A Case for Synthesis of Recursive Quantum Unitary Programs

HAOWEI DENG*, University of Maryland, College Park, USA
RUNZHOU TAO*, Columbia University, USA
YUXIANG PENG, University of Maryland, College Park, USA
XIAODI WU, University of Maryland, College Park, USA

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 *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, the first 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 10 quantum unitary programs including quantum arithmetic programs, quantum eigenvalue inversion, quantum teleportation and Quantum Fourier Transformation, which can be readily transpiled to executable programs on major quantum platforms, e.g., Q#, IBM Qiskit, and AWS Braket.

CCS Concepts: • **Software and its engineering** → **Domain specific languages**; **Domain specific languages**; • **Hardware** → *Quantum computation.*

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

ACM Reference Format:
Haowei Deng, Runzhou Tao, Yuxiang Peng, and Xiaodi Wu. 2024. A Case for Synthesis of Recursive Quantum Unitary Programs. *Proc. ACM Program. Lang.* 8, POPL, Article 59 (January 2024), 30 pages. https://doi.org/10.1145/3632901

1 INTRODUCTION

Quantum programming is a key step in enabling the various application of quantum computing such as factorization, simulation of physics and optimization. However, programming quantum computers is hard due to unintuitive quantum mechanics. To help ease the programming of quantum computers, circuit synthesis techniques have been proposed to automatically generate quantum circuits [Amy et al. 2013; de Brugiere 2020; Kang and Oh 2023; Kitaev 1997; Saeedi et al. 2011; Shende et al. 2006; Younis et al. 2021].

Unfortunately, these synthesis frameworks underperform when the number of qubits of the quantum circuit to synthesize is as large as 5. For example, QFAST [Younis et al. 2021], a recent quantum circuit synthesis framework, can only synthesize QFT and adder circuit up to 5 qubits. And

*Both authors contributed equally to the paper.

Authors' addresses: Haowei Deng, University of Maryland, College Park, College Park, MD, USA, hwdeng@umd.edu; Runzhou Tao, Columbia University, New York, NY, USA, runzhou.tao@columbia.edu; Yuxiang Peng, University of Maryland, College Park, College Park, MD, USA, ypeng15@umd.edu; Xiaodi Wu, University of Maryland, College Park, College Park, MD, USA, xwu@cs.umd.edu.

ACM 2475-1421/2024/1-ART59
https://doi.org/10.1145/3632901

QSyn [Kang and Oh 2023], a quantum circuit synthesis method based on user-supplied components, needs an average time of 687.5 seconds for solving 4-qubit problems and fails to synthesize a 6-qubit circuit within one hour. These methods cannot scale with the rapid development of qubit numbers in hardware, with more than 1000 qubits by the end of 2023 estimated by IBM [Gambetta 2022]. Moreover, synthesizing a quantum circuit will even fail at the start because it is impossible to write the exponential-sized matrix specifying the synthesis goal. For example, QSyn requires 16 input-output state vector pairs as the specification for a 4-qubit Toffoli circuit. Additionally, the circuit generated by the synthesizer is hard to understand by humans, which prevents potential human customization to the circuit after synthesis.

In this work, we propose QSynth, the first synthesis framework for inductive quantum programs. In contrast to previous frameworks focusing on circuits, the synthesis target of QSynth is *inductively-defined families of quantum circuits* without mid-circuit measurements (i.e. unitaries). QSynth can exploit the inductive structure of quantum programs and, compared to previous circuit synthesis methods, 1) generate quantum circuits with an arbitrary number of qubits, 2) allow user to use a single input-output style specification for synthesis, and 3) produces more readable and structured quantum program code that are easy to customize by human, as illustrated by Figure 1.

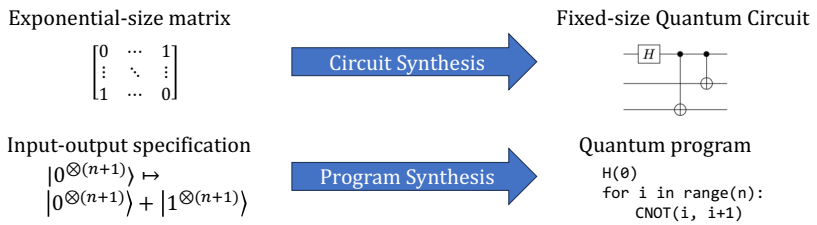


Fig. 1. Circuit synthesis vs. program synthesis. Circuit synthesis takes an exponential-sized matrix or state vector pairs as input and generates a fixed-size circuit, while program synthesis takes in an input-output specification and generates a program denoting a family of circuits for any input size.

To enable the synthesis of inductive quantum programs, there are three challenges. First, the synthesis framework requires a representation of the specification of quantum programs. Previous methods such as quantum Hoare triple [Ying 2012], path-sum [Amy 2018], and tree automaton [Chen et al. 2023] are all defined on fixed-dimension quantum systems and thus cannot be directly applied. Naive extensions of these representations with a variable qubit size do not work because symbolically representing matrices and automata is hard, limiting their usage for synthesis and verification. We introduce a specification language in QSynth, which enables intuitive input-output style specification for quantum programs and supports all path-sum quantum states with an arbitrary number of qubits. We also propose the *hypothesis-amplitude* ($h-\alpha$) specification that uses two functions to specify quantum programs. The specification written in QSynth-spec language will be compiled into $h-\alpha$ specification to use in the verification step, which avoids SMT-unfriendly matrices (or graphical representations like automata).

The second challenge is to find a subset of quantum programs that is both expressive enough and efficient to synthesize. Previous programming languages for verifying quantum programs are either low-level circuit languages that do not support inductive structure (e.g., SQIR [Hietala et al. 2021]) or high-level languages that do not have a detailed structure of unitaries (e.g., Quantum-while language used in Quantum Hoare Logic [Ying 2012]). We design the Inductive-SQIR (ISQIR) language, an extension of SQIR that supports inductively defined quantum programs. We also

define a Hoare-type verification logic that verifies an ISQIR program with respect to an $h - \alpha$ specification.

Finally, the synthesizer needs automated verification of inductive quantum programs. This requires reasoning about matrices, complex numbers, and formulas with a non-fixed number of terms, all of which have very limited support in SMT solvers. To solve this challenge, we define *parameterized path-sum amplitudes*, which, together with a sparsity constraint, can be efficiently encoded in SMT solvers. We further show that this set of expressions is expressive enough to specify and synthesize many quantum programs.

We evaluate QSynth on 10 inductive quantum programs including state preparation, arithmetic and textbook quantum algorithm procedures. We showcase that QSynth can synthesize practical quantum programs including quantum adders [Cuccaro et al. 2004; Feynman 1985], a quantum subtractor, the eigenvalue inversion for HHL [Lloyd 2010] algorithm, quantum teleportation [Bennett et al. 1993] and Quantum Fourier Transform [Coppersmith 2002]. All synthesis processes succeed in 5 minutes, while many of the programs cannot be synthesized by previous methods when the number of qubits is larger than 6. A further investigation of synthesized programs shows that QSynth successfully captures the inductive structure of the targeting problem and produces better programs than human-written programs in Qiskit.

Contributions. Our contributions in this paper are multi-folded.

- We propose QSynth, the first synthesis framework for inductively-defined quantum unitary circuit family.
- We introduce the QSynth-spec language that enables input-output style specification for inductive quantum programs, and the hypothesis-amplitude ($h - \alpha$) specification for scalable verification of quantum programs.
- We develop the syntax and the semantics of the inductive SQIR (ISQIR) language that supports recursively defined families of quantum unitary circuits, and a Hoare-type logic for proving the correctness of ISQIR program, with an $h - \alpha$ specification as predicates.
- We design Parameterized path-sum amplitude (PPSA) function, which leads to the efficient encoding of the verification process into SMT instances.
- We evaluate QSynth with a benchmark of 10 quantum programs, and show that QSynth is able to synthesize practical quantum programs.

2 QUANTUM PRELIMINARIES

In this section, we introduce the background knowledge about quantum program. We recommend readers to refer Nielsen and Chuang [2010] for more details about quantum computing.

2.1 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 $|\beta|^2$.

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

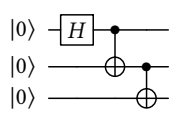
$2^2 = 4$ length vector where each component corresponds to (the square root of) the probability of measuring $|00\rangle$, $|01\rangle$, $|10\rangle$, and $|11\rangle$, respectively. We may also write these four kets as $|0\rangle$, $|1\rangle$, $|2\rangle$, $|3\rangle$. Sometimes a multi-qubit state 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*.

2.2 Quantum Programs

Quantum programs are composed of a series of *quantum operations*, each of which acts on a subset of qubits. Quantum operations can be expressed as matrices, and their application to a state is 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 $\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$. 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 2(a). In these circuits, each horizontal wire represents a qubit and boxes on these wires indicate quantum operations, or *gates*. The circuit in Fig 2(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*. 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 2(a) is shown in Fig 2(c), where I is the 2×2 identity matrix.



(a) Quantum circuit

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

(b) Unitary SQIR

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

(c) Matrix expression

Fig. 2. Example quantum program: 3-qubit GHZ state preparation.

2.3 Unitary SQIR

SQIR [Hietala et al. 2021] is a simple quantum language embedded in the Coq proof assistant. SQIR's *unitary fragment* is a sub-language for expressing programs consisting of unitary gates.

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). In Coq, a unitary SQIR program `U` hence has type `ucom g n`. `U` has type `ucom g n`, where g identifies the gate set and n is the size of the global register.

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 2(a).

$$[\![P_1;\ P_2]\!]_d = [\![P_2]\!]_d \times [\![P_1]\!]_d;$$

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

Fig. 3. 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
    | S n' ⇒ ghz n'; CNOT n' n
  end.
```

Semantics. The semantics of unitary SQIR is shown in Fig 3. A program P is well-typed if every gate's index arguments are within the bounds of the global register and no index is repeated. The program's semantics follows from the composition of the matrices that correspond to each of the applications of its unitary gates. 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 $apply_3$ 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.

Suppose that M_1 and M_2 are the matrices corresponding to unitary gates P_1 and P_2, which we want to apply to a quantum state vector $|\psi\rangle$. Matrix multiplication is associative, so $M_2(M_1|\psi\rangle)$ is equivalent to $(M_2M_1)|\psi\rangle$. Moreover, multiplying two unitary matrices yields a unitary matrix. As such, the semantics of SQIR program $P_1;\ P_2$ is naturally described by the unitary matrix M_2M_1. Fig 2(b) shows an example unitary SQIR program for the circuit in Fig 2(a).

2.4 Path-sum Representation

Path-sum, proposed by recent works on quantum program verification [Amy 2018; Chareton et al. 2020], is a representation for describing quantum states based on Feynman's *path integral* formalism of quantum mechanics, which is widely applied to circuit simulation [Bravyi and Gosset 2016; Koh et al. 2017] and optimization [Amy et al. 2018, 2014; Amy and Mosca 2019]. The idea of this formalism is to describe a quantum state's amplitude by an integral over all paths leading to that state. In practice, a discrete sum-over-path technique rather than integral is typically used [Amy 2018; Amy et al. 2014; Amy and Mosca 2019; Bacon et al. 2008; Bravyi and Gosset 2016; Chareton et al. 2020; Dawson and Nielsen 2005; Koh et al. 2017; Montanaro 2017]. We can describe a sum-over-path abstractly as a discrete set of paths $T \in \mathbb{Z}_2^m$, together with an amplitude function ψ and a state transformation f, both depending on specific path y, to represent a unitary U:

$$U : |x\rangle \mapsto \sum_{y \in T} \psi(x, y)|f(x, y)\rangle.$$

All representations based on the sum-over-path in these previous works[Amy 2018; Bacon et al. 2008; Chareton et al. 2020; Dawson and Nielsen 2005; Koh et al. 2017; Montanaro 2017] share in common that the amplitude $\psi(x, y)$ of all possible paths have the same *magnitude* and only the *phases* are different. These forms of the same *magnitude* but different *phase* are so typical in many quantum algorithms that they can be used as a succinct representation in the verification, which makes them useful for synthesis purposes.

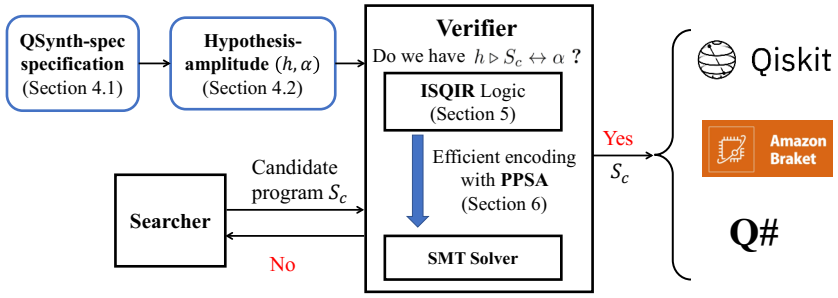
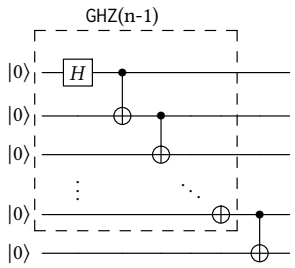


Fig. 4. QSynth Overview.

$S_{GHZ} :=$ **fix**$_1$ *Id*
const H 0
const ID
const 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

Fig. 5. $(n+1)$-qubit GHZ state preparation programs in different programming languages. (a) ISQIR program S_{GHZ}. (b) The quantum circuit represented by S_{GHZ}. (c) Qiskit function compiled from S_{GHZ}. Statement `circ.cx` in Qiskit means appending a CNOT gate to the circuit. Qiskit's semantic requires the program always use class `QuantumCircuit(n)` to initialize a circuit. So QSynth compiler will wrap program S with an outside function GHZ to initialize the circuit.

3 OVERVIEW

The workflow of QSynth is shown in Fig 4. First, the user writes an input-output style specification of the program to be synthesized using the QSynth-spec language. Then, QSynth will translate the specification into a hypothesis-amplitude (h-α) pair for later verification. Meanwhile, QSynth will invoke a syntax-guided program searcher [Gulwani et al. 2017] to generate all possible candidate programs written in the ISQIR language within a given search space. For each candidate ISQIR program, QSynth will use the unitary ISQIR logic to verify whether the program satisfies the (h-α) specification. The verification process will be done in an SMT solver in which all the numbers are encoded in the form of parameterized path-sum amplitudes (PPSA). If the verification process succeeds, a correct program is synthesized and the program can be translated by QSynth's ISQIR compiler into commercial quantum programming languages including Qiskit, Q# and Braket, as shown in Figure 5. If the verification fails, the searcher will try the next candidate program.

Next, we will walk through QSynth's components using the synthesis of the GHZ state preparation program as an example.

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. [1989] and is widely used in quantum information, e.g., [Hillery et al. 1999; Liao et al. 2014; Xia et al. 2006; Zhong-Xiao and Yun-Jie 2006]. 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 synthesize an ISQIR program S such that $[\![\{\!\{S\}\!\}(n)]\!]$, the instantiation of S with $n \in \mathbb{N}$ where $\mathbb{N}$ denotes the natural number in the following paper, 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. Then, a direct matrix representation that might require $2^{n+1} \times 2^{n+1}$ is less desirable.

In QSynth, the specification is given in an input-output manner using the QSynth-spec language in the form of GHZ : $|0_{n+1}\rangle \mapsto |0_{n+1}\rangle \uplus |(2^{n+1}-1)_{n+1}\rangle$, where $|a\rangle \uplus |b\rangle$ means the normalized sum $(|a\rangle + |b\rangle)/\sqrt{2}$. Then, the input-output style specification is compiled into the following *hypothesis-amplitude* $(h-\alpha)$ *specification* (Definition 4.1) for later verification. An instantiation of $(h-\alpha)$ to the GHZ target program is given as follows:

$$h := \{(n,x,y) \mid x = 0\}, \qquad \alpha_{GHZ}(n,x,y) := \frac{1}{\sqrt{2}}\delta(y = 0 \vee y = 2^{n+1}-1), \tag{3.1}$$

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 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 (3.1). 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 (3.1).

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 QSynth searcher has identified a candidate S_{GHZ}, same as Fig 5 (a), on the left of Fig 6. The program S_{GHZ} is constructed by a FIX syntax with subprograms S_0 = `H 0`, S_L = `ID` and S_R = `CNOT n-1 n`, which is a recursive program with the base case $S_{GHZ}(0) := S_0$ and the inductive case, $S_{GHZ}(n) := S_L(n); S_{GHZ}(n-1); S_R(n)$. This ISQIR program is equivalent to the recursive Qiskit program in Figure 5c. The FIX syntax, similar to the fixpoint in Coq, enables inductive structures in ISQIR programs.

QSynth verifier leverages the newly developed unitary ISQIR logic (Section 5.2) 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 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.

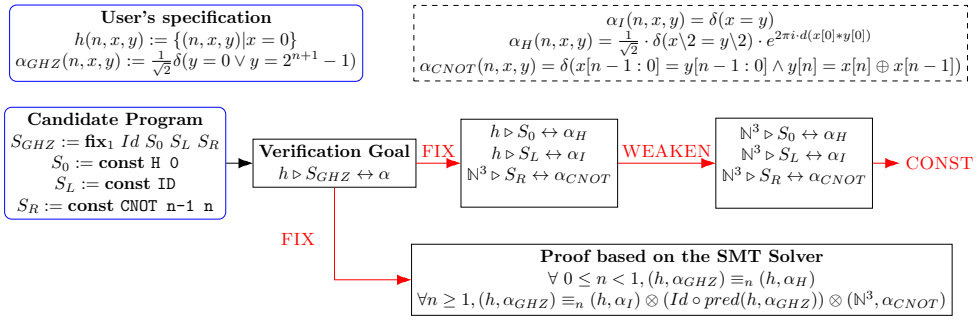


Fig. 6. An example of synthesizing $n+1$-qubit GHZ state preparation program.

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 6) 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 6). After applying the FIX rule 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 6) are predefined in QSynth and can be verified directly using the CONST rule. 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 $\delta(x\backslash 2 = y\backslash 2)$, where $x\backslash 2$ and $y\backslash 2$ are integer division (e.g. $10\backslash 3 = 3$), 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. After this verification step, the synthesis terminates and QSynth compiles S_{GHZ} into programs as shown in Fig 5.

SMT solving with PPSA. We continue with the two formulas in the bottom right corner box in Fig 6 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 quantum program verification [Amy 2018; Chareton et al. 2020], QSynth restricts α into a succinct path-sum representation yet with rich enough expressiveness (elaborated on in Section 6.1), called the parameterized path-sum amplitude (PPSA) in Definition 6.1. In PPSA representation, the non-zero amplitudes over x, y share the same magnitude, which depends on n, but can have different phases. Thus, instead of representing a general amplitude function $\alpha(n,x,y)$, it suffices to represent $\alpha(n,x,y)$ by components. For instance, α_H, α_{GHZ} defined in Fig 6, 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: $\delta(y = 0 \vee y = 2^{n+1} - 1)$ for α_{GHZ} and $\delta(x\backslash 2 = y\backslash 2)$ for α_H.

- 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 $(h_1, \alpha_1) \otimes (h_2, \alpha_2)$ calculates the one when composing two ISQIR programs together, both formally in Definition 5.4. In this example, the composition with (h, α_I) is trivial since α_I represents an identity matrix, and the second formula becomes

$$\begin{gathered}\forall n \geq 1, \forall x\ y \in \mathbb{N}, (h, \alpha_{GHZ}) \equiv_n (h, \alpha'), \\ \text{where } \alpha'(n, x, y) = \sum_{z \in \mathbb{N}} \alpha_{GHZ}(n-1, x, z)\alpha_{CNOT}(n, z, y), \\ \alpha_{CNOT}(n, x, y) = \delta(x[n-1:0] = y[n-1:0] \wedge y[n] = x[n] \oplus x[n-1]).\end{gathered} \tag{3.2}$$

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 (3.2) 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 5). 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 6.2), 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. We only apply this sparsity constraint to SQIR programs predefined in QSynth, and *non-sparse* functions can be generated as synthesis specifications. 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

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

Here the expression $y \oplus (y[n-1] \text{«} n)$ 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 $z = y \oplus (y[n-1] \text{«} n)$. With this sparsity of α_{CNOT}, we can simplify the formula (3.2) to

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

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

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

Organization. In Section 4 we describe the QSynth's specification language and hypothesis-amplitude specification. In Section 5 we introduce the ISQIR programming language for the inductive quantum circuit family, and its associated Hoare-type logic, In Section 6 we introduce the PPSA encoding. In Section 7 we discuss the implementation and the evaluation of QSynth on a benchmark of 10 programs. In Section 8 we discuss the related work. In Section 9 we discuss the limitation of QSynth and future work. In Section 10 we give the conclusion.

4 SPECIFICATION

In this section, we explain how specifications are provided and processed in QSynth. Users provide the input-output specification in the QSynth-spec language (defined in Section 4.1). Then, the specification will be compiled into a hypothesis-amplitude pair (defined in Section 4.2) as the predicate for later verification. We describe the compilation process in Section 4.3

4.1 QSynth-spec Interface

To allow users to give the synthesis specification more intuitively, we design language QSynth-spec shown in Fig 7. Programmers provide the synthesis specification in QSynth-spec, and QSynth will compile it to the corresponding hypothesis-amplitude specification.

(Spec)	$\mathcal{S}$	::=	$\mathcal{I} \mapsto O$
(Input)	$\mathcal{I}$	::=	$\lvert c_l\rangle \mid \lvert v[\ell]\rangle \mid \mathcal{I}_1 \otimes \mathcal{I}_2$
(Output)	O	::=	$\lvert E\rangle \mid e^{i\cdot E} \cdot O' \mid O_1 \otimes O_2 \mid e^{i\cdot E_1}\lvert c_1\rangle \uplus e^{i\cdot E_2}\lvert c_2\rangle \mid \biguplus_{z\in\{0,1\}^l} \delta(B) \cdot e^{-2\pi i\cdot E}\lvert z\rangle$
(VarExp)	E	::=	$c_\ell \mid v \mid \textbf{uop}\ E' \mid E_1\ \textbf{bop}\ E_2$
(BoolExp)	B	::=	$E_1\ \textbf{rop}\ E_2 \mid B_1 \wedge B_2 \mid B_1 \vee B_2$
(Length)	ℓ	::=	$c \mid \mathsf{n} \mid \textbf{uop}\ \ell' \mid \ell_1\ \textbf{bop}\ \ell_2$
(Variable)	v	$\in$	Variables *(Constant)* $c \in \mathbb{N}$

Fig. 7. QSynth-spec syntax. **uop** and **bop** are common unary operators and binary operators (e.g. + - * /) . **rop** are common relation operators (e.g. ==, !=, >, <). $\delta(B)$ is a function that returns 1 if B is True and returns 0 otherwise. $\uplus$ means normalized summation.

A QSynth-spec specification is given in the form $Input \mapsto Output$. $Input$ indicates the input quantum state to the desired unitary. It is a ket expression constructed by l-bit constant numbers c_l, a $l-$bit variable $v[l]$, or the tensor product of two quantum states. Programmers can arbitrarily declare variables in the $Input$. For example, consider the $Input$ specification for a n-bit quantum adder: $|A\rangle|B\rangle|0\rangle^n \mapsto |A\rangle|B\rangle|A+B\rangle$.

$$|0\rangle|A[n]\rangle|B[n]\rangle|0_n\rangle \mapsto |c_0\rangle|A\rangle|B\rangle|A+B\rangle \tag{4.1}$$

where c_0 is the carry bit of $A+B$. We simplify $|\phi\rangle \otimes |\psi\rangle$ as $|\phi\rangle|\psi\rangle$ and omit the length specification for the one-bit state $|0\rangle, |1\rangle$. This specification indicates the qubit q_0 is initialized as $|0\rangle$ to hold the carry bit; qubits $q_1 \sim q_n$ and qubits $q_{n+1} \sim q_{2n}$ store two n-bit numbers $A[n], B[n]$; qubit $q_{2n+1} \sim q_{3n}$ are initialized to $|0\rangle^n$ to store the sum of $A[n], B[n]$.

$Output$ suggests the state transformed from $Input$ by the target unitary program. $Output$ can be constructed by a variable expression E, a state shifted by the phase e^{iE}, or the tensor product of two

states. The variables that appear in E are bound: they can only be a variable declared in *Input* or the sum variable y when it is in the sum scope. Output can also be constructed as the superposition state with the same magnitude but different phase by $e^{i\cdot E_1}|c_1\rangle \uplus e^{i\cdot E_2}|c_2\rangle$ and $\biguplus_{z\in\{0,1\}^l} \delta(B)\cdot e^{-2\pi i\cdot E}|z\rangle$. For example, consider the specifications for n-qubit GHZ state program and QFT program.

$$\mathrm{GHZ}_n : |0_n\rangle \mapsto |0_n\rangle \uplus |(2^n-1)_n\rangle \qquad \mathrm{QFT}_n : |x[n]\rangle \mapsto \left\biguplus_{y\in\{0,1\}^n} e^{-2\pi i\cdot\frac{xy}{2^n}}\right. \tag{4.2}$$

Programmers can omit the amplitude normalization term, which will be calculated by QSynth during the compilation from QSynth-spec to the hypothesis-amplitude specification.

4.2 Hypothesis-amplitude Specification for Verification

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 [Ying 2012] 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 quantum circuit families, 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 quantum circuits.

Definition 4.1. *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, S is a quantum circuit family parameterized with a natural number n, 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

$$h \triangleright S \leftrightarrow \alpha \iff \forall (n, x, y) \in h,\ \langle y|[\![\{\!\{S\}\!\}(n)]\!]|x\rangle = \alpha(n, x, y). \tag{4.3}$$

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. [1] 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, efficiently encoding into SMT instances are only known in restricted cases of h, α as discussed in Section 6.

4.3 From QSynth-spec to Hypothesis-amplitude Specification

Given a QSynth-spec specification $I \mapsto O$, QSynth compiles it to the hypothesis-amplitude specification in two steps: (1) generates the corresponding hypothesis h and a variable map Π based on I, where Π maps variables claimed in I to the qubits; (2) generates the corresponding amplitude function α based on O and Π.

Generate Π and h from I. QSynth first calculates the total number of input qubits by adding up the length specifications of constants and variables in the input. This number depends on the parameter n. Then, QSynth assigns indexes from low to high for each variable and constant in I, according to the order they appear in I. The assignment of the variables is included in the variable map Π. For each constant number c represented by qubits $q_a \sim q_b$, QSynth adds expression $x[b:a] = c$ into the hypothesis h. For example, consider the specification for the quantum adder circuit $I : |0\rangle|A[n]\rangle|B[n]\rangle|0_n\rangle$, the variable map Π and the hypothesis h generated by QSynth are:

$$\Pi = \{A \mapsto q_1 \sim q_n, B \mapsto q_{n+1} \sim q_{2n}\}, \qquad h = \{x[0] = 0, x[3n : 2n+1] = 0\} \tag{4.4}$$

Generate α from O and Π. QSynth will then compute the value $\alpha(n, x, y)$ from the output O and the variable map Π. QSynth first calculates the number of qubits. Then, it calculates the normalization factor k which is the total number of qubits in the summed variable plus the number of additions. Then, QSynth evaluates the output into a basis state $eval_{\Pi,x,y}(O)$ by replacing input variables with slices of x according to Π and summed variables with corresponding slices of y together with a phase factor $e^{i\phi(O,\Pi,x,y)}$. Finally, the value of $\alpha(n, x, y)$ will be $1/\sqrt{2^k} e^{i\phi(O,\Pi,x,y)} \cdot \delta(eval_{\Pi,x,y}(O) = y)$. If the output contains additions, there will be a basis state and a phase factor for each term and the α value will be the sum of their α. For example, the α's of GHZ and QFT in Equation 4.2 are

$$\alpha_{GHZ}(n, x, y) = \frac{1}{\sqrt{2}}\delta(y = 0 \vee y = 2^{n+1} - 1), \qquad \alpha_{QFT}(n, x, y) = \frac{1}{\sqrt{2^n}} e^{2\pi i x \cdot y / 2^n} \tag{4.5}$$

5 INDUCTIVE SQIR AND ITS LOGIC

QSynth's goal is to synthesize programs with inductive structures. To that end, we extend the existing intermediate representation SQIR [Hietala et al. 2021] into a language called Inductive SQIR (ISQIR) that defines a family of quantum circuits inductively in Section 5.1. In Section 5.2, we also develop a logic for reasoning about an ISQIR program with respect to an $h - \alpha$ specification.

5.1 Inductive SQIR (ISQIR)

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

Definition 5.1 (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.

[1] 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.

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

$$\llbracket \textbf{\textit{const}}\ P \rrbracket(n) = P; \qquad \llbracket \textbf{\textit{seq}}\ S1\ S2 \rrbracket(n) = \llbracket S1 \rrbracket(n); \llbracket S2 \rrbracket(n);$$
$$\llbracket \textbf{\textit{relabel}}\ \pi\ S \rrbracket(n) = \texttt{map\_qb}(\pi(n), \llbracket S \rrbracket(n));$$
$$\llbracket \textbf{\textit{fix}}_k \rrbracket(n) = \begin{cases} P_n, & n < k \\ \llbracket S_L \rrbracket(n); \texttt{map\_qb}(\pi(n), \llbracket \textbf{\textit{fix}}_k \rrbracket(n-1)); \llbracket S_R \rrbracket(n); & else \end{cases}$$

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 $\llbracket P \rrbracket$, 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 5.1. *As an example, recall the GHZ program written in ISQIR syntax:*

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

*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 5.3 (Permutation Syntax).

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

*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, 2 to 3, and 3 to 4.

5.2 Unitary ISQIR Logic

We develop *unitary ISQIR logic* shown in Fig 8 to reason about ISQIR program's semantics with respect to the hypothesis-amplitude specification (h, α) with helper functions in Def 5.4. The soundness is formally proven in Theorem 5.1, whose proof is postponed to Appendix A.1 in the supplementary material.

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 5.4. *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 5.1. *The rules of unitary ISQIR logic in Fig 8 are sound.*

6 EFFICIENT ENCODING BY PARAMETERIZED PATH-SUM AMPLITUDE

In this section we explain how to encode the hypothesis-amplitude triple introduced In Definition 4.1 and the functions and relations in Definition 5.4 into the SMT instance.

6.1 Encoding The Hypothesis-amplitude Triple

The hypothesis-amplitude triple in Definition 4.1 includes a hypothesis set h and an amplitude function α, which will be encoded separately.

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

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

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

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

Fig. 8. The unitary ISQIR logic.

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,

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

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 6.1. *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 of n and it decides the magnitude of all 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.*
- *$\{V_i\}_m$ 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. 9 shows the syntax we allow to construct the expressions β, B, V in a PPSA function. "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 arithmetic operators (i.e. + - * \ %) and bit-wise operators (i.e. & | $\oplus$ « »). All these operators 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,

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

Fig. 9. Syntax of the PPSA function.

let $V_1 = (6)_{10} = (110)_2$, we have $V_1[0] = 0, V_1[2:1] = (11)_2 = 3$. Z3 SMT solver supports all these syntaxes.

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 5.4 into the SMT solver. This will be discussed in Section 6.2.

Our design for the PPSA function is inspired by Feynman's *sum-over-path* formalism described in Section 2.4 which has inspired many quantum state representations. 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 express most famous quantum algorithms (e.g., [Amy 2018; Chareton et al. 2020]), 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 h-α triple while restricting the amplitude function to PPSA are listed in Table 1. More examples are provided in Section 7.

Table 1. Examples of Unitary Operators represented by the H-α Specification.

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

6.2 Encoding Reasoning Based on ISQIR Logic

To enable SMT-based automation in reasoning, one needs to encode the functions and the equivalence relations of h, α defined in Definition 5.4 into SMT instances. In the cases of the relabeling functions $(\pi \circ h), (\pi \circ \alpha)$, 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.

Encoding the composition function. Recall the $\alpha_1 * \alpha_2$ function from Definition 5.4:

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

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., [Amy 2018]) 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 6.2. *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 6.1. *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

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

*Intuitively, $\mathcal{X}, \mathcal{Y}$ are constructed in this way since **const** H 0 only modifies the 0th qubit.*

Now we explain how to encode α 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 6.1 (α-sparsity). *Suppose $\alpha, \alpha_1, \alpha_2$ are amplitude functions:*

- *Let P be a unitary SQIR program and $\mathbb{N}^3 \triangleright \textbf{const}\ P \leftrightarrow \alpha$, 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 9. So when composing two amplitude functions α_1, α_2, there is always at least one sparse function and our composition strategy can work.

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

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

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

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

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

7 EXPERIMENTAL CASE STUDIES

We demonstrate six additional case studies and provide the output Qiskit programs compiled from synthesized ISQIR programs for better illustration. Then in Section 7.7, we compare the performance of QSynth against the previous quantum circuit synthesis frameworks, QFAST [Younis et al. 2021] and Qsyn [Kang and Oh 2023].

7.1 Quantum Adder

Motivation and Background. Quantum circuits for arithmetic operations are required for quantum algorithms. One important example is the adder circuit. Feynman [1985] first proposes the quantum *full adder* circuit to implement $|0\rangle|A\rangle|B\rangle|0\rangle^{\otimes n} \rightarrow |c_0\rangle|A\rangle|B\rangle|A+B\rangle$ where A, B 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 $A + B$. To reduce the qubit usage, Cuccaro et al. [2004] 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) with the specification in Equation 4.1. QSynth generates a program S_a as shown in Fig 10(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.

```
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
```

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

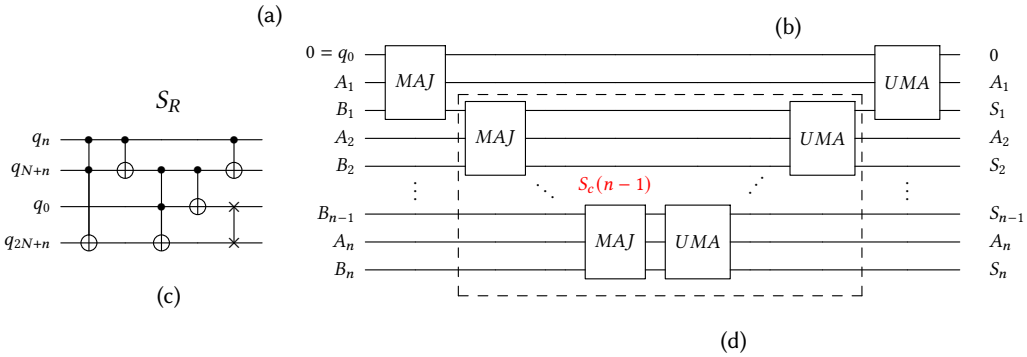


Fig. 10. (a)(c) Full quantum adder program S_a written in Qiskit. `circ.ccx(a,b,c)` means appending a Toffoli gate that controlled by qubit q_a, q_b on qubit q_c to the circuit `circ`. The circuit S_R is exactly a one-bit quantum full adder circuit. (b)(d) Cuccaro's quantum ripple-carry adder program written in Qiskit language.

Cuccaro's Adder Synthesis. To reduce the number of qubits in the circuit, we want to synthesize an in-place adder. The specification is given as:

$$|0\rangle|A[n]\rangle|B[n]\rangle \mapsto |c_0\rangle|A\rangle|A+B\rangle \tag{7.1}$$

With this specification, QSynth generates program S_c shown in Fig 10(b)(d). We use Cuccaro's MAJ and UMA circuit structures as predefined modules in the synthesis. QSynth flattens these two modules in the compilation process.

7.2 Quantum Subtractor

Another important quantum arithmetic operation is the quantum subtractor. A classical n-bit subtractor is usually implemented based on the two's complement theory (i.e. $B - A = B + \bar{A} + 1$ where $\bar{A}$ flips each bit in A), which is also used by many existing quantum libraries (e.g. QLib [Lin et al. 2014],QPanda [Dou et al. 2022]). However, this method requires additional ancilla qubits to build the "+1" operation. To reduce the qubit usage, we let QSynth synthesize a n-bit subtractor using the same number of qubits in the n-bit ripple adder.

Synthesis with QSynth. We let QSynth synthesize a program with the specification below.

```
def RippleSubtractor(N):
    circuit=QuantumCircuit(2*N+1)
    def S(circ, n):
        if(n==0):
            circ.id(0)
        else:
            circ.x(2*N-n+1)
            circ.cx(N-n+1,2*N-n+1)
            circ.cx(N-n+1,0)
            circ.ccx(0,2*N-n+1,N-n+1)
            S(circ,n-1)
            circ.ccx(0,2*N-n+1,N-n+1)
            circ.cx(N-n+1,0)
            circ.cx(0,2*N-n+1)
            circ.x(2*N-n+1)
    S(circuit,N)
    return circuit
```

Fig. 11. N-bit ripple subtractor program written in Qiskit language.

```
def ConditionalAdder(N):
    circ=QuantumCircuit(2*N+2)
    def S(circ, n):
        if(n==0):
            circ.id(0)
        else:
            circ.ccx(0,N-n+2,2*N-n+2)
            circ.ccx(0, N-n+2,N-n+1)
            circ.ccx(N-n+1,2*N-n+2,N-n+2)
            S(circ,n-1)
            circ.ccx(N-n+1,2*N-n+2,N-n+2)
            circ.ccx(0,N-n+2,N-n+1)
            circ.ccx(0,N-n+2,2*N-n+2)
    S(circuit,N)
    return circuit
```

Fig. 12. N-bit conditional adder program written in Qiskit language.

$$|0\rangle|A[n]\rangle|B[n]\rangle \mapsto |c_0\rangle|A\rangle|B-A\rangle$$

This specification is similar to the specification for Cuccaro's Adder (Equation 7.1) except for changing $B+A$ to $B-A$. With this specification, QSynth generates the program shown in Fig 11. This program equals to the circuit shown in Fig 13, indicating the logical expression $B-A=\overline{\bar{B}+A}$, which is different from the traditional subtractor implementation. This subtractor generated by QSynth uses no ancilla qubits and saves quantum resources.

7.3 Quantum Conditional Adder

Motivation and Background. Quantum Conditional Adder is a necessary arithmetic operation for many known quantum algorithms, including quantum multiplier and Thapliyal et al. [2019]'s quantum long division algorithm. A n-bit Quantum conditional adder circuit implements the transformation $|ctrl\rangle|0\rangle|A\rangle|B\rangle \mapsto |ctrl\rangle|ctrl*c0\rangle|A\rangle|ctrl*A+B\rangle$. It sums A_n and B_n when the control qubit $|ctrl\rangle$ is in state $|1\rangle$ and keeps the state unchanged when $|ctrl\rangle$ is in state $|0\rangle$.

One way to construct such a program is by replacing each gate in Cuccaro's adder program (i.e., the program in Fig 10(b)) with its conditional version, which is also the circuit generated by Qiskit. However, this method needs four-qubit Toffoli gates, which needs 14 CNOT gates to implement, increasing the total count of CNOT gates in the decomposed circuit. We let QSynth synthesize a target program using only X gate, CNOT gate, and Toffoli gate to find a better solution.

Synthesis with QSynth. We let QSynth synthesize a program with the specification below

$$|f[1]\rangle|0\rangle|A[n]\rangle|B[n]\rangle \mapsto |f\rangle|f*c_0\rangle|A\rangle|f*A+B\rangle$$

The term $f*A+B$ indicates qubit q_0 is the flag qubit. With this specification, QSynth generates program S shown in Fig 12. We compare the resource count between the conditional adder circuits generated by QSynth and Qiskit, which is shown in Fig 14. All circuits are decomposed with $\{u_3, CNOT\}$ gateset by Qiskit's decomposition pass for comparison, where u_3 is a generic single-qubit rotation gate. The conditional adder programs generated by QSynth always use fewer quantum resources compared to the one from Qiskit.

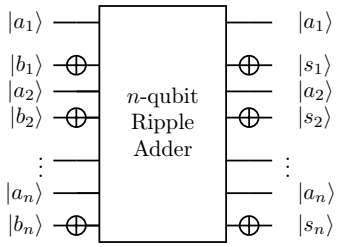


Fig. 13. N-bit ripple subtractor circuit

Qubit	QSynth		Qiskit	
	u_3	cnot	u_3	cnot
4	182	144	244	208
5	228	180	305	260
6	274	216	366	312
7	320	252	427	364

Fig. 14. Comparison of resource count between the conditional adder circuit generated by QSynth and Qiskit.

7.4 Eigenvalue Inversion

Background and Motivation. Eigenvalue inversion is a necessary arithmetic step in HHL[Lloyd 2010], a quantum algorithm for linear systems of equations. Given a c-qubit eigenvalue state $|\lambda\rangle$, the eigenvalue inversion circuit needs to calculate $|1/\lambda\rangle$. In practice, only the first n decimal places of the $1/\lambda$, denoted as d_n, and the remainder r_0 are kept. The precision n depends on the accuracy requirement of the algorithm.

Synthesis with QSynth. We let QSynth synthesize a program that can calculate the first n decimal places of $1/\lambda$ and keep the remainder r_0 for further use.

Since QSynth's specification syntax only supports binary integers, we use the fact $2^n = \lambda * d_n + r_0$ where d_n is the quotient we want to give the specification. The example below shows our intuition.

Base 10: $1/7 = 0.142857142857... \Leftrightarrow 10^6/7 = 142857.142857... \quad \Leftrightarrow 10^6 = 142857 * 7 + r_0$

Base 2: $1/7 = 0.001001001001... \Leftrightarrow 2^6/7 = 001001.001001... \quad \Leftrightarrow 2^6 = (001001)_2 * 7 + r_0$

The specification we send to QSynth for the n-bit precision eigenvalue inversion program is

$$|0\rangle|\Lambda[c]\rangle|0_n\rangle|1\rangle|0_{c-1}\rangle \mapsto |0\rangle|\Lambda\rangle|2^n\%\Lambda\rangle_c|2^n/\Lambda\rangle$$

The specification uses $\Lambda[c]$ to represent the signed input c-bit eigenvalue λ. It suggests qubit $q_{c+1} \sim q_{2c}$ store the remainder (i.e. $|2^n\%\Lambda\rangle$) and qubit $q_{2c+1} \sim q_{2c+n}$ store the quotient (i.e. $|2^n/\Lambda\rangle$).

With this specification, QSynth generates the program shown in Fig 15. Fig 16 shows the corresponding circuit for $n = 3, c = 3$. This program is a variant of Thapliyal [Thapliyal et al. 2019]'s general quantum division circuit. Compared to calculating $|1/\lambda\rangle$ with Thapliyal's division circuit, which is constructed by $max(n, c)$-qubit subtractor and conditional adder, this program uses c-qubit one. This significantly reduces the number of qubits required when n is large.

7.5 Quantum Fourier Transform

Motivation and Background. Quantum Fourier Transform (QFT) [Coppersmith 2002] is the classical discrete Fourier Transform applied to the vector of amplitudes of a quantum state. QFT is a part of many quantum algorithms, notably Shor's algorithm [Shor 1994], QPE algorithm [Kitaev 1995], and algorithms for the hidden subgroup problem [Ettinger and Hoyer 1999]. A QFT over $\mathbb{Z}_{2^n}$ can

```
def inversion(N):
  circ = QuantumCircuit(N+2*c+1)
  def S(circ, n):
    if n<1:
      pass
    else:
      Append(circ,SUB(c),0,1,c+n+1)
      Append(circ,C_ADD(c-1),
        2*c+n,c,c+n)
      circ.x(2*c+n)
      circ=S(circ,n-1)
    return circ
  circ=S(circ,N)
  return circ
```

Fig. 15. N-bit precision eigenvalue inversion program.

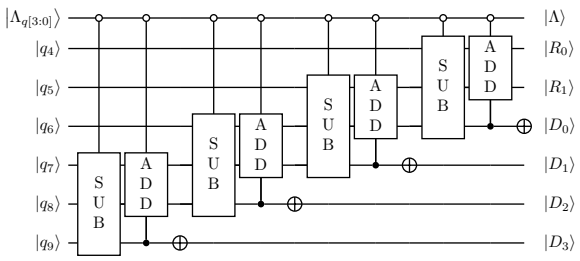


Fig. 16. Eigenvalue inversion circuit for $n = 3, c = 3$.

be expressed as a map in two equivalent forms:

$$\text{QFT: } |x\rangle \mapsto \frac{1}{\sqrt{2^n}} \sum_{y=0}^{2^n-1} e^{\frac{2\pi i \cdot xy}{2^n}} |y\rangle \qquad \text{OR} \qquad \text{QFT: } |x\rangle \mapsto \bigotimes_{k=0}^{n-1} |z_k\rangle \tag{7.2}$$

$$|z_k\rangle = \frac{1}{\sqrt{2}}(|0\rangle + e^{2\pi i \cdot [0.x_k x_{k-1} \ldots x_0]} |1\rangle), \qquad [0.x_m \ldots x_1 x_0] = \sum_{k=0}^{m} \frac{x_k}{2^{m-k+1}}. \tag{7.3}$$

Synthesis with QSynth. We use the specification in Equation 4.2 to synthesize a program S_Q that $S_Q(n)$ returns the $n+1$-qubit QFT circuit.

QSynth fails to synthesize a simple fixpoint structure QFT circuit. Therefore we synthesize it in two steps to help QSynth synthesize a nested structure circuit. First, 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. The specification is:

$$|X[n+1]\rangle \mapsto |X_0 X_1 \ldots X_{n-1}\rangle \otimes (|0\rangle \uplus e^{\frac{2\pi i \cdot X}{2^n}} \cdot |1\rangle)$$

With this specification, the synthesizer generates the program as shown in Fig 17. Statement `circ.cp(pi/2**n, N-n, N)` in Qiskit means a controlled phase rotation gate R_n[2] on qubit q_N controlled by qubit q_{N-n}.

Then 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 in Fig 18.

7.6 n-qubit Quantum Teleportation

Motivation and Background. Quantum teleportation[Bennett et al. 1993] is one of the most famous quantum applications that can be implemented in the near term. It is a technique for transferring quantum information from a sender at one location to a receiver some distance away. The sender does not have to know the particular quantum state being transferred. Moreover, the recipient's location can be unknown, but to complete the quantum teleportation, classical information needs to be sent from sender to receiver. The right-hand side circuit shows the process for sending one-qubit state $|\phi\rangle$ from Alice to Bob.

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

```
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
```

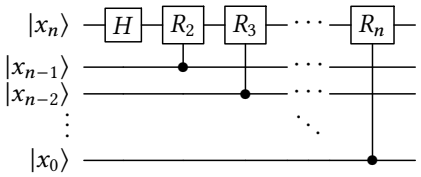


Fig. 17. Qiskit program Zn that transforms qubit q_n to state $|z_n\rangle$

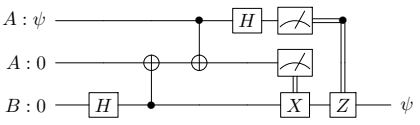
```
def QFT(N):
    circuit = QuantumCircuit(N+1)
    def S(circ,n):
        if n == 0:
            circ.h(n)
        else:
            circ.append(Zn(n))
            S(circ, n-1)
    S(circuit, N)
    return circuit
```

Fig. 18. N + 1-bit QFT program written in Qiskit language.

```
def teleport(N):
    circuit=QuantumCircuit(3*N)
    def S(circ, n):
        if(n<0):
            return
        else:
            circ.h(N+n)
            circ.cx(N+n, 2*N+n)
            circ.cx(n, N+n)
            circ.h(n)
            S(circ,n-1)
    S(circuit,N)
    return circuit
```

Fig. 19. N-qubit quantum teleportation program.

Synthesis with QSynth. We let QSynth synthesize the unitary circuit part (before the measurement) of the n-qubit quantum teleportation process. Suggest Alice wants to send a state $|\phi_n\rangle$ stored in n data qubits to Bob, and they each have n more ancilla qubits which are initialized to state $|0\rangle^n$ for the teleportation process. We follow the same measurement strategy as the one-qubit teleportation: Alice will measure $|\phi_n\rangle$ and Alice's n aniclla qubits after the unitary circuit, then Bob apply bit-wise CZ and CX gate to Bob's n ancilla qubits based on the result. Assume the data qubits, Alice's ancilla qubits and Bob's ancilla qubits are in state $|a_n\rangle, |b_n\rangle, |c_n\rangle$ respectively after the measurement, Bob can use bit-wise CZ and CX gate to reproduce the state $|\phi_n\rangle$ if $(-1)^{\oplus a_n}(|b_n\rangle \oplus |c_n\rangle) = |\phi_n\rangle$. Here $|b_n\rangle \oplus |c_n\rangle$ is bit-wise XOR operation (e.g. $|101\rangle \oplus (|100\rangle + |110\rangle) = |001\rangle + |011\rangle$) and $\oplus a_n$ is a reduction XOR operation on a_n (e.g. $\oplus 101 = 1 \oplus 0 \oplus 1 = 0$). With this intuition, we let QSynth use the specification below to synthesize the unitary circuit part of a n-qubit quantum teleportation program,



$$|\phi[n]\rangle|0_n\rangle|0_n\rangle \mapsto \bigl\lfloor + \bigr\rfloor_{z\in\{0,1\}^{3n}} \delta(\phi = b_n \oplus c_n)\cdot e^{-\pi i\cdot(\oplus a_n)}|z\rangle$$

where $a_n = z[n-1:0], b_n = z[2n-1:n], c_n = z[3n-1, 2n]$. Fig 19 shows the program generated by QSynth.

7.7 Performance Evaluation

In this section, we compare the performance of QSynth and previous circuit synthesis methods.

Implementation. In the experiment, we use the Syntax-Guided Top-down Tree Search [Gulwani et al. 2017] as the searcher with the following bounds on the search space: (1) when searching a candidate program under ISQIR syntax in Definition 5.1, we set the maximum program length to 10 and enumerate the value of k in the FIX syntax in $\{1, 2, 3\}$; (2) shorter candidate programs are sent to the verifier first; (3) when searching a permutation π under the syntax in Definition 5.3, we set the maximum syntax derivation depth to 4; (4) when deriving the syntax rule $e ::= m, m \in \mathbb{N}$, we enumerate $m \in [0, 3]$. All benchmarks in this paper can be synthesized under these bounds.

The implementation of QSynth uses 1k lines of Python. All the experiments are run with Z3 solver version 4.8.9 and Python 3.8.

Benchmarks. Table 2 summarizes all 10 benchmarks. They are in three categories: arithmetic circuits, state preparation and sub-programs widely used in quantum algorithms. Many benchmarks are collected from textbooks [Nielsen and Chuang 2010]. Arithmetic circuits are frequently used in quantum oracle designs which is necessary for most famous quantum algorithms (e.g. Simons, Shor's algorithms, Grover search algorithm). State preparation is necessary for the setup of many quantum applications (e.g. quantum teleportation). We also collect the necessary quantum sub-program used in quantum algorithms from their paper (e.g. HHL algorithm).

Table 2. Summary of all benchmarks used in the evaluation.

Benchmark Type	Benchmark Name	Description	Gate Set
State Preparation	n-GHZ	Greenberger–Horne–Zeilinger state [Greenberger et al. 1989]	H,CX
	n-Uniform	n-qubit uniform distribution state	H, X, Y, Z
Arithmetic	n-full-Add	n-qubit full adder	QFT, CX, SWAP, CCX, X
	n-Add	n-qubit in place adder	MAJ, UMA, QFT, CS, CZ, X
	n-Sub	n-qubit in place subtractor	MAJ, UMA, QFT, CS, CZ, X
	Cond n-Add	n-qubit conditional in place adder	Toffoli, QFT, CX, CS, CZ, X
Algorithm Module	n-QFT	n-qubit Quantum Fourier Transform	H,CS,CT, SW AP
	Inversion	n-qubit precision eigenvalue inversion for HHL algorithm [Lloyd 2010]	c-adder, subtractor,X
	n-Toff	n-qubit Toffoli gate	Toffoli, CX, X
	n-Teleport	n-qubit quantum teleportation	H, CX

We compare the performance of QSynth against QFAST [Younis et al. 2021] and Qsyn [Kang and Oh 2023]. For QSynth, we use input-output style specification written in QSynth-spec as input. For the other two frameworks, we use their specification interfaces and try to synthesize circuits with $n = 3, 4, 5, 6$. We use the same gate set when comparing QSynth and Qsyn in each case, while for QFast we use its hard-coded gate set.

We stop the synthesis and regard the synthesis as a failure if the running time is over 1 hour. All runtimes are a median of three runs.

Table 3 shows the running time of all the experiments. We can see that QSynth successfully synthesizes programs in all 10 benchmarks in at most 5 minutes, while QFAST and Qsyn fail to

Table 3. Running time of all benchmarks used in the evaluation.

Benchmark Name	QSynth time (s)	QFAST time (s) 3	4	5	6	Qsyn time (s) 3	4	5	6
n-GHZ	1.793	0.122	2.48	59.1	-*	0.091	1.33	49.2	-*
n-Uniform	0.415	0.027	1.25	32.7	-*	0.096	1.40	39.8	-*
n-Full-Add	288.1	617	-*	-*	-*	812.4	-*	-*	-*
n-Add	170.4	942	3511	-*	-*	132	2818	-*	-*
n-Sub	168.6	667	-*	-*	-*	287	3240	-*	-*
Cond n-Add	185.2	851	-*	-*	-*	192	1804	-*	-*
n-Toff	66.7	<0.01	21.8	679	-*	<0.01	16.5	354	-*
n-QFT	145.49	21.4	397	3598	-*	67.2	473	-*	-*
Inversion	93.5	1622	-*	-*	-*	22.5	740	-*	-*
n-Teleport	185.1	897	-*	-*	-*	614	-*	-*	-*

* Time out after 1 hour.

synthesize circuits for $n = 6$. From the result, we can see that when the size increases, the time of QFAST and Qsyn indeed grow exponentially, while QSynth only pays a fixed cost for all sizes.

The synthesis time of QSynth is comparable to the time to synthesize a corresponding circuit with size 3, with an exception of the n-Toff benchmark. We note that this is because 3-Toffoli is exactly a single Toffoli gate and 4-Toffoli can be done using two Toffoli gates, which are straightforward for QFAST and Qsyn to search. In contrast, QSynth's search is longer because it needs to consider the inductive structure and corner cases of $n = 1$ and 2. Nevertheless, QSynth quickly outperforms other frameworks on n-Toff at $n = 5$.

8 RELATED WORK

Synthesis of quantum circuits. Many methods have been proposed to synthesis quantum circuits of a fixed size [Amy et al. 2013; de Brugiere 2020; Deng et al. 2023a; Kang and Oh 2023; Kitaev 1997; Saeedi et al. 2011; Shende et al. 2006; Xu et al. 2023; Younis et al. 2021]. These methods do not consider the inductive structure of quantum programs and do not scale due to their exponential blowup with the number of qubits.

Synthesis of classical programs. The tasks of synthesizing classical programs are intensively-studied in the recent decades [Gulwani et al. 2017; Kitzelmann 2009]. The problem definitions of program synthesis are diverse and orienting, including syntax-guided synthesis [Alur et al. 2013, 2018; Hu and D'Antoni 2018; Jha et al. 2010], example-guided synthesis [Gulwani 2011; Gulwani et al. 2012; Polozov and Gulwani 2015], semantics-guided synthesis [D'Antoni et al. 2021; Kim et al. 2021], and resource-guided synthesis [Hu et al. 2021; Knoth et al. 2019]. The modern approaches to solve these problems make use of sophisticated search algorithms, such as enumerative search with pruning [Gulwani et al. 2011; Phothilimthana et al. 2016], constraint solver like satisfiability modulo theory (SMT) solver [Feng et al. 2017; Jha et al. 2010; Solar-Lezama 2008], and machine learning [Liang et al. 2010; Menon et al. 2013]. We refer curious readers to surveys [Gulwani et al. 2017; Kitzelmann 2009] for a comprehensive picture on the development of classical program synthesis techniques. Many synthesis frameworks are developed into productive tools. For example, the Sketch [Solar-Lezama 2008] framework completes programs with holes by specifications. Rosette [Torlak and Bodik 2013] builds solvers into the language to automatically fill in holes when programming. However, QSynth needs to deal with unique challenges from quantum programs.

Verification of 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. QWIRE [Paykin et al. 2017] embedded the formal verification of quantum programs manually in the Coq proof assistant. QBricks [Chareton et al. 2020] do formal verification of quantum programs semi-manually using Why3. Quantum abstract interpretation [Yu and Palsberg 2021] provides efficient tools to test the properties of quantum programs. In particular, the path-sum representation [Amy 2018] of quantum program semantics inspired our representation. Chen et al. [2023] uses tree automaton to verify fixed-size quantum circuits. It is hard to generalize to general-size cases because graphical structures like automaton are hard to symbolically model in SMT solvers.

Quantum Hoare logic [Ying 2012] uses quantum predicates and Hoare triples to express and derive properties of quantum programs. Its language, quantum while language does not have the detailed structure of unitary executions. Its specifications are quantum predicate matrices. Therefore, it cannot be applied to synthesize quantum unitary circuit families.

The use of SMT solvers to automate the reasoning has also appeared in Giallar [Tao et al. 2022], Quartz [Xu et al. 2022] and symQV [Bauer-Marquart et al. 2023], although they only work on quantum circuit compilation passes, quantum circuit optimizations or fixed-size quantum circuit verification.

9 DISCUSSION AND FUTURE WORKS

QSynth comes with several limitations. At a high level, QSynth sacrifices the expressiveness of the specification due to the limitation in efficient SMT encodings. For example, the lack of the equivalence verification of general complex functions forces us to consider the special form of amplitude in Definition 6.1. The sparsity requirement of α is another such restriction. Any relief of such restrictions would enlarge the space of programs that can be synthesized by QSynth.

Another limitation is that QSynth cannot directly synthesize programs involving nested fixed-point structures. Synthesizing a program nested loop or fixed-point structure is also a challenging problem in the classical domain. This is because the loop invariant of the program in nested loop structures is unknown and usually non-trivial to figure out. So we make the current QSynth only expand the FIX syntax at most once when generating the candidate program and avoid directly sending a program in nested fixed-point structure to the QSynth verifier. A natural next step is to include the search for the loop invariant in nested loop structures as part of QSynth, and to verify the candidate program against both the specification and the generated loop invariant.

Many quantum algorithms such as Bernstein-Vazirani [Bernstein and Vazirani 1997] and Deustch-Jozsa [Deutsch and Jozsa 1992] have quantum oracle as part of the program. However, it is hard to synthesize and verify quantum programs with oracles because it requires higher-order logic to quantify over arbitrary oracles. It is also an interesting next step to extend QSynth to support quantum oracles.

The design of ISQIR inherits concrete qubit indices from SQIR. This design introduces complications in the inductive variant that have to be addressed with explicit permutations and relabelings. We will explore the possibility of using a different representation of variables to make synthesis less complicated in the future.

Besides extending the expressiveness of specification and the support of more complicated quantum programs, it is also interesting to improve the synthesis performance by integrating classical program synthesis techniques into the quantum domain, including counterexample-guided synthesis, and various search heuristics.

10 CONCLUSION

We present QSynth, the first 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 seven quantum unitary programs and QSynth can generate programs better than the standard solutions in the quantum subtractor and quantum conditional adder cases. These programs can be readily transpiled to executable programs on major quantum platforms, e.g., Q#, IBM Qiskit.

QSynth constitutes the first step toward a fully automated quantum program synthesis framework, which can 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.

ACKNOWLEDGMENTS

We thank anonymous reviewers for constructive suggestions that improve the presentation of the paper. H.D., Y.P., and X.W. was partially funded by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Quantum Testbed Pathfinder Program under Award Number DE-SC0019040, Air Force Office of Scientific Research under award number FA9550-21-1-0209, the U.S. National Science Foundation grant CCF-1942837 (CAREER) and a Sloan research fellowship.

DATA-AVAILABILITY STATEMENT

QSynth is available from Zenodo DOI 10.5281/zenodo.10054966 [Deng et al. 2023b] and the latest version from Github

REFERENCES

Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo MK Martin, Mukund Raghothaman, Sanjit A Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. *Syntax-guided synthesis.* IEEE. https://doi.org/10.1109/FMCAD.2013.6679385

Rajeev Alur, Rishabh Singh, Dana Fisman, and Armando Solar-Lezama. 2018. Search-based program synthesis. *Commun. ACM* 61, 12 (2018), 84–93. https://doi.org/10.1145/3208071

Matthew Amy. 2018. Towards large-scale functional verification of universal quantum circuits. *arXiv preprint arXiv:1805.06908* (2018). https://doi.org/10.4204/eptcs.287.1

Matthew Amy, Parsiad Azimzadeh, and Michele Mosca. 2018. On the controlled-NOT complexity of controlled-NOT–phase circuits. *Quantum Science and Technology* 4, 1 (2018), 015002. https://doi.org/10.1088/2058-9565/aad8ca

Matthew Amy, Dmitri Maslov, and Michele Mosca. 2014. 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. https://doi.org/10.1109/TCAD.2014.2341953

Matthew Amy, Dmitri Maslov, Michele Mosca, and Martin Roetteler. 2013. 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. https://doi.org/10.1109/TCAD.2013.2244643

Matthew Amy and Michele Mosca. 2019. T-count optimization and Reed–Muller codes. *IEEE Transactions on Information Theory* 65, 8 (2019), 4771–4784. https://doi.org/10.1109/TIT.2019.2906374

Dave Bacon, Wim van Dam, and Alexander Russell. 2008. Analyzing algebraic quantum circuits using exponential sums. *unpublished: see tinyurl. com/qpo7s2* (2008).

Fabian Bauer-Marquart, Stefan Leue, and Christian Schilling. 2023. symQV: Automated Symbolic Verification of Quantum Programs. In *Formal Methods*, Marsha Chechik, Joost-Pieter Katoen, and Martin Leucker (Eds.). Springer International Publishing, Cham, 181–198. https://doi.org/10.1007/978-3-031-27481-7_12

Charles H Bennett, Gilles Brassard, Claude Crépeau, Richard Jozsa, Asher Peres, and William K Wootters. 1993. Teleporting an unknown quantum state via dual classical and Einstein-Podolsky-Rosen channels. *Physical review letters* 70, 13 (1993), 1895. https://doi.org/10.1103/PhysRevLett.70.1895

Ethan Bernstein and Umesh Vazirani. 1997. Quantum Complexity Theory. *SIAM J. Comput.* 26, 5 (1997), 1411–1473. https://doi.org/10.1137/S0097539796300921 arXiv:https://doi.org/10.1137/S0097539796300921

Sergey Bravyi and David Gosset. 2016. Improved classical simulation of quantum circuits dominated by Clifford gates. *Physical review letters* 116, 25 (2016), 250501. https://doi.org/10.1103/PhysRevLett.116.250501

Christophe Chareton, Sbastien Bardin, Franccois Bobot, Valentin Perrelle, and Benoit Valiron. 2020. A Deductive Verification Framework for Circuit-building Quantum Programs. *arXiv preprint arXiv:2003.05841* (2020). https://doi.org/10.1007/978-3-030-72019-3_6

Yu-Fang Chen, Kai-Min Chung, Ondrej Lengál, Jyun-Ao Lin, Wei-Lun Tsai, and Di-De Yen. 2023. An Automata-based Framework for Verification and Bug Hunting in Quantum Circuits. *Proceedings of the ACM on Programming Languages* 7, PLDI (2023), 1218–1243. https://doi.org/10.1145/3591270

Don Coppersmith. 2002. An approximate Fourier transform useful in quantum factoring. *arXiv preprint quant-ph/0201067* (2002).

Steven A Cuccaro, Thomas G Draper, Samuel A Kutin, and David Petrie Moulton. 2004. A new quantum ripple-carry addition circuit. *arXiv preprint quant-ph/0410184* (2004). https://doi.org/10.48550/arXiv.quant-ph/0410184

Christopher M Dawson and Michael A Nielsen. 2005. The solovay-kitaev algorithm. *arXiv preprint quant-ph/0505030* (2005). https://doi.org/10.26421/QIC6.1-6

Timothee Goubault de Brugiere. 2020. *Methods for optimizing the synthesis of quantum circuits.* Ph. D. Dissertation. Universite Paris-Saclay.

Haowei Deng, Yuxiang Peng, Michael Hicks, and Xiaodi Wu. 2023a. Automating NISQ Application Design with Meta Quantum Circuits with Constraints (MQCC). *ACM Transactions on Quantum Computing* 4, 3 (2023), 1–29. https://doi.org/10.1145/3579369

Haowei Deng, Zhoutao Run, Yuxiang Peng, and Xiaodi Wu. 2023b. QSynth Zenodo. (2023). https://doi.org/10.5281/zenodo.10054966

David Deutsch and Richard Jozsa. 1992. Rapid solution of problems by quantum computation. *Proceedings of the Royal Society of London. Series A: Mathematical and Physical Sciences* 439, 1907 (1992), 553–558. https://doi.org/10.1098/rspa.1992.0167

Menghan Dou, Tianrui Zou, Yuan Fang, Jing Wang, Dongyi Zhao, Lei Yu, Boying Chen, Wenbo Guo, Ye Li, Zhaoyun Chen, et al. 2022. QPanda: high-performance quantum computing framework for multiple application scenarios. *arXiv preprint arXiv:2212.14201* (2022). https://doi.org/10.48550/arXiv.2212.14201

Loris D'Antoni, Qinheping Hu, Jinwoo Kim, and Thomas Reps. 2021. Programmable program synthesis. In *International Conference on Computer Aided Verification*. Springer, 84–109. https://doi.org/10.1007/978-3-030-81685-8_4

Mark Ettinger and Peter Hoyer. 1999. A quantum observable for the graph isomorphism problem. *arXiv preprint quant-ph/9901029* (1999). https://doi.org/10.48550/arXiv.quant-ph/9901029

Yu Feng, Ruben Martins, Yuepeng Wang, Isil Dillig, and Thomas W Reps. 2017. Component-based synthesis for complex APIs. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. 599–612. https://doi.org/10.1145/3009837.3009851

Richard P Feynman. 1985. Quantum mechanical computers. *Optics news* 11, 2 (1985), 11–20. https://doi.org/10.1007/BF01886518

Jay Gambetta. 2022. IBM Quantum Roadmap to build quantum-centric supercomputers. https://research.ibm.com/blog/ibm-quantum-roadmap-2025

Daniel M Greenberger, Michael A Horne, and Anton Zeilinger. 1989. Going beyond Bell's theorem. (1989), 69–72. https://doi.org/10.1007/978-94-017-0849-4_10

Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. *ACM Sigplan Notices* 46, 1 (2011), 317–330. https://doi.org/10.1145/1926385.1926423

Sumit Gulwani, William R. Harris, and Rishabh Singh. 2012. Spreadsheet Data Manipulation Using Examples. *Commun. ACM* 55, 8 (aug 2012), 97–105. https://doi.org/10.1145/2240236.2240260

Sumit Gulwani, Vijay Anand Korthikanti, and Ashish Tiwari. 2011. Synthesizing geometry constructions. *ACM SIGPLAN Notices* 46, 6 (2011), 50–61. https://doi.org/10.1145/1993316.1993505

Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. 2017. Program Synthesis. *Foundations and Trends® in Programming Languages* 4, 1-2 (2017), 1–119. https://doi.org/10.1561/2500000010

Kesha Hietala, Robert Rand, Shih-Han Hung, Xiaodi Wu, and Michael Hicks. 2021. A Verified Optimizer for Quantum Circuits. *Proc. ACM Program. Lang.* 5, POPL, Article 37 (Jan. 2021), 29 pages. https://doi.org/10.1145/3434318

Mark Hillery, Vladimir Buzek, and André Berthiaume. 1999. Quantum secret sharing. *Physical Review A* 59, 3 (1999), 1829. https://doi.org/10.1103/PhysRevA.59.1829

Qinheping Hu, John Cyphert, Loris D'Antoni, and Thomas Reps. 2021. Synthesis with asymptotic resource bounds. In *International Conference on Computer Aided Verification*. Springer, 783–807. https://doi.org/10.1007/978-3-030-81685-8_37

Qinheping Hu and Loris D'Antoni. 2018. Syntax-guided synthesis with quantitative syntactic objectives. In *International Conference on Computer Aided Verification*. Springer, 386–403. https://doi.org/10.1007/978-3-319-96145-3_21

Susmit Jha, Sumit Gulwani, Sanjit A Seshia, and Ashish Tiwari. 2010. Oracle-guided component-based program synthesis. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, Vol. 1. IEEE, 215–224. https://doi.org/10.1145/

1806799.1806833

Chan Gu Kang and Hakjoo Oh. 2023. Modular Component-Based Quantum Circuit Synthesis. *Proceedings of the ACM on Programming Languages* 7, OOPSLA1 (2023), 348–375. https://doi.org/10.1145/3586039

Jinwoo Kim, Qinheping Hu, Loris D'Antoni, and Thomas Reps. 2021. Semantics-guided synthesis. *Proceedings of the ACM on Programming Languages* 5, POPL (2021), 1–32. https://doi.org/10.1145/3434311

A Yu Kitaev. 1995. Quantum measurements and the Abelian stabilizer problem. *arXiv preprint quant-ph/9511026* (1995). https://doi.org/10.48550/arXiv.quant-ph/9511026

A Yu Kitaev. 1997. Quantum computations: algorithms and error correction. *Russian Mathematical Surveys* 52, 6 (1997), 1191. https://doi.org/10.1070/RM1997v052n06ABEH002155

Emanuel Kitzelmann. 2009. Inductive programming: A survey of program synthesis techniques. In *International workshop on approaches and applications of inductive programming*. Springer, 50–73. https://doi.org/10.1007/978-3-642-11931-6_3

Tristan Knoth, Di Wang, Nadia Polikarpova, and Jan Hoffmann. 2019. Resource-guided program synthesis. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 253–268. https://doi.org/10.1145/3314221.3314602

Dax Enshan Koh, Mark D Penney, and Robert W Spekkens. 2017. Computing quopit Clifford circuit amplitudes by the sum-over-paths technique. *Quantum Information & Computation* (2017). https://doi.org/10.26421/QIC17.13-14-1

Percy Liang, Michael I Jordan, and Dan Klein. 2010. Learning programs: A hierarchical Bayesian approach. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*. 639–646.

Ci-Hong Liao, Chun-Wei Yang, and Tzonelish Hwang. 2014. Dynamic quantum secret sharing protocol based on GHZ state. *Quantum information processing* 13, 8 (2014), 1907–1916. https://doi.org/10.1007/s11128-014-0779-x

Chia-Chun Lin, Amlan Chakrabarti, and Niraj K Jha. 2014. Qlib: Quantum module library. *ACM Journal on Emerging Technologies in Computing Systems (JETC)* 11, 1 (2014), 1–20. https://doi.org/10.1145/2629430

Seth Lloyd. 2010. Quantum algorithm for solving linear systems of equations. In *APS March Meeting Abstracts*, Vol. 2010. D4–002. https://doi.org/10.1103/PhysRevLett.103.150502

Aditya Menon, Omer Tamuz, Sumit Gulwani, Butler Lampson, and Adam Kalai. 2013. A machine learning framework for programming by example. In *International Conference on Machine Learning*. PMLR, 187–195.

Ashley Montanaro. 2017. Quantum circuits and low-degree polynomials over. *Journal of Physics A: Mathematical and Theoretical* 50, 8 (2017), 084002. https://doi.org/10.1088/1751-8121/aa565f

Michael A. Nielsen and Isaac L. Chuang. 2010. *Quantum Computation and Quantum Information: 10th Anniversary Edition*. Cambridge University Press. https://doi.org/10.1017/CBO9780511976667

Jennifer Paykin, Robert Rand, and Steve Zdancewic. 2017. QWIRE: a core language for quantum circuits. *ACM SIGPLAN Notices* 52, 1 (2017), 846–858. https://doi.org/10.1145/3009837.3009894

Phitchaya Mangpo Phothilimthana, Aditya Thakur, Rastislav Bodik, and Dinakar Dhurjati. 2016. Scaling up superoptimization. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*. 297–310. https://doi.org/10.1145/2872362.2872387

Oleksandr Polozov and Sumit Gulwani. 2015. Flashmeta: A framework for inductive program synthesis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 107–126. https://doi.org/10.1145/2858965.2814310

Mehdi Saeedi, Robert Wille, and Rolf Drechsler. 2011. Synthesis of quantum circuits for linear nearest neighbor architectures. *Quantum Information Processing* 10, 3 (2011), 355–377. https://doi.org/10.1007/s11128-010-0201-2

Vivek V Shende, Stephen S Bullock, and Igor L Markov. 2006. Synthesis of quantum-logic circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 25, 6 (2006), 1000–1010. https://doi.org/10.1109/TCAD.2005.855930

Peter W Shor. 1994. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th annual symposium on foundations of computer science*. Ieee, 124–134. https://doi.org/10.1109/SFCS.1994.365700

Armando Solar-Lezama. 2008. *Program synthesis by sketching*. University of California, Berkeley.

Runzhou Tao, Yunong Shi, Jianan Yao, Xupeng Li, Ali Javadi-Abhari, Andrew W Cross, Frederic T Chong, and Ronghui Gu. 2022. Giallar: push-button verification for the qiskit Quantum compiler. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 641–656. https://doi.org/10.1145/3519939.3523431

Himanshu Thapliyal, Edgard Munoz-Coreas, TSS Varun, and Travis S Humble. 2019. Quantum circuit designs of integer division optimizing T-count and T-depth. *IEEE transactions on emerging topics in computing* 9, 2 (2019), 1045–1056. https://doi.org/10.1109/iNIS.2017.34

Emina Torlak and Rastislav Bodik. 2013. Growing solver-aided languages with Rosette. In *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*. 135–152. https://doi.org/10.1145/2509578.2509586

Yan Xia, Chang-Bao Fu, Shou Zhang, Suc-Kyoung Hong, Kyu-Hwang Yeon, and Chung-In Um. 2006. Quantum dialogue by using the GHZ state. *arXiv preprint quant-ph/0601127* (2006). https://doi.org/10.48550/arXiv.quant-ph/0601127

Amanda Xu, Abtin Molavi, Lauren Pick, Swamit Tannu, and Aws Albarghouthi. 2023. Synthesizing Quantum-Circuit Optimizers. *Proceedings of the ACM on Programming Languages* 7, PLDI (2023), 835–859. https://doi.org/10.1145/3591254

Mingkuan Xu, Zikun Li, Oded Padon, Sina Lin, Jessica Pointing, Auguste Hirth, Henry Ma, Jens Palsberg, Alex Aiken, Umut A Acar, et al. 2022. Quartz: superoptimization of Quantum circuits. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 625–640. https://doi.org/10.1145/3519939.3523433

Mingsheng Ying. 2012. Floyd–hoare logic for quantum programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 33, 6 (2012), 1–49. https://doi.org/10.1145/2049706.2049708

Ed Younis, Koushik Sen, Katherine Yelick, and Costin Iancu. 2021. Qfast: Conflating search and numerical optimization for scalable quantum circuit synthesis. In *2021 IEEE International Conference on Quantum Computing and Engineering (QCE)*. IEEE, 232–243. https://doi.org/10.1109/QCE52317.2021.00041

Nengkun Yu and Jens Palsberg. 2021. Quantum abstract interpretation. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 542–558. https://doi.org/10.1145/3453483.3454061

Man Zhong-Xiao and Xia Yun-Jie. 2006. Controlled bidirectional quantum direct communication by using a GHZ state. *Chinese Physics Letters* 23, 7 (2006), 1680. https://doi.org/10.1088/0256-307X/23/7/007

Received 2023-07-11; accepted 2023-11-07