,z)\alpha_2(n,z,y).$$

Since the summation is over $z \in \mathbb{N}$, by definition, the $\alpha_1 * \alpha_2$ function is a composition of two infinite-dimension unitaries, and hence cannot be calculated directly. All existing symbolic matrix multiplication methods can only deal with a fixed dimension or a fixed number of terms (e.g., [3]) and hence are not applicable in our case.

Fortunately, we observe that in many cases, non-zero values of the function α are sparse, making the composition possible. *In particular, we show the possibility of computing the function $\alpha_1 * \alpha_2$ when one of α_1 or α_2 is sparse.* The sparsity of α is precisely defined as

Definition 5.2. *We say a function $\alpha : \mathbb{N}^3 \to \mathbb{C}$ is **sparse** iff: there exist two functions $\mathcal{X}, \mathcal{Y} : \mathbb{N}^2 \to \{\mathbb{N}\}$ and for any inputs, the sets returned by $\mathcal{X}, \mathcal{Y}$ always have constant sizes (i.e., independent of inputs n, x, y), and further satisfy*

$$\forall n\ x\ y \in \mathbb{N}, \quad \alpha(n,x,y) \neq 0 \to x \in \mathcal{X}(n,y) \wedge y \in \mathcal{Y}(n,x).$$

We denote such sparsity by $\alpha \trianglelefteq (\mathcal{X}, \mathcal{Y})$.

Intuitively, when $\alpha \trianglelefteq (\mathcal{X}, \mathcal{Y})$ holds, for any given $n_0, x_0 \in \mathbb{N}$, $\alpha(n_0, x_0, y)$ has non-zero values only on a finite of y points, the set of which is $\mathcal{Y}(n_0, x_0)$. The same intuition holds for $\mathcal{X}$ except for the case when n_0, y_0 are given.

Example 5.1. We show the amplitude function that can represent the ISQIR program **const** H 0 and its sparsity tuple $\mathcal{X}, \mathcal{Y}$ as an example.

$$\mathbb{N}^3 \triangleright \textbf{const H } 0 \leftrightarrow \alpha_H, \qquad \alpha_H(n,x,y) = \frac{1}{\sqrt{2}} \delta(x \backslash 2 = y \backslash 2) \cdot e^{2\pi i \cdot \frac{x[0] * y[0]}{2}}$$

$$\alpha_H \trianglelefteq (\mathcal{X}, \mathcal{Y}), \qquad \mathcal{X}(n,y) = \{y, y \oplus 1\}, \qquad \mathcal{Y}(n,x) = \{x, x \oplus 1\}.$$

The operation $\oplus$ is a bit-wise operation and the expression $x \oplus 1$ flips the 0th bit of x (e.g. $(101)_2 \oplus 1 = (100)_2 = 4$). The expression $\delta(x\backslash 2 = y\backslash 2)$ in α_H indicates that

$$\forall n\ x\ y \in \mathbb{N}, \quad \alpha_H(n,x,y) \neq 0 \longrightarrow x \in \mathcal{X}(n,y) \wedge y \in \mathcal{Y}(n,x)$$

Intuitively, $\mathcal{X}, \mathcal{Y}$ are constructed in this way because the program **const** H 0 only modify the state of the 0th qubit.

Now we explain how to encode function $\alpha_1 * \alpha_2$ when one of α_1, α_2 is sparse. Suppose α_2 is sparse and we have $\alpha_2 \trianglelefteq (\mathcal{X}, \mathcal{Y})$, we know that $\alpha_2(n,z,y) \neq 0$ only when $z \in \mathcal{X}(n,y)$. So $\alpha_1 * \alpha_2$ can be calculated by

$$(\alpha_1 * \alpha_2)(n,x,y) = \sum_{z\in\mathbb{N}} \alpha_1(n,x,z)\alpha_2(n,z,y) = \sum_{z\in\mathcal{X}(n,y)} \alpha_1(n,x,z)\alpha_2(n,z,y).$$

The summation on the right hand has only a fixed number of terms by sparsity which allows encoding into SMT instances. Similarly, when α_1 is sparse and $\alpha_1 \trianglelefteq (\mathcal{X}, \mathcal{Y})$, we have

$$(\alpha_1 * \alpha_2)(n,x,y) = \sum_{z\in\mathbb{N}} \alpha_1(n,x,z)\alpha_2(n,z,y) = \sum_{z\in\mathcal{Y}(n,x)} \alpha_1(n,x,z)\alpha_2(n,z,y).$$

Moreover, sparsity of α can be established in many cases. (Proof in Appendix A.1).

Theorem 5.2 (α-sparsity). *Suppose $\alpha, \alpha_1, \alpha_2$ are amplitude functions:*

- *Let P be a unitary SQIR program and $\mathbb{N}^3 \triangleright$ **const** $P \leftrightarrow \alpha$, then α is sparse.*
- *If α is sparse and π is a series of injective natural number mappings, then $\pi \circ \alpha$ is sparse.*
- *If both α_1, α_2 are sparse, so is $\alpha_1 * \alpha_2$.*

The above theorem shows that the α functions for all SQIR programs, and for relabeling a SQIR program or composing two SQIR programs are sparse. So non-sparse αs only appear in the fixpoint syntax. The candidate program from our searcher has at most one fixpoint due to the challenge discussed in Section 2.4. So when composing two amplitude functions α_1, α_2, there is always at least one sparse function and our composition strategy can work.

5.2.2 Encoding the equivalence relations. Given a hypothesis h and two complex functions α, α', suppose the functions α, α' are in the form:

$$\alpha(n,x,y) = \frac{1}{\sqrt{\beta(n)}} \sum_{i=0}^{m} \delta(B_i(n,x,y)) \cdot e^{2\pi i \cdot d(V_i(n,x,y))}, \quad \alpha'(n,x,y) = \frac{1}{\sqrt{\beta'(n)}} \sum_{i=0}^{m'} \delta(B'_i(n,x,y)) \cdot e^{2\pi i \cdot d(V'_i(n,x,y))}.$$

QSynth verifier checks the equivalence $\alpha \equiv^h \alpha'$ by the checking following SMT instance and rejects the equivalence when it the SMT solver gives a negative result.

$$\begin{aligned}\forall n\ x\ y \in \mathbb{N},\ \ h(n,x,y) \longrightarrow & \beta(n) = \beta'(n) \wedge \sum_{i=0}^{m} B_i(n,x,y) = \sum_{i=0}^{m'} B'_i(n,x,y) \\ & \wedge \sum_{i=0}^{m} B_i(n,x,y) * V_i(n,x,y) = \sum_{i=0}^{m'} B'_i(n,x,y) * V'_i(n,x,y).\end{aligned}$$

6 EXPERIMENTAL CASE STUDIES

We demonstrate additional case studies: the N-qubit Quantum Adder and the N-qubit Quantum Fourier Transform program. We provide Qiskit programs compiled from synthesized ISQIR programs for better illustration, while keeping the ISQIR programs in Appendix A.2.

6.1 Quantum Adder

Motivation and Background. Quantum circuits for arithmetic operations are required for quantum algorithms. One important example is the adder circuit. Feyman [20, 47] first proposes the quantum *full adder* circuit to implement $|0\rangle|A_n\rangle|B_n\rangle|0\rangle^{\otimes n} \rightarrow |c_0\rangle|A_N\rangle|B_n\rangle|A_n + B_n\rangle$ where A_n, B_n are n-bit natural number. The first $|0\rangle$ is the carry bit and it is changed to carry value $|c_0\rangle$ after the addition. This design unfortunately needs n more qubits to store the sum of $A_n + B_n$. To reduce the qubit usage, Cuccaro[13] proposed a new *ripple-carry adder* that uses n fewer qubits than the full adder design. When given different specifications, QSynth can synthesize both adder circuits.

Full Adder Synthesis. We let QSynth synthesize a program S_a that $S_a(n)$ provides a n-qubit full quantum adder (i.e. $S_a(0)$ is an identity unitary). Let QSynth use qubit q_0 as the initial carry bit, and use $q_1, ...q_n$ to represent natural number A_n, use qubit $q_{n+1}, q_{n+2}, ..., q_{2n}$ to represent natural number B_n and use qubit $q_{2n+1}, q_{2n+3}, ..., q_{3n}$ to store $A_n + B_n$. To synthesize such a program S_f, the hypothesis specification h is given as:

$$h := \{x < 2^{3n+1} \wedge y < 2^{3n+1} \wedge x[0] = 0 \wedge x[3n : 2n + 1] = 0\}.$$

This hypothesis suggests: (1) we only focus on qubits $q_0 \sim q_{3n}$ (i.e. $x < 2^{3n+1} \wedge y < 2^{3n+1}$); (2) initial carry bit q_0 is 0 (i.e. $x[0] = 0$); (3) the qubits to store the sum are initialized as $|0\rangle^n$ (i.e. $x[3n : 2n + 1] = 0$). The amplitude specification α is given as

$$\alpha(n, x, y) := \delta(y[2n : 1] = x[2n : 1] \wedge C_n + y[0] \cdot 2^n = A_n + B_n).$$

Here $A_n = x[n : 1], B_n = x[2n : n + 1]$ represent the two numbers to sum and $C_n = y[3n : 2n + 1]$ represents the final state of qubit $q_{2n+1} \sim q_{3n}$ which should store the sum. This amplitude specification suggests: (1)A_n, B_n stay unchanged (i.e. $y[2n : 1] = x[2n : 1]$); (2) qubits $2n + 1 \sim 3n$ will store the sum of A_n, B_n and qubit q_0 stores the carry value (i.e. $y[0] \cdot 2^n + C_n = A_n + B_n$).

QSynth synthesize a program S_a as shown in Fig 8(a)(c) (i.e. Qiskit function `full_adder`). When $n = 0$, S_a does nothing since the circuit only contains the carry bit. When $n \geq 1$, S_a first call $S_a(n-1)$ recursively to get a $n-1$-bit full adder to calculate $|A_{n-1} + B_{n-1}\rangle$ and the carry bit is stored in qubit 0. Then S_a uses the one-bit adder circuit S_R to sum the highest bit in A_n and B_n.

Cuccaro's Adder Synthesis. To reduce the number of qubits in the circuit, we want to synthesize an in place adder: $|0\rangle|A_n\rangle|B_n\rangle \mapsto |c_0\rangle|A_n\rangle|A_n + B_n\rangle$. We let QSynth synthesize a program $S_c(n)$ that use qubit 0 as the carry bit to sum two n-bit numbers represented by qubits $1 \sim n$ and qubits $n + 1 \sim 2n$ respectively, and stores the result in qubits $n + 1 \sim 2n$. The specification is given as

$$h := \{x < 2^{2n+1} \wedge y < 2^{2n+1} \wedge x[0] = 0\}$$
$$\alpha(n, x, y) := \delta(y[n : 1] = x[n : 1] \wedge y[0] \cdot 2^n + B'_n = A_n + B_n)$$

where $B'_n = y[2n : n + 1]$ represents the final state of qubit $q_{n+1} \sim q_{2n}$. With this specification, QSynth generates program S_c shown in Fig 8(b)(d) (i.e. Qiskit function `Cuccaro_adder`).

6.2 Quantum Fourier Transform

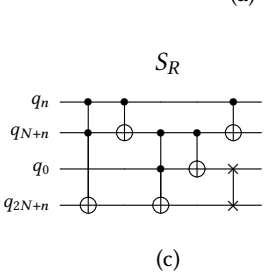
Motivation and Background. Quantum Fourier Transform (QFT) [12] is the classical discrete Fourier Transform applied to the vector of amplitudes of a quantum state, where we usually consider vectors of length $N = 2^n$. QFT is a part of many quantum algorithms, notably Shor's

```
def full_adder(N):
    circuit = QuantumCircuit(3N+1)
    def Sa(circ, n):
        if n==0:
            return
        else:
            Sa(circ,n-1)
            # See the circuit SR below
            circ.ccx(n, N+n, 2N+n)
            circ.cx(n, N+n)
            circ.ccx(N+n, 0, 2N+n)
            circ.cx(N+n, 0)
            circ.cx(n, N+n)
            circ.swap(0, 2N+n)
    Sa(circuit,N)
    return circuit
```

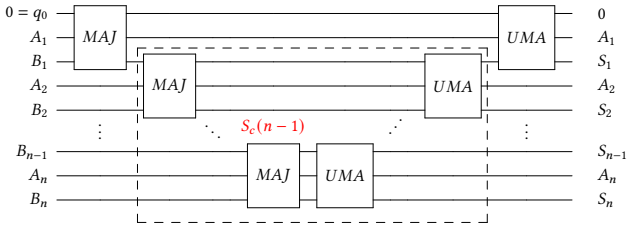
(a)

```
def Cuccaro_adder(N):
    circuit = QuantumCircuit(2N+1)
    def Sc(circ, n):
        if n==0:
            return
        else:
            # MAJ
            circ.cx(N+1, 1)
            circ.cx(N+1, 0)
            circ.ccx(0, 1, N+1)
            Sc(circ,n-1)
            # UMA
            circ.ccx(0, 1, N+1)
            circ.cx(N+1, 0)
            circ.cx(0, 1)
    Sc(circuit,N)
    return circuit
```

(b)



(c)



(d)

Fig. 8. (a)(c) Full quantum adder program S_a written in Qiskit. `circ.ccx(a,b,c)` means appending a Toffoli gate that controlled by qubit q_a, q_b on qubit q_c to the circuit `circ`. The circuit S_R is exactly a one-bit quantum full adder circuit. (b)(d) Cuccaro's quantum ripple-carry adder program written in Qiskit language.

algorithm [54] for factoring and computing the discrete logarithm, the quantum phase estimation algorithm [37] for estimating the eigenvalues of a unitary operator, and algorithms for the hidden subgroup problem [17]. A QFT over $\mathbb{Z}_{2^n}$ can be expressed as a map in two equivalent forms:

$$\text{QFT: } |x\rangle \mapsto \frac{1}{\sqrt{2^n}} \sum_{y=0}^{2^n-1} e^{\frac{2\pi i \cdot xy}{2^n}} |y\rangle \qquad \text{OR} \qquad \text{QFT: } |x\rangle \mapsto \bigotimes_{k=0}^{n-1} |z_k\rangle \tag{6.1}$$

$$|z_k\rangle = \frac{1}{\sqrt{2}}(|0\rangle + e^{2\pi i \cdot [0.x_k x_{k-1} \ldots x_0]} |1\rangle), \qquad [0.x_m \ldots x_1 x_0] = \sum_{k=0}^{m} \frac{x_k}{2^{m-k+1}}. \tag{6.2}$$

Synthesis Process. Suppose we want to synthesize a program S_Q that $S_Q(n)$ returns the $n+1$-qubit QFT circuit, then the (h, α) specification can be designed as

$$h := \{x < 2^{n+1} \wedge y < 2^{n+1}\}, \qquad \alpha(n,x,y) := \frac{1}{\sqrt{2^{n+1}}} \cdot e^{2\pi i \cdot d((x \cdot y) \gg (n+1))}. \tag{6.3}$$

Note that $e^{-2\pi i \cdot (x \cdot y) \gg (n+1)}$ equals $e^{-2\pi i \cdot \frac{(x \cdot y)}{2^{n+1}}}$. QSynth fails to synthesize a simple fixpoint structure QFT circuit so we synthesize it in two steps to help QSynth synthesize a nested structure circuit.

In the first step, we let the synthesizer generate a program S_z that transforms the state of qubit n into state $|z_n\rangle$ and keep the state of qubits $q_0 \sim q_{n-1}$ unchanged. From the Equation 6.2 we know

```
def Zn(N):
    circ = QuantumCircuit(N+1)
    def S(circ,n):
        if n == 0:
            circ.h(N)
        else:
            S(circ,n-1)
            circ.cp(pi/2**n , N-n, N)
    S(circ,N)
    return circ
```

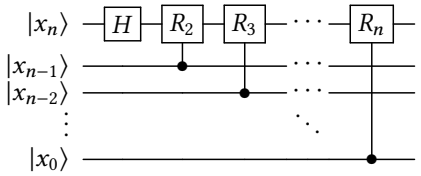


Fig. 9. Qiskit program Zn that transforms qubit q_n to state $|z_n\rangle$

```
def QFT(N):
    circuit = QuantumCircuit(N+1)
    def S(circ,n):
        if n == 0:
            circ.h(n)
        else:
            circ.append(Zn(n))
            S(circ, n-1)
    S(circuit, N)
    return circuit
```

Fig. 10. $N+1$-bit QFT program written in Qiskit language.

Benchmark	Time (s)
GHZ	0.6
Full Adder	47.6
Cuccaro Adder	98.4
QFT	55.1 (step 1), 8.2 (step 2)

Fig. 11. QSynth's runtime in seconds of the synthesis for example cases.

the (h, α) specification for the desired program S_z is:

$$h' := \{x < 2^{n+1} \wedge y < 2^{n+1}\}, \quad \alpha'(n,x,y) := \delta(y[n-1:0] = x[n-1:0]) \cdot e^{2\pi i \cdot d(\delta(y[n]=1)\cdot x)}$$

$\delta(y[n-1:0] = x[n-1:0])$ suggests qubits $q_0 \sim q_{n-1}$ remain unchanged. With this specification, the synthesizer generates the program as shown in Fig 9. Statement `circ.cp(pi/2**n, N-n, N)` in Qiskit means a controlled phase rotation gate R_n[5] on qubit q_N controlled by qubit q_{N-n}. We insert this ISQIR program S_z (i.e. Qiskit program Zn) into the database so QSynth can use it for further synthesis, which leads to the QFT program as shown in Fig 10.

6.3 Implementation and Performance

QSynth Searcher is based on the basic Syntax-Guided Top-down Tree Search [25]. We add several search bounds to keep a finite search space. QSynth can synthesize all example programs in this paper under these bounds. When searching a candidate program under ISQIR syntax in Definition 4.1, we set the maximum program length to 10 and enumerate the value of k in the FIX syntax in $\{1, 2, 3\}$. Shorter candidate programs are sent to the verifier first.

When searching a permutation π under the syntax in Definition 4.3, we set the maximum syntax derivation depth to 4. When deriving the syntax rule $e ::= m, m \in \mathbb{N}$, we enumerate $m \in [0, 3]$.

Fig 11 presents the performance of QSynth over synthesis examples. All runtimes are median of three runs. We let QSynth use SQIR's standard gate library as the search space of const SQIR program, which includes common one-qubit, two-qubit gate (e.g., Pauli gate, `CNOT` gate, swap gate, controlled phase rotation gate) and Toffoli gate as the only three-qubit gate.

[5]Precisely, R_n is a single-qubit unitary $\begin{pmatrix} 1 & 0 \\ 0 & \omega_n \end{pmatrix}$ where $\omega_n = exp(2\pi i/2^n)$.

REFERENCES

[1] Alur, R., Bodik, R., Juniwal, G., Martin, M. M., Raghothaman, M., Seshia, S. A., Singh, R., Solar-Lezama, A., Torlak, E., and Udupa, A. *Syntax-guided synthesis.* IEEE, 2013.

[2] Alur, R., Singh, R., Fisman, D., and Solar-Lezama, A. Search-based program synthesis. *Communications of the ACM 61*, 12 (2018), 84–93.

[3] Amy, M. Towards large-scale functional verification of universal quantum circuits. *arXiv preprint arXiv:1805.06908* (2018).

[4] Amy, M., Azimzadeh, P., and Mosca, M. On the controlled-not complexity of controlled-not–phase circuits. *Quantum Science and Technology 4*, 1 (2018), 015002.

[5] Amy, M., Maslov, D., and Mosca, M. Polynomial-time t-depth optimization of clifford+ t circuits via matroid partitioning. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 33*, 10 (2014), 1476–1489.

[6] Amy, M., Maslov, D., Mosca, M., and Roetteler, M. A meet-in-the-middle algorithm for fast synthesis of depth-optimal quantum circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 32*, 6 (2013), 818–830.

[7] Amy, M., and Mosca, M. T-count optimization and reed–muller codes. *IEEE Transactions on Information Theory 65*, 8 (2019), 4771–4784.

[8] Bacon, D., van Dam, W., and Russell, A. Analyzing algebraic quantum circuits using exponential sums. *unpublished: see tinyurl. com/qpo7s2* (2008).

[9] Bravyi, S., and Gosset, D. Improved classical simulation of quantum circuits dominated by clifford gates. *Physical review letters 116*, 25 (2016), 250501.

[10] Ceska, M., Hensel, C., Junges, S., and Katoen, J.-P. Counterexample-driven synthesis for probabilistic program sketches. In *International Symposium on Formal Methods* (2019), Springer, pp. 101–120.

[11] Chareton, C., Bardin, S., Bobot, F., Perrelle, V., and Valiron, B. A deductive verification framework for circuit-building quantum programs. *arXiv preprint arXiv:2003.05841* (2020).

[12] Coppersmith, D. An approximate fourier transform useful in quantum factoring. *arXiv preprint quant-ph/0201067* (2002).

[13] Cuccaro, S. A., Draper, T. G., Kutin, S. A., and Moulton, D. P. A new quantum ripple-carry addition circuit. *arXiv preprint quant-ph/0410184* (2004).

[14] Dawson, C. M., and Nielsen, M. A. The solovay-kitaev algorithm. *arXiv preprint quant-ph/0505030* (2005).

[15] de Brugiere, T. G. *Methods for optimizing the synthesis of quantum circuits.* PhD thesis, Universite Paris-Saclay, 2020.

[16] D'Antoni, L., Hu, Q., Kim, J., and Reps, T. Programmable program synthesis. In *International Conference on Computer Aided Verification* (2021), Springer, pp. 84–109.

[17] Ettinger, M., and Hoyer, P. A quantum observable for the graph isomorphism problem. *arXiv preprint quant-ph/9901029* (1999).

[18] Feng, Y., Martins, R., Wang, Y., Dillig, I., and Reps, T. W. Component-based synthesis for complex apis. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (2017), pp. 599–612.

[19] Feser, J. K., Chaudhuri, S., and Dillig, I. Synthesizing data structure transformations from input-output examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2015), PLDI '15, Association for Computing Machinery, p. 229–239.

[20] Feynman, R. P. Quantum mechanical computers. *Optics news 11*, 2 (1985), 11–20.

[21] Greenberger, D. M., Horne, M. A., and Zeilinger, A. Going beyond bell's theorem. 69–72.

[22] Gulwani, S. Automating string processing in spreadsheets using input-output examples. *ACM Sigplan Notices 46*, 1 (2011), 317–330.

[23] Gulwani, S., Harris, W. R., and Singh, R. Spreadsheet data manipulation using examples. *Commun. ACM 55*, 8 (aug 2012), 97–105.

[24] Gulwani, S., Korthikanti, V. A., and Tiwari, A. Synthesizing geometry constructions. *ACM SIGPLAN Notices 46*, 6 (2011), 50–61.

[25] Gulwani, S., Polozov, O., and Singh, R. Program synthesis. *Foundations and Trends® in Programming Languages 4*, 1-2 (2017), 1–119.

[26] Guo, Z., Cao, D., Tjong, D., Yang, J., Schlesinger, C., and Polikarpova, N. Type-directed program synthesis for restful apis. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (New York, NY, USA, 2022), PLDI 2022, Association for Computing Machinery, p. 122–136.

[27] Guo, Z., James, M., Justo, D., Zhou, J., Wang, Z., Jhala, R., and Polikarpova, N. Program synthesis by type-guided abstraction refinement. *Proc. ACM Program. Lang. 4*, POPL (dec 2019).

[28] Hietala, K., Rand, R., Hung, S.-H., Wu, X., and Hicks, M. A verified optimizer for quantum circuits. *Proc. ACM Program. Lang. 5*, POPL (Jan. 2021).

[29] Hillery, M., Buzek, V., and Berthiaume, A. Quantum secret sharing. *Physical Review A 59*, 3 (1999), 1829.

[30] Hu, Q., Cyphert, J., D'Antoni, L., and Reps, T. Synthesis with asymptotic resource bounds. In *International Conference on Computer Aided Verification* (2021), Springer, pp. 783–807.

[31] Hu, Q., and D'Antoni, L. Syntax-guided synthesis with quantitative syntactic objectives. In *International Conference on Computer Aided Verification* (2018), Springer, pp. 386–403.

[32] James, M. B., Guo, Z., Wang, Z., Doshi, S., Peleg, H., Jhala, R., and Polikarpova, N. Digging for fold: synthesis-aided api discovery for haskell. *Proceedings of the ACM on Programming Languages 4*, OOPSLA (2020), 1–27.

[33] Jha, S., Gulwani, S., Seshia, S. A., and Tiwari, A. Oracle-guided component-based program synthesis. In *2010 ACM/IEEE 32nd International Conference on Software Engineering* (2010), vol. 1, IEEE, pp. 215–224.

[34] Jha, S., and Seshia, S. A. A theory of formal synthesis via inductive learning. *Acta Informatica 54*, 7 (2017), 693–726.

[35] Ji, R., Xia, J., Xiong, Y., and Hu, Z. Generalizable synthesis through unification. *Proceedings of the ACM on Programming Languages 5*, OOPSLA (2021), 1–28.

[36] Kim, J., Hu, Q., D'Antoni, L., and Reps, T. Semantics-guided synthesis. *Proceedings of the ACM on Programming Languages 5*, POPL (2021), 1–32.

[37] Kitaev, A. Y. Quantum measurements and the abelian stabilizer problem. *arXiv preprint quant-ph/9511026* (1995).

[38] Kitaev, A. Y. Quantum computations: algorithms and error correction. *Russian Mathematical Surveys 52*, 6 (1997), 1191.

[39] Kitzelmann, E. Inductive programming: A survey of program synthesis techniques. In *International workshop on approaches and applications of inductive programming* (2009), Springer, pp. 50–73.

[40] Knoth, T., Wang, D., Polikarpova, N., and Hoffmann, J. Resource-guided program synthesis. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2019), pp. 253–268.

[41] Koh, D. E., Penney, M. D., and Spekkens, R. W. Computing quopit clifford circuit amplitudes by the sum-over-paths technique. *arXiv preprint arXiv:1702.03316* (2017).

[42] Liang, P., Jordan, M. I., and Klein, D. Learning programs: A hierarchical bayesian approach. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)* (2010), pp. 639–646.

[43] Liao, C.-H., Yang, C.-W., and Hwang, T. Dynamic quantum secret sharing protocol based on ghz state. *Quantum information processing 13*, 8 (2014), 1907–1916.

[44] Menon, A., Tamuz, O., Gulwani, S., Lampson, B., and Kalai, A. A machine learning framework for programming by example. In *International Conference on Machine Learning* (2013), PMLR, pp. 187–195.

[45] Montanaro, A. Quantum circuits and low-degree polynomials over. *Journal of Physics A: Mathematical and Theoretical 50*, 8 (2017), 084002.

[46] Nori, A. V., Ozair, S., Rajamani, S. K., and Vijaykeerthy, D. Efficient synthesis of probabilistic programs. *ACM SIGPLAN Notices 50*, 6 (2015), 208–217.

[47] of Technology (TU Delft), Q. D. U. "code example: Quantum full adder". https://www.quantum-inspire.com/kbase/full-adder/.

[48] Phothilimthana, P. M., Thakur, A., Bodik, R., and Dhurjati, D. Scaling up superoptimization. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems* (2016), pp. 297–310.

[49] Polikarpova, N., Kuraj, I., and Solar-Lezama, A. Program synthesis from polymorphic refinement types. *ACM SIGPLAN Notices 51*, 6 (2016), 522–538.

[50] Polozov, O., and Gulwani, S. Flashmeta: A framework for inductive program synthesis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (2015), pp. 107–126.

[51] Saad, F. A., Cusumano-Towner, M. F., Schaechtle, U., Rinard, M. C., and Mansinghka, V. K. Bayesian synthesis of probabilistic programs for automatic data modeling. *Proceedings of the ACM on Programming Languages 3*, POPL (2019), 1–32.

[52] Saeedi, M., Wille, R., and Drechsler, R. Synthesis of quantum circuits for linear nearest neighbor architectures. *Quantum Information Processing 10*, 3 (2011), 355–377.

[53] Shende, V. V., Bullock, S. S., and Markov, I. L. Synthesis of quantum-logic circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 25*, 6 (2006), 1000–1010.

[54] Shor, P. W. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th annual symposium on foundations of computer science* (1994), Ieee, pp. 124–134.

[55] Solar-Lezama, A. *Program synthesis by sketching*. University of California, Berkeley, 2008.

[56] Srivastava, S., Gulwani, S., and Foster, J. S. From program verification to program synthesis. In *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (2010), pp. 313–326.

[57] Tao, R., Shi, Y., Yao, J., Li, X., Javadi-Abhari, A., Cross, A. W., Chong, F. T., and Gu, R. Giallar: push-button verification for the qiskit quantum compiler. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (2022), pp. 641–656.

[58] Torlak, E., and Bodik, R. Growing solver-aided languages with rosette. In *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software* (2013), pp. 135–152.
[59] Wootters, W. K., and Zurek, W. H. A single quantum cannot be cloned. *Nature 299* (1982), 802–803.
[60] Xia, Y., Fu, C.-B., Zhang, S., Hong, S.-K., Yeon, K.-H., and Um, C.-I. Quantum dialogue by using the ghz state. *arXiv preprint quant-ph/0601127* (2006).
[61] Xu, M., Li, Z., Padon, O., Lin, S., Pointing, J., Hirth, A., Ma, H., Palsberg, J., Aiken, A., Acar, U. A., et al. Quartz: superoptimization of quantum circuits. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (2022), pp. 625–640.
[62] Ying, M. Floyd–hoare logic for quantum programs. *ACM Transactions on Programming Languages and Systems (TOPLAS) 33*, 6 (2012), 1–49.
[63] Younis, E., Sen, K., Yelick, K., and Iancu, C. Qfast: Conflating search and numerical optimization for scalable quantum circuit synthesis. In *2021 IEEE International Conference on Quantum Computing and Engineering (QCE)* (2021), IEEE, pp. 232–243.
[64] Yu, N., and Palsberg, J. Quantum abstract interpretation. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (2021), pp. 542–558.
[65] Zhong-Xiao, M., and Yun-Jie, X. Controlled bidirectional quantum direct communication by using a ghz state. *Chinese Physics Letters 23*, 7 (2006), 1680.

A APPENDIX

A.1 Proofs

Proof of Theorem 4.2

Proof. We verify each rule's soundness by reasoning about the semantics of the program constructs.

Weaken. For $(n,x,y) \in h' \subset h$, there is $\langle y|[\![\{\!\{S\}\!\}(n)]\!]|x\rangle = \alpha(n,x,y)$, hence $h \triangleright S \hookrightarrow \alpha$.

Const. For any $(n,x,y) \in \mathbb{N}^3$, note $\{\!\{\textbf{const P}\}\!\}(n) = P$ and $\alpha(n,x,y) = [\![P]\!](x,y)$. This makes $\langle y|[\![\{\!\{\textbf{const P}\}\!\}(n)]\!]|x\rangle = \alpha(n,x,y)$, and $\mathbb{N}^3 \triangleright \textbf{const P} \hookrightarrow \alpha$.

Replace. For any $(n,x,y) \in h$, since $\langle y|[\![\{\!\{S\}\!\}(n)]\!]|x\rangle = \alpha(n,x,y) = \alpha'(n,x,y)$, we conclude $h \triangleright S \hookrightarrow \alpha'$.

Relabel. For any $(n,x,y) \in h$, there is $\langle y|[\![\{\!\{S\}\!\}(n)]\!]|x\rangle = \alpha(n,x,y)$. Because $\pi(n)$ is injective, for any $(n,u,v) \in \pi \circ h$, there exists x,y such that $\pi(n,x) = u$ and $\pi(n,y) = v$. Then $\langle u|[\![\{\!\{\textbf{relabel } \pi\ S\}\!\}(n)]\!]|v\rangle = \langle \pi(n,y)|[\![\{\!\{\textbf{relabel } \pi\ S\}\!\}(n)]\!]|\pi(n,x)\rangle = \langle y|[\![\{\!\{S\}\!\}(n)]\!]|x\rangle = \alpha n,x,y = \alpha'(n,u,v)$.

Seq. For any $(n,x,y) \in h$, note $\langle y|[\![\{\!\{\textbf{seq } S_1\ S_2\}\!\}(n)]\!]|x\rangle = \langle y|[\![\{\!\{S_2\}\!\}(n)]\!][\![\{\!\{S_1\}\!\}(n)]\!]|x\rangle = \sum_z \langle y|[\![\{\!\{S_2\}\!\}(n)]\!]|z\rangle\langle$ For any z, if $(n,x,z) \in h_1 \wedge \alpha_1(n,x,z) = 0$, then $\langle z|[\![\{\!\{S_2\}\!\}(n)]\!]|x\rangle = \alpha_1(n,x,z) = \alpha_1(n,x,z)\alpha_2(n,z,y)$. Similarly, if $(n,z,y) \in h_2 \wedge \alpha_2(n,z,y) = 0$, it equals to $\alpha_1(n,x,z)\alpha_2(n,z,y)$. If $(n,x,z) \in h_1 \wedge (n,z,y) \in h_2$, the term also becomes $\alpha_1(n,x,z)\alpha_2(n,z,y)$. Hence we have $h \triangleright \textbf{seq } S_1\ S_2 \hookrightarrow \alpha$.

Fix. Similarly, we denote $\textbf{fix}_k\ \pi\ P_0\ \cdots\ P_{k-1}\ S_L\ S_R$ as $\textbf{fix}_k$. For any $(i,x,y) \in h$ where $i < k$, by $(h_i,\alpha_i) \equiv_i (h,\alpha)$, we know $(i,x,y) \in h_i$ and $\alpha(i,x,y) = \alpha_i(i,x,y)$. Since $h_i \triangleright \textbf{const P}_i \hookrightarrow \alpha_i$ and $(h,\alpha) \equiv_i (h_i,\alpha_i)$, we have $\langle y|[\![\{\!\{\textbf{fix}_k\}\!\}(i)]\!]|x\rangle = \langle y|[\![P_i]\!]|x\rangle = \alpha_i(i,x,y) = \alpha(i,x,y)$. For any $(i,x,y) \in h$ such that $i \geq k$, note $\{\!\{\textbf{fix}_k\}\!\} = \{\!\{S_L\}\!\}(i); \texttt{map\_qb}(\pi(i), \{\!\{\textbf{fix}_k\}\!\}(i-1)); \{\!\{S_R\}\!\}(i)$. According to the proof for Relabel and Seq, with $(h,\alpha) \equiv_i (h_L,\alpha_L) \otimes (\pi \circ \text{pred } (h,\alpha)) \otimes (h_R,\alpha_R)$, we have $\langle y|[\![\{\!\{\textbf{fix}_k\}\!\}(i)]\!]|x\rangle = \alpha(i,x,y)$. □

Proof of Theorem 5.2

Proof. We first prove that given a SQIR program P and we have $\mathbb{N}^3 \triangleright \textbf{const P} \hookrightarrow \alpha$, then α is sparse.

Since P is a SQIR program, P is a sequence of applications of gates to fixed number of qubits and we have

$$\alpha(n,x,y) = \{\!\{P\}\!\}_{xy} \tag{A.1}$$

Without loss of generality, suppose $\{\!\{P\}\!\}$ is a unitary applied to qubits $q_0, q_1, ..., q_m, m \in \mathbb{N}$. Let function $\mathcal{X}, \mathcal{Y}$ be

$$\mathcal{X}(n,y) = \begin{cases} \{k \mid k < 2^{m+1}, k \in \mathbb{N}\}, & y < 2^{m+1} \\ \{\textbf{mk}(y,0,m) + k \mid k < 2^{m+1}, k \in \mathbb{N}\}, & \text{otherwise} \end{cases} \tag{A.2}$$

$$\mathcal{Y}(n,x) = \begin{cases} \{k \mid k < 2^{m+1}, k \in \mathbb{N}\}, & x < 2^{m+1} \\ \{\textbf{mk}(x,0,m) + k \mid k < 2^{m+1}, k \in \mathbb{N}\}, & \text{otherwise} \end{cases} \tag{A.3}$$

where $\mathbf{mk}(x, 0, m)$ means to set from the 0th bit to the mth bits (in the order from low to high) in x to 0 (e.g. $\mathbf{mk}((1111)_2, 0, 1)) = (1100)_2 = (12)_{10}$). It is easy to see that for arbitrary inputs, the size of the sets returned by $\mathcal{Y}, \mathcal{X}$ is always 2^{m+1}. Now we prove that

$$\forall n, x, y \in \mathbb{N}, \alpha(n, x, y) \neq 0 \rightarrow y \in \mathcal{Y}(n, x) \tag{A.4}$$

Formula $\alpha(n, x, y) \neq 0 \rightarrow x \in \mathcal{X}(n, y)$ can be proved in the same way. When $\alpha(n, x, y) \neq 0$:

- if $x < 2^{m+1}$, since $\{\!|P|\!\}$ is a unitary applied to qubits $q_0, q_1, ..., q_m$, $\langle x|\{\!|P|\!\}|y\rangle = 0$ for every $y \geq 2^{m+1}$. So we know that $y < 2^{m+1}$ and $y \in \mathcal{Y}(n, x)$.
- Otherwise, since $\{\!|P|\!\}$ does not change the state of qubits other than qubits $q_0, q_1, q_2, ..., q_m$, we should have $\mathbf{mk}(x, 0, m) = \mathbf{mk}(y, 0, m)$ if $\alpha(n, x, y) \neq 0$. Then we have $y \in \mathcal{Y}(n, x)$

So we have $\alpha \trianglelefteq (\mathcal{X}, \mathcal{Y})$ and we prove that α is sparse.

Next we prove that given a sparse amplitude function α and an injective mapping π, function $\pi \circ \alpha$ is sparse.

Since α is sparse, there exists functions $\mathcal{X}, \mathcal{Y}$ and we have $\alpha \trianglelefteq (\mathcal{X}, \mathcal{Y})$. Let functions $\mathcal{X}^\pi, \mathcal{Y}^\pi$ be

$$\mathcal{X}^\pi(n, y) = \{\pi^{-1}(k) \mid k \in \mathcal{X}(n, \pi(y))\} \tag{A.5}$$

$$\mathcal{Y}^\pi(n, x) = \{\pi^{-1}(k) \mid k \in \mathcal{Y}(n, \pi(x))\} \tag{A.6}$$

Then we have

$$\begin{aligned}(\pi \circ \alpha)(n, x, y) \neq 0 &\Longrightarrow \alpha(n, \pi(x), \pi(y)) \neq 0 \\ &\Longrightarrow \pi(x) \in \mathcal{X}(n, \pi(y)) \wedge \pi(y) \in \mathcal{Y}(n, \pi(x)) \\ &\Longrightarrow x \in \{\pi^{-1}(k) \mid k \in \mathcal{X}(n, \pi(y))\} \wedge y \in \{\pi^{-1}(k) \mid k \in \mathcal{Y}(n, \pi(x)\} \\ &\Longrightarrow x \in \mathcal{X}^\pi(n, y) \wedge y \in \mathcal{Y}^\pi(n, x)\end{aligned}$$

So we have

$$\forall n\ x\ y, (\pi \circ \alpha)(n, x, y) \neq 0 \rightarrow y \in \mathcal{Y}^\pi(n, x) \wedge x \in \mathcal{X}^\pi(n, y)$$

So we have , $\pi \circ \alpha \trianglelefteq (\mathcal{X}^\pi, \mathcal{Y}^\pi)$ and we prove that $\pi \circ \alpha$ is sparse.

Finally we prove that given two sparse amplitude function α_1, α_2, function $\alpha_1 * \alpha_2$ is sparse. Since α_1, α_2 are sparse, suppose we have $\alpha_1 \trianglelefteq (\mathcal{X}_1, \mathcal{Y}_1)$ and $\alpha_2 \trianglelefteq (\mathcal{X}_2, \mathcal{Y}_2)$. We also have

$$(\alpha_1 * \alpha_2)(n, x, y) = \sum_{z \in \mathcal{X}_1(n, y)} \alpha_1(n, x, z)\alpha_2(n, z, y)$$

Let functions $\mathcal{X}, \mathcal{Y}$ be

$$\mathcal{X}(n, y) := \{\mathcal{X}_1(n, z) | z \in \mathcal{X}_2(n, y)\} \tag{A.7}$$

$$\mathcal{Y}(n, x) := \{\mathcal{Y}_2(n, z) | z \in \mathcal{Y}_1(n, x)\} \tag{A.8}$$

Then we have

$$(\alpha_1 * \alpha_2)(n, x, y) \neq 0 \Longrightarrow \exists z, \alpha_1(n, x, z) \neq 0 \wedge \alpha_2(n, z, y) \neq 0 \tag{A.9}$$

$$\Longrightarrow \exists z, z \in \mathcal{Y}_1(n, x) \wedge z \in \mathcal{X}_2(n, y) \wedge x \in \mathcal{X}_1(n, z) \wedge y \in \mathcal{Y}_2(n, z) \tag{A.10}$$

$$\Longrightarrow x \in \mathcal{X}(n, y) \wedge y \in \mathcal{Y}(n, x) \tag{A.11}$$

So we have

$$\forall n\ x\ y, (\alpha_1 * \alpha_2)(n, x, y) \neq 0 \rightarrow x \in \mathcal{X}(n, y) \wedge y \in \mathcal{Y}(n, x) \tag{A.12}$$

So $\alpha_1 * \alpha_2 \trianglelefteq (\mathcal{X}, \mathcal{Y})$ and we prove that $\alpha_1 * \alpha_2$ is sparse. □

A.2 ISQIR Programs in Case Study

Adder Program. The Qiskit full adder program in Fig 8 is compiled from the ISQIR program S_a below

$$
\begin{aligned}
S_a &:= \textbf{fix}_1\ \pi\ (\textbf{const}\ \text{I})\ (\textbf{const}\ \text{I})\ S_R \\
\pi &:= (\textbf{shift}\ 1\ n\ 1) \cdot (\textbf{shift}\ n+1\ 2n\ 2) \\
S_R &:= \textbf{const}\ \text{CCX 1 n+1, 2n+1}; \textbf{const}\ \text{CNOT 1 n+1}; \textbf{const}\ \text{CCX n+1 0 2n+1}; \\
&\quad\ \textbf{const}\ \text{CNOT n+1 0}; \textbf{const}\ \text{CNOT 1 n+1}; \textbf{const}\ \text{SWAP 0 2n+1}
\end{aligned}
$$

The Qiskit Cucarro adder program in Fig **??** is compiled from the ISQIR program S_c below

$$
\begin{aligned}
S_c &:= \textbf{fix}_1\ \pi\ (\textbf{const}\ \text{I})\ S_L\ S_R \\
S_L &:= \textbf{const}\ \text{CNOT n+1 1}; \textbf{const}\ \text{CNOT n+1 0}; \textbf{const}\ \text{Toffoli 0 1 n+1} \\
S_R &:= \textbf{const}\ \text{Toffoli 0 1 n+1}; \textbf{const}\ \text{CNOT n+1 0}; \textbf{const}\ \text{CNOT 0 1} \\
\pi &:= (\textbf{shift}\ 1\ n\ 1) \cdot (\textbf{shift}\ n+1\ 2n\ 2)
\end{aligned}
$$

Quantum Fourier Transfrom. The Qiskit program that generates state $|z_n\rangle$ in Fig 9 is compiled from the ISQIR program S_z below

$$
\begin{aligned}
S_z &:= \textbf{fix}_1\ w\ (\textbf{const}\ \texttt{H}\ \texttt{0})\ (\textbf{const}\ \text{I})\ S_R \\
S_R &:= \textbf{const}\ (\text{CRZ}(\frac{2 \cdot \pi}{2^n})\ 0\ \text{n}) \\
w &:= \textbf{shift}\ 0\ n\ 1
\end{aligned}
$$

The Qiskit QFT program in Fig 10 is compiled from the ISQIR S_Q below

$$
S_Q := \textbf{fix}_1\ Id\ (\textbf{const}\ \texttt{H}\ \texttt{0})\ (\textbf{const}\ \text{CRZN})\ (\textbf{const}\ \text{I})
$$